A Toolkit for Analysis of Deep Learning Experiments*

Jim O' Donoghue[1] and Mark Roantree[1]

Insight Centre for Data Analytics, School of Computing,
Dublin City University, Dublin, IRELAND
`jodonoghue, mroantree@computing.dcu.ie`

Abstract. Learning experiments are complex procedures which generate high volumes of data due to the number of updates which occur during training and the number of trials necessary for hyper-parameter selection. Often during runtime, interim result data is purged as the experiment progresses. This purge makes rolling-back to interim experiments, restarting at a specific point or discovering trends and patterns in parameters, hyper-parameters or results almost impossible given a large experiment or experiment set. In this research, we present a data model which captures all aspects of a deep learning experiment and through an application programming interface provides a simple means of storing, retrieving and analysing parameter settings and interim results at any point in the experiment. This has the further benefit of a high level of interoperability and sharing across machine learning researchers who can use the model and its interface for data management.

1 Introduction

In order to tune and optimise machine learning models, a wide range of parameters are required. Finding the best combination of parameters is often complex and time consuming, as parameter optimisation requires careful monitoring of each batch of results, which are generated during an update in training. These results should also be monitored with respect to different combinations of hyper-parameters. Hyper-parameters (HPs) are those not learned by the model but instead given as inputs to the algorithm before training. One needs to choose a set of HPs which allow model parameters to reach a configuration that optimises a particular performance goal on a dataset for an algorithm during training. Grid and manual search are the most widely used strategies for HP optimisation and in both cases, many HP configurations are run to view their effect on algorithm training in order to determine the best parameters. The main issues in trying to find *good* parameter settings can be listed as follows:

– Functions require many complete iterations of training to find the optimal hyper-parameter configuration - often a manual and lengthy process which can lack empirical rigour.

* Research funded by Science Foundation Ireland, grant number SFI/12/RC/2289.

- In the majority of experiments interim results are stored in-memory and subsequently discarded, save for the final, most accurate learner(s), result(s) and hyper-parameters. This makes backtracking to an earlier parameter set at a point in the experiment, or the analysis of interim learners, impossible.
- There are few languages defined for the exchange of data mining and machine learning (ML) functions and parameters, which provides a barrier to sharing and exchanging the complete set of results captured during the experiment.

There have been a number attempts to address the above problems via frameworks such as CRISP-DM and SEMMA [2]. However, these frameworks are abstract and require the development of more fine-grained methodologies before any benefits can be accrued. To date, a number of ontologies have been created for example, to describe: machine learning experiments [13]; or data mining concepts in general [8]. However, ontologies are expensive to construct, often suited to specific domains and require a significant learning curve for researchers.

1.1 Contribution

In this research, we present the Parameter Optimisation for Learning (POL) data model which captures all aspects of machine learning experiments. The data model formalises the description of a deep learning experiment along with parameters and result data; this enables the design of an application programming interface (API) for the data management of both, facilitating storage and deeper analysis of each trial and learner in the experiment. In specific terms, using a JSON API for our data model provides a platform for historical analyses and comparison across these analyses; a high level of interoperability enabling our results to be shared and evaluated by others; and more efficient learning through the ability to pause experiments, resume from any checkpoint and iterate on results. Our evaluation demonstrates how to achieve a reduced *HP-search-space*, one important requirement in machine learning experiments.

Terminology. For a paper which covers both data modelling and machine learning, it is necessary to clarify the terminology we will use throughout the paper. The conceptual *model* presented in this paper captures all of the *data properties* of a machine learning experiment. For this reason we will use the term **data model** to refer to our representation of these data concepts. When discussing the machine learning aspects of our work, we will use the terms **algorithm** or **learner** to refer to the model being learned.

Paper Structure. This paper is structured as follows: in the following section, we provide an overview of related research; in §3, we describe our conceptual model which captures all aspects of deep learning experiments and analysis; in §4, we present a deployment architecture which uses our conceptual model as an interface layer to deep learning functions, parameter settings and result data which are stored using NoSQL (Mongo) technology; our evaluation is described and discussed in §5; and finally, in §6, we provide some conclusions.

2 Related Research

The first model interchange format for predictive data mining functions was PMML [7]. They aimed to provide a mechanism for working with different types of predictive models by defining a convenient language for importing and exporting these models between different systems. Their experience with DM applications had shown the usefulness that a flexible interchange mechanism would provide and they argued that previous interchange formats were proprietary. However, PMML lacks a conceptual abstract model to describe a machine learning experiment, nor does it describe or utilise experiment databases which store interim training results. Instead, its focus is purely on model deployment and interchange and therefore, does not sufficiently describe the hyper-parameter optimisation process, an important function of training deep learning models.

With the Portable Format for Analytics (PFA) [1], the authors provide an abstract description of a machine learning model allowing user-defined algorithms and models. While PFA incorporates JSON for its implementation model, its aim is somewhat different to our own. Similar to our approach, PFA provides a mechanism to export and exchange models where not previously possible. The main difference is that PFA focuses on the deployment of their model to production environments whereas we focus on the analysis of all aspects of a machine learning experiment. This enables the building of more robust learning models and aids in hyper-parameter selection. As PFA is a 'mini-language' rather than a data-model, it provides the capability to take in data and score this data according to the algorithm it has learned. As a result, it is a complex process whereas we aimed to define a light-weight, simple format that allows a formal description of a deep learning experiment and model.

In [5], the authors present the MEX vocabulary, a lightweight interchange format for ML experiments, which is an extension to the PROV-O ontology [10]. Their aim is similar to our own, but instead of taking a data-modelling approach, their methodology focuses on a linked-data, semantic web paradigm. Again, similar to the research presented here, [5] aims to provide a means to describe the elements of a learning experiment instead of exhaustively defining all aspects of the knowledge discovery process. However, a physical or implementation model for this ontology is not provided, nor an interface to persistent storage for later evaluation of experiment results. We also believe that the RDF graph-store does not provide as high a level of interoperability offered by JSON.

3 A Conceptual Model for Deep Learning Methods

The goal of our conceptual model is to capture all data properties of the ML *experiment*. There are three broad aspects to our model: model-parameter, hyper-parameter and (interim and final) result data management. The highest level of abstraction is an *experiment* and within that entity are all objects and attributes required to describe parameter and result data. Thus, we refer to our data model as the Parameter Optimisation for Learning (POL) model.

3.1 Model Overview

In Figure 1, we present a detailed illustration of the POL data model and the 3 levels of data capture required. At the highest level, the **Experiment** class is the entry point to the model and has a 1-to-many relationship with the **Learner** class, meaning an experiment can have multiple occurrences of a learner. As the search space settings remain constant for an experiment, the **HyperParam-SearchSpace** is also present at this level. The **Learner** (at one level down from **Experiment**) has a 1-to-many relationship with the **Layer** class, allowing the algorithm to have one or more layers. Within **Learner**, there are three main concepts: *parameters* (weights and biases) which are represented by the **LayerConfiguration**, **Layer** and **Tensor** classes; *hyper-parameters* which are represented by the **HyperParameters** class; and *results* (output from any iteration of the algorithm) which are represented by the **LearnerPerformance**, **Performance**, **ConfusionMatrix** and **Tensor** classes as well as **Indices** which describes the dataset configuration which generated those results. Result data is captured at this layer as the entire Learner is used to produce results. At the lowest level of the data model is the **Layer** class which contains the weights and biases for a Layer in the Learner and as these are multi-dimensional mathematical objects they are represented by the **Tensor** class.

3.2 Model Details

In this section, we provide a detailed description of two of the more important classes of POL data model: `Learner` and `HyperParameters`, as space restrictions prevent a full description of the entire model. `Learner` is described by:

- `learner_type`: Name of the learning algorithm used, for example: restricted boltzmann machine or recurrent neural network (RNN).
- `learning_type`: Learning task: reinforcement, supervised, unsupervised, etc.
- `optimisation_method`: Optimisation method for the learning function e.g. mini-batch stochastic gradient-descent (MSGD).
- `hyper_parameters`: Input and fixed parameters used by the optimisation process and initialised within the search-space bounds.
- `layers`: A list of `Layer` objects, which transform features into more abstract features or classifications and predictions. A `Layer` contains the model-parameters, weights and biases which make up a `Learner`.
- `performance`: Instance of the `Learner_Performance` object, containing a result snapshot for an update, or the final result if training has finished.
- `trained`: Boolean attribute to indicate if the `Learner's` model-parameters are optimised. Otherwise, the instance is a snapshot of a particular update.

To represent hyper-parameter optimisation, 2 classes are required in our model: **HyperParamSearchSpace** and **HyperParameters**. The search-space class defines an upper and lower bound for each HP, from which n_trial *hyper-parameters* are instantiated to find the best setting. We now describe `HyperParameters` which details a single configuration generated within the space:

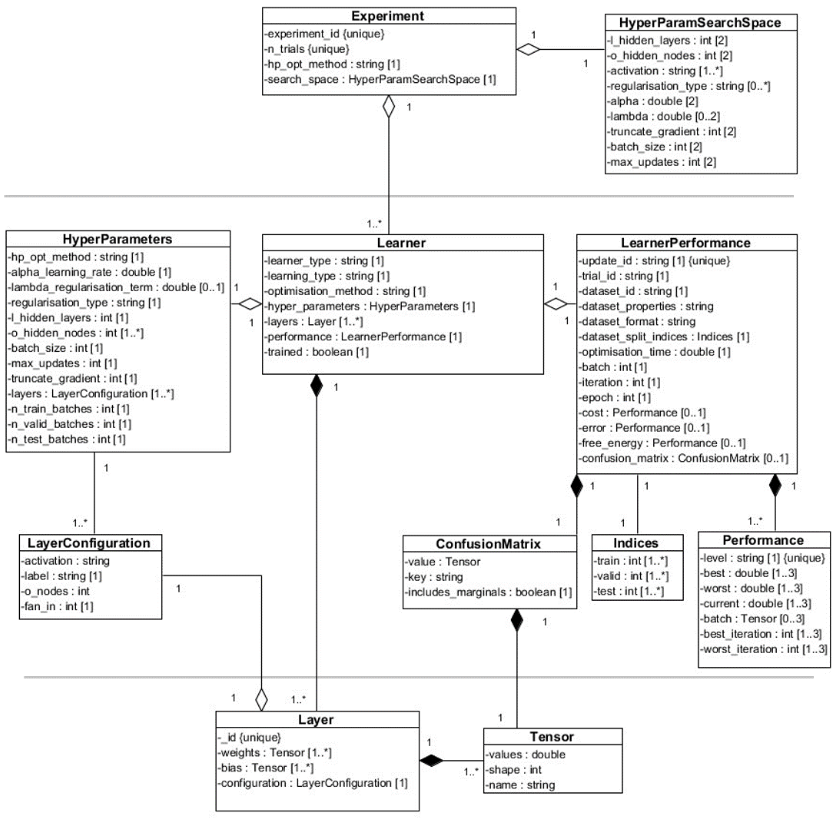
Experiment
-experiment_id {unique}
-n_trials {unique}
-hp_opt_method : string [1]
-search_space : HyperParamSearchSpace [1]
HyperParamSearchSpace
-l_hidden_layers : int [2]
-o_hidden_nodes : int [2]
-activation : string [1..*]
-regularisation_type : string [0..*]
-alpha : double [2]
-lambda : double [0..2]
-truncate_gradient : int [2]
-batch_size : int [2]
-max_updates : int [2]
HyperParameters
-hp_opt_method : string [1]
-alpha_learning_rate : double [1]
-lambda_regularisation_term : double [0..1]
-regularisation_type : string [1]
-l_hidden_layers : int [1]
-o_hidden_nodes : int [1..*]
-batch_size : int [1]
-max_updates : int [1]
-truncate_gradient : int [1]
-layers : LayerConfiguration [1..*]
-n_train_batches : int [1]
-n_valid_batches : int [1]
-n_test_batches : int [1]
Learner
-learner_type : string [1]
-learning_type : string [1]
-optimisation_method : string [1]
-hyper_parameters : HyperParameters [1]
-layers : Layer [1..*]
-performance : LearnerPerformance [1]
-trained : boolean [1]
LearnerPerformance
-update_id : string [1] {unique}
-trial_id : string [1]
-dataset_id : string [1]
-dataset_properties : string
-dataset_format : string
-dataset_split_indices : Indices [1]
-optimisation_time : double [1]
-batch : int [1]
-iteration : int [1]
-epoch : int [1]
-cost : Performance [0..1]
-error : Performance [0..1]
-free_energy : Performance [0..1]
-confusion_matrix : ConfusionMatrix [0..1]
LayerConfiguration
-activation : string
-label : string [1]
-o_nodes : int
-fan_in : int [1]
ConfusionMatrix
-value : Tensor
-key : string
-includes_marginals : boolean [1]
Indices
-train : int [1..*]
-valid : int [1..*]
-test : int [1..*]
Performance
-level : string [1] {unique}
-best : double [1..3]
-worst : double [1..3]
-current : double [1..3]
-batch : Tensor [0..3]
-best_iteration : int [1..3]
-worst_iteration : int [1..3]
Layer
-_id {unique}
-weights : Tensor [1..*]
-bias : Tensor [1..*]
-configuration : LayerConfiguration [1]
Tensor
-values : double
-shape : int
-name : string
1
1..*

Fig. 1. POL Conceptual Model

- `hp_opt_method`: The name of the algorithm used to optimise the hyper-parameters.
- `alpha_learning_rate`: Determines the magnitude of parameter updates for one step of gradient descent (GD).
- `lambda_regularisation_term`: Determines the penalty placed on very large or small weights and biases or null if dropout or no regularisation is applied.
- `regularisation_type`: Type of regularisation applied to the model-parameters, for example L1, L2, dropout, dropconnect or none.
- `l_hidden_layers`: Number of hidden layers in a `Learner`; 1 or less is considered shallow whereas anything more is considered deep.
- `o_hidden_nodes`: List where each element describes the number of hidden nodes in each layer.
- `batch_size`: Number of dataset rows to use in MSGD, which affects the algorithm's learning ability. A size of one is synonymous with stochastic GD and a size equal to the number of training samples equates to batch GD.

- `max_updates`: Maximum number of GD updates to apply to a learning function, the bounds depend on the number of model-parameters and rows in the dataset; used as an exit parameter or a patience parameter in early-stopping.
- `truncate_gradient`: Describes how far in the past to pass errors in back-propagation through time.
- `layers`: A list of `LayerConfiguration` objects which detail the setup and label of each layer in the architecture.
- `n_train_batches, n_valid_batches, n_test_batches`: Number of batches in training, validation and test sets, respectively.

4 Deployment Architecture

We now describe the system architecture where the POL is deployed. It comprises of: Data Storage, Interoperable and Application layers, shown in Figure 2.

Data Storage Layer. In order to develop an interoperable API to deliver the goals specified in our introduction, the POL data model was implemented in JSON and currently uses MongoDB for storage. This facilitates a direct mapping between the JSON API and the NoSQL database (MongoDB). The efficient storage of parameters and result data, together with the exploitation of key properties of the NoSQL databases to construct the experiment database form part of a future research submission.

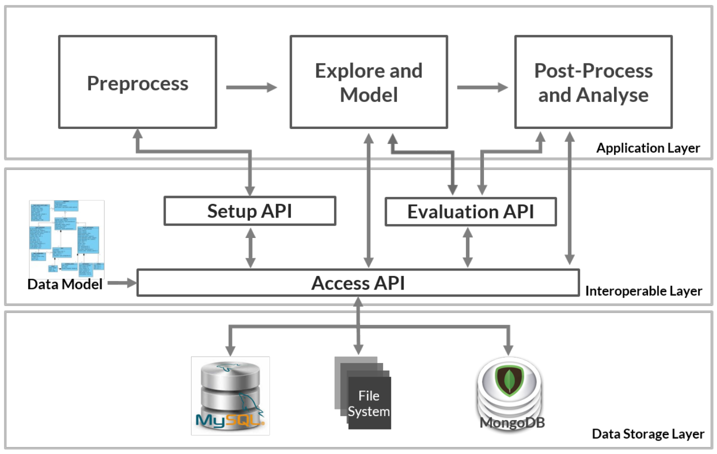

Fig. 2. Operational Architecture

Interoperable Layer. The goal of the Interoperable Layer is to facilitate greater flexibility in the learning process but also to facilitate sharing of results for comparison and analysis. The layer has 3 libraries to achieve those goals: the **Setup** library contains all functions to instantiate an experiment, read in the data and configure the database in order for results and snapshots to be processed; the **Evaluation** library contains functions to analyse and rank the performance of different trial-runs in the learning process; and the **Access** library abstracts storage details from the higher level libraries. The Access API is developed using JSON and is a direct implementation of the POL data model. This API contains all of the functionality to write and read attributes before and during a deep learning experiment. The Setup and Evaluation libraries are developed using Python and are currently accessible using Python APIs. These libraries provide higher level functionality for experiment setup and evaluation, both use the Access API to read and write to Mongo.

Application Layer. The major applications which use the toolkit represents different aspects of learning and deep learning experiments: experimental setup (Preprocess); learning (Explore/Model) and evaluation (Post-Process & Analyse). Applications can either interact directly with the Access API to design their own experiments and evaluation functions or use the Python library APIs for easier manipulation of experimental data.

5 Evaluation and Analysis

The aim of our evaluation was *not* to build the most accurate model, but to demonstrate how our interoperable toolkit can be used for the management and analysis of learning experiments. Specifically the aim is HP search space optimisation. The analysis of interim results across all trial-runs was used to fine-tune the full set of hyper-parameter bounds.

The dataset used for evaluation was generated from a series of sensors worn by athletes during Gaelic Football matches and is described elsewhere [11]. Random search was employed as our HP optimisation procedure [3] and 90 *trials* were carried out for 2 *runs* each, giving 180 *trial-runs*. Table 1 shows the *search space* for experiments. Algorithm parameters were randomly initialised according to [6], save for hidden to hidden rectified linear unit (ReLU) weights, initialised according to [9] and optimised with MSGD and Early Stopping [12], [4].

Table 1. Hyper Parameters and Bounds

Hyper-parameter	**Bounds (low, high)**	**Description**
activation	(relu, logistic)	hidden layer activation
n_hidden_nodes	(1, 10)	number of hidden layer nodes
truncate_gradient	(5, 100)	number of time-steps to BP errors
learning_rate_α	(0.0001, 0.9)	co-efficient for weight updates
max_updates	(10, 10000)	max possible updates performed
batch_size	(60, 600)	samples in mini-batch update

5.1 Search Space Reduction: Results and Commentary

We first present summary experiment statistics in Table 2. The experiment consisted of 180 *trial-runs*, during which 40,830 epochs were iterated. The average size of a learner and result *snapshot* was 0.59MB, leading to *Trial-Runs* being 10.657MB in total and *Updates* nearly amounting to 2.5GB. Unlike most machine learning experiments where only the final result is captured, interim results were recorded for every epoch. It is also possible to store results for each batch update within epochs, but this level of granularity was not used in our evaluation due to the obvious cost/benefit in terms of storage and speed.

In our *Evaluation* library, `reduce_search_space` performs an analysis which uses a set of queries to access results from multiple interim trial-runs. All Learners are evaluated, with the top-k snapshots returned through `get_top_k_ids` (*Evaluation* library), which then facilitates the retrieval of associated hyper-parameters through `get_hyper_parameters` in the *Access* library. The `coalesce _hyper_parameters` analyses the top-k hyper-parameter settings and generates summary statistics before finally, a reduced search space is generated through `reduce_search_space`.

Table 2. Experiment Statistics

Collection	Count	Size(MB)	Avg.Object Size(MB)
trial-runs	180	10.657	0.059
updates	40,089	2,377.193	0.059

Table 3 shows the result of `coalesce_hyper_parameters` for the Top 20 performing HPs in the 180 *trial-runs*. It calculates the mean, standard deviation, minimum and maximum and the 25th, 50th (median) and 75th percentiles. We have also shown sample HP frequency distribution histograms, which are output from `visualise_hp_distribution` (Evaluation library) in Figures 3, 4, 6 and 5. To best understand the beneficial effects of using our system, we now explore the outputs from `coalesce_hyper_parameters` and `visualise_hp_distribution`, key functions that allow analysis of interim results from many learners, which is only possible with persistent data management. We have omitted histograms for epochs, updates and batch size due to space restrictions.

Activations. We will first consider the activations of the Top 20 *Learners* in our 180 trial runs, shown in Figure 3. Activations are categorical strings and therefore, require different analyses to numeric HPs. The count of Learners built with ReLUs is close to those with Logistic activations. This result suggests both activations have similar performance but as ReLU outweighs logistic by a ratio of 11:9, `reduce_search_space` evaluates the ReLU to be the higher performing activation.

Hidden Nodes. The (hidden) nodes summary in Table 3, shows the average value for hidden node count is 4.360 with a median of 4. A median *below* the distribution average (right-skewed distribution), suggests the ideal parameter for

Table 3. Hyper-Parameter Summary Statistics

ID	nodes	truncate	alpha	max_updates	steps	epochs	batch_size
mean	4.360	53.980	0.404	3966.870	3520.900	157.900	289.240
std	2.765	26.251	0.284	2704.307	3569.484	208.755	178.533
min	1	6	0	132	30	2	68
25%	2	33	0.139	1394	326	8	118
50%	4	55	0.373	3729	2230	88	242
75%	6.250	74.250	0.655	6332	7332	155	451.750
max	10	100	0.889	9899	9932	722	619

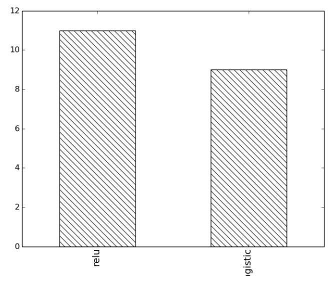

Fig. 3. Activations

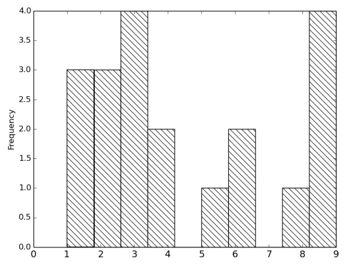

Fig. 4. Hidden Nodes

this HP would be 4 or less, confirmed in the plot of the frequency distribution in Figure 4. This means the latent features which describe the input are actually low in cardinality, which shows the dimensionality can likely be reduced.

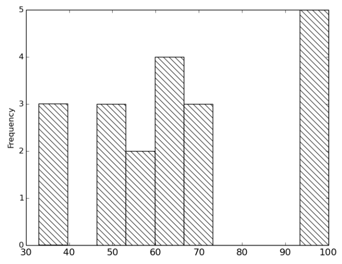

Fig. 5. Gradient Truncate

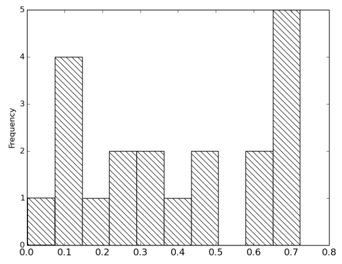

Fig. 6. Learning Rate Alpha

Truncate_Gradient. Figure 5 shows the number of time-steps recorded for optimising backpropagation through time. The value at the 75th percentile for *truncate gradient* are 74.25 in Table 3, with the mean and median at 53.98 and 55 respectively, indicating a skewed distribution. There are two possibilities for the distribution centring at these values. The first is that time-points near

$t_{i-55} : t_i$ have the greatest impact on t_{i+10}, meaning all activity for 55 seconds before time-point t_i has the greatest effect on predictions. The second possibility is that for time-points > 55 seconds, the gradient disappears, but this is unlikely as good performance was also demonstrated in the range 90 to 100.

Learning_Rate. From Table 3 and Figure 6, we can see only one configuration in the top 20 had a value below 0.1. The mean was 0.404 and the median was at 0.373, giving a right skew. Both values are quite large for a learning rate and suggest that GD is quite steep.

Table 4. Reduced Hyper-Parameter Search Space

Hyper-parameter	Bounds (low, high)
activation	(relu)
n_hidden_nodes	(1,7)
truncate_gradient	(29, 81)
learning_rate_α	(0.89, 0.657)
max_updates	(30, 9932)
batch_size	(63, 421)

The above analyses show that the median better represents the central tendency of all parameters. Also, these distributions do not lend themselves to a parametric analysis as shown in the graphs. Therefore, our selection methodology for the bounds of a reduced *search-space* consisted of taking the median and standard deviations for each hyper-parameter resulting from `coalesce_hyper_parameters` (Table 3) and generating a new bound in the range (median - std. dev., median + std. dev.), save for max_updates where we instead use the max and min values, as these parameters had a close to uniform distribution. The more realistic search bounds presented in Table 4 can only be determined using an analysis of the stored history of earlier experiments. This also facilitates augmenting random search with coordinate descent, a process not possible if we simply determine the single best configuration.

6 Conclusions

A wide range of parameters are used when optimising machine learning models. Finding the best combination of parameters in high volumes of output data, across potentially high numbers of experiments is difficult. In this paper, we addressed this issue through the development of the POL data model, which captures the entire set of parameters used in learning experiments and models. The aim of our research to facilitate the optimisation process by providing data management, analysis and optimisation functions through a standard interface, developed for the POL data model. In effect, a persistent data-store allows us to store *all* models and *all* updates, generating multiple outputs from a single experiment and a direct means of querying interim results. Our evaluation shows how using interim results, distributions can be generated for each hyper-parameter of the top 20 performing learners, which can then be analysed and used to determine an empirically reduced search-bounds in which to optimise hyper-parameters. This also allows for the extraction of confidence intervals for each hyper-parameter, which is the focus of future research.

References

1. PFA: Portable format for analytics (version 0.8.1). Tech. rep., Data Mining Group - PFA Working Group (2015)
2. Azevedo, A., Santos, M.F.: Kdd, SEMMA and CRISP-DM: a parallel overview. In: IADIS European Conference on Data Mining 2008, Amsterdam, The Netherlands, July 24-26, 2008. Proceedings. pp. 182–185 (2008)
3. Bergstra, J., Bengio, Y.: Random search for hyper-parameter optimization. J. Mach. Learn. Res. 13, 281–305 (Feb 2012)
4. Bottou, L.: Stochastic gradient learning in neural networks. Proceedings of Neuro-Nımes 91(8) (1991)
5. Esteves, D., Moussallem, D., Neto, C.B., Soru, T., Usbeck, R., Ackermann, M., Lehmann, J.: Mex vocabulary: a lightweight interchange format for machine learning experiments. In: Proceedings of the 11th International Conference on Semantic Systems. pp. 169–176. ACM (2015)
6. Glorot, X., Bengio, Y.: Understanding the difficulty of training deep feedforward neural networks. In: International conference on artificial intelligence and statistics. pp. 249–256 (2010)
7. Grossman, R., Bailey, S., Ramu, A., Malhi, B., Hallstrom, P., Pulleyn, I., Qin, X.: The management and mining of multiple predictive models using the predictive modeling markup language. Information and Software Technology 41(9), 589–595 (1999)
8. Keet, C., dAmato, C., Khan, Z., Lawrynowicz, A.: Exploring reasoning with the DMOP ontology. In: 3rd Workshop on Ontology Reasoner Evaluation (ORE14). vol. 1207, pp. 64–70 (2014)
9. Le, Q.V., Jaitly, N., Hinton, G.E.: A simple way to initialize recurrent networks of rectified linear units. arXiv preprint arXiv:1504.00941 (2015)
10. Lebo, T., Sahoo, S., McGuinness, D., Belhajjame, K., Cheney, J., et al.: Prov-o: The prov ontology. w3c recommendation, 30 april 2013. World Wide Web Consortium (2013)
11. O'Donoghue, J., Roantree, M., Cullen, B., Moyna, N., Sullivan, C.O., McCarren, A.: Anomaly and event detection for unsupervised athlete performance data. In: Proceedings of the LWA 2015 Workshops, Trier, Germany, October 7-9, 2015. pp. 205–217 (2015)
12. Prechelt, L.: Early stopping-but when? In: Neural Networks: Tricks of the trade, pp. 55–69. Springer (1998)
13. Vanschoren, J., Soldatova, L.: Exposé: An ontology for data mining experiments. In: International workshop on third generation data mining: Towards service-oriented knowledge discovery (SoKD-2010). pp. 31–46 (2010)