DILF: Deep Incremental Learning Framework over TensorFlow

Camilo Narváez, Information Technology Research Group, Universidad del Cauca, Sector Tulcán Edificio FIET, Office 422, Popayán, Colombia, camilonar@unicauca.edu.co

David Muñoz, Information Technology Research Group, Universidad del Cauca, Sector Tulcán Edificio FIET, Office 422, Popayán, Colombia, davidmu@unicauca.edu.co

Carlos Cobos, Information Technology Research Group, Universidad del Cauca, Sector Tulcán Edificio FIET, Office 422, Popayán, Colombia, ccobos@unicauca.edu.co

Abstract.

Incremental Learning seeks to improve the accuracy of models in which training data only become available over time. TensorFlow is one of several libraries used in implementing such solutions, but its scale and high learning curve make for an overly complex task. This paper presents Deep Incremental Learning Framework (DILF), an object-oriented framework (based on TensorFlow) implemented in python that facilitates the design, implementation, testing, and comparison of incremental learning algorithms using different neural network architectures, and on various datasets. DILF is divided into four highly independent and extensible modules, namely: 1) extraction, transformation, and load, 2) network architectures, 3) training, and 4) experiments.

Keywords:
Incremental Learning; Deep Learning; Neural Networks

1. Motivation and Significance

In Artificial Intelligence (AI), training models requires both quantity and quality of data to generalize most accurately the problem to be solved. All data are not always available in the initial training, but obtained in batches or increments, over time. To maintain or improve quality, measured in accuracy, recall, F-measure, etc. [1] in such cases, models must be retrained using all of the available data (old and new), a time-costly task. Deep Learning (DL) faces similar difficulties, especially when the success of a model depends largely on the high volumes of data used in training deep neural networks (DNNs) [2].

Incremental learning (IL) has therefore emerged as an option for training AI models, including DL ones, when there are incremental data flows over time and the aim is to avoid re-training models from scratch. IL seeks to develop a method in DL capable of refining the weights of a DNN based on new data, without the network forgetting what has already been learned (Flexibility and Stability Trade-Off) and with a quality similar or superior to that obtained were the DNN to be trained from scratch with all of the available data up to a certain point in time. This would mean a substantial reduction in the time required for refining models, as well as in the computational resources needed for their training [1].

High-level libraries such as Keras [3] are available for developing DL solutions. These provide a simple and clear syntax and a complete documentation, so that problems can be solved using very few lines of code. Unfortunately, this type of library is not suitable if the existing learning algorithms require to be modified in order to become

incremental or to include new algorithms with this characteristic, since the high degree of abstraction of the operation blocks used makes this task very complex.

Other libraries that allow access to lower level aspects include TensorFlow (TF) [4], Caffe [5], and Torch [6]. Implementing solutions using these libraries directly, however, is also a complex task that requires same or more time than with high-level libraries. In addition, as a result of freedom of use of the operators existing in these libraries, most proposals are developed in a single scripting file and do not comply with basic principles of software design; less so in the case of an organized, reusable and extensible architecture. IL proposals developed to date have used this approach and the available code is undocumented, not modularized, mixes data with code, does not control versions, and is difficult to understand, modify, debug, apply to other problems (datasets) or use with other DNN architectures, due to its high coupling and low cohesion (basic principles of software design).

This paper presents Deep Incremental Learning Framework (DILF), an object-oriented framework implemented in python that facilitates the design, implementation, testing, evaluation and comparison of IL algorithms in DL. DILF establishes a set of weakly coupled modules and classes with clearly defined tasks (high cohesion), which makes it easier to separate the responsibilities of each component and encourages the development of new incremental learning algorithms that are more extensible and with replicable results. A more prolific and rapid development of the research area would thus be expected. These modules allow loading data in two different formats, defining new learning algorithms (incremental or not), training and validating the results of a DNN with existing and new algorithms, and configuring experiments with different datasets (problems), network architectures and learning algorithms. Connector methods are also provided to eliminate much of the verbosity and complexity of TF, so that the focus is on the research and construction of new learning algorithms, and not on the number of operators TF has.

DILF incorporates four datasets widely used in incremental learning: MNIST, Fashion-MNIST, CIFAR-10 and Caltech 101. These were first divided into 5 megabatches (increments) and are presented sequentially to the learning algorithms. Division of the megabatches was done for two incremental scenarios: the first with megabatches that include new classes and the second with megabatches that have unbalanced classes, i.e. classes appear in multiple megabatches in different proportions. In DILF, all the learning algorithms available in TF such as RMSProp, Adam, and Gradient Descent can be used; in addition, two IL algorithms proposed by the framework authors are included.

2. Software Description

2.1. Software Architecture

In Fig. 1. a high-level view is shown of the developed Framework divided into 4 modules:

- **Extraction, Transformation, and Load (ETL) module:** focused on loading data for training and testing (light green background in Fig. 1.).
- **Network Architectures module:** allows the integration of new neural network architectures (light blue background in Fig. 1.).
- **Training module:** responsible for training a network and designed to add new IL algorithms without affecting the other modules (light red background in Fig. 1.).
- **Experiments module:** makes it possible to create and perform experiments on multiple architectures, algorithms and datasets (light yellow background in Fig. 1.).

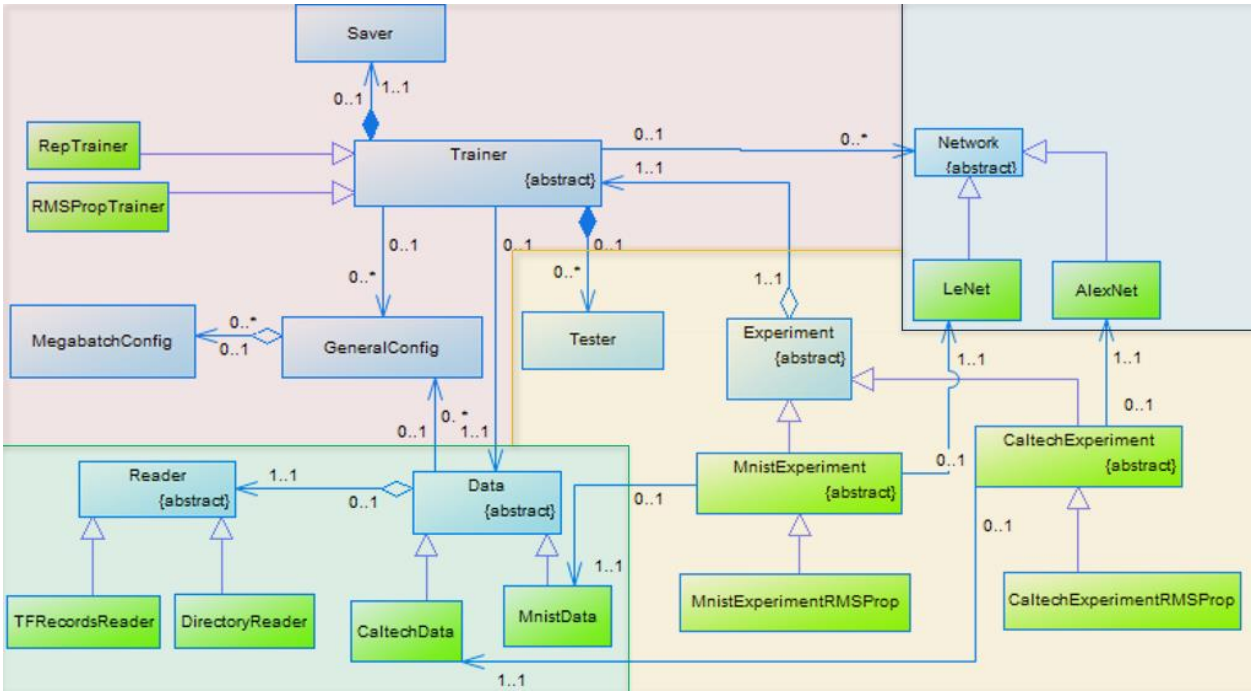

Fig. 1 Framework Architecture. The blue classes represent the core, and the green classes are a sample of subclasses associated with the implementation of domain-specific algorithms.

2.2. Software Functionalities

***Data Pipeline**:* The ETL module allows the loading of datasets by dividing them into megabatches and batches as shown in Fig. 2. Megabatches are used to facilitate execution of IL on different data sets, and batches are used to give flexibility in the feeding of data for training the DNN. There are two main classes in this module: *Reader*, which focuses on the extraction of data from disk; and *Data*, responsible for processing and loading data in the form of batches. ETL stages thus become independent. DILF provides *Reader* implementations for two commonly used formats: TensorFlow TFRecords and a directories hierarchy like the one used in Caltech 101 [7] or Tiny Imagenet [8]. In Data, implementations are provided to process several datasets widely used in DL research, such as MNIST [9], Fashion-MNIST [9], CIFAR-10 [10], and Caltech-101 [7]. Implementations are based on the TF Dataset class, which allows parallelization of operations thanks to the functions it already provides. This also makes it possible to use the ETL module independently of the rest of the framework, if it is only required to load data easily and efficiently.

***Defining a Network Architecture**:* The *Network* class makes it possible to represent a generic DNN that can have N layers of different types. This class provides several methods aimed at facilitating the development of networks with different architectures, offering useful functions that support the creation of various types of layers such as ReLU, convolutions and pooling. This class was adapted and extended from the Caffe-Tensorflow library [11] to facilitate the aggregation of multiple DNN architectures without affecting the other parts of the model. Additionally, it supports the use of Transfer Learning and allows freezing some layers during training. The framework provides some well-known network architectures, such as LeNet [12] and AlexNet [13], as well as other generic architectures.

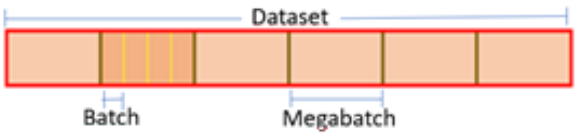

Fig. 2 Division of a dataset into megabatches and batches

***Training a Neural Network**:* The *Trainer* class is the central component of the Training module. It allows executing the training process according to the provided configuration. For this, three methods are used that work at different levels: train, to prepare the environment, start the training and train the network in a complete dataset; train_megabatch, to perform the training on a dataset megabatch, comprising multiple batches; and _train_batch, to apply the chosen optimizer selected to train the network with a data batch. Handling the training this way makes it possible to apply different configurations at each level of specificity (e.g., a global learning rate, but with a different number of epochs for each megabatch) and supports carrying out an incremental training. Additionally, *Trainer* features _create_loss and _create_optimizer functions that allow each subclass to use different loss measures (e.g. Cross Entropy) and optimizers (e.g. RMSProp).

***Define an experiment**:* the experiments module is responsible for uniting the functionality of the previous modules for carrying out an experiment. Here, the specific class of each module to be used is defined, and training and validation configured. It furthermore offers support for completely incremental (i.e. only data from the current megabatch is used) or cumulative training (all megabatches seen so far are used), with the possibility of adding more training modes if required. Moreover, it provides the *Tester* class to validate the model by calculating and storing metrics in a log file that can be read by TensorBoard. It also provides an implementation to record loss and accuracy during training. It is possible to replace or extend this class to use other metrics.

3. Illustrative Example

Set out below are the steps for using RMSProp algorithm [14] for training over the MNIST dataset [9] with LeNet [12]. Implementation of other algorithms and integration of other datasets use similar steps.

***Step 1 - Data pipeline**:* To use the input pipeline and support the new dataset, it is necessary to create a class that inherits from *Data*, implementing the following methods: 1) _build_generic_data_tensor() where the tensors corresponding to images and labels are built, and 2) close() where any file opened by the pipeline is closed. Implementation should also take account of the way in which the dataset is stored in disk, to use the appropriate *Reader*. In this case, TFRecordsReader is used as data source for the pipeline.

Implementation of the _build_generic_data_tensor method enables building the training tensors and evaluation in the *Data* super class. Before starting the training, different operations such as data transformation, data augmentation, and random ordering can be applied to the data. An example of implementation is shown in **Fig. 3**, where a data transformation is performed using the map function. Subsequently, use is made of the prepare_basic_dataset function that incorporates the application of several common transformations, such as ordering, use of caching, repetition for several epochs, and division into batches. Finally an iterator is created and the tensors corresponding to images and labels are obtained. To make use of an already created Pipeline, it is only necessary to invoke the build_train_data_tensor and build_test_data_tensor methods to obtain, respectively, the training and validation data.

```
def _build_generic_data_tensor(self, reader_data, shuffle, augmentation, testing, skip_count=0):

    filenames = reader_data[0]
    dataset = tf.data.TFRecordDataset(filenames, num_parallel_reads=len(self.general_config.train_configurations))
    dataset = dataset.map(self.parser, num_parallel_calls=8)
    dataset = self.prepare_basic_dataset(dataset, shuffle=shuffle, cache=True, repeat=testing,
                                         skip_count=skip_count, shuffle_seed=const.SEED)

    iterator = dataset.make_initializable_iterator()
    images_batch, target_batch = iterator.get_next()
    return iterator, images_batch, target_batch
```

Fig. 3 Source code of _build_generic_data_tensor

***Step 2 - Defining a Network Architecture**:* To create a new network architecture, it is necessary to create a new class that inherits from *Network* class and implements the setup() method, defining and linking each of the layers in the appropriate order. In this case, a LeNet network is created by linking the conv, pool and fc methods of the *Network* class, and assigning the input tensor via feed.

***Step 3 - Training a Neural Network**:* To implement the RMSProp algorithm it is necessary to create a class that inherits from *Trainer* class, and that implements the following methods: 1) _create_loss() where the loss measure to be used is defined, e.g. Softmax Cross Entropy from TF; 2) _create_optimizer () where the optimizer to be used is defined; and 3) _train_batch () where the samples of a batch are received and the optimizer is applied.

The created class can be seen in **Fig. 4**. The loss measure used is Softmax Cross Entropy and it must be provided with the output of the network and the ground truth labels of the samples to be calculated. Mask_tensor is also used, which stores the classes seen so far, to simulate a training in which the DNN increases its number of output nodes over time. The optimizer provided is RmsProp, and the learning rate, loss measure, and list of variables should be passed to it to be optimized. It is this component that modifies the weights of the network during training. To use this component, it is only necessary to instantiate a class by providing the configuration, input pipeline and network to be trained.

```
class RMSPropTrainer(Trainer):

    def _create_loss(self, tensor_y: tf.Tensor, net_output: tf.Tensor):
        return tf.losses.softmax_cross_entropy(tf.multiply(tensor_y, self.mask_tensor),
                                               tf.multiply(net_output, self.mask_tensor))

    def _create_optimizer(self, config: GeneralConfig, loss: tf.Tensor, var_list=None):
        return tf.train.RMSPropOptimizer(config.learn_rate).minimize(loss, var_list=var_list)

    def _train_batch(self, sess, image_batch, target_batch, tensor_x: tf.Tensor,
                     tensor_y: tf.Tensor, train_step: tf.Operation, loss: tf.Tensor,
                     increment: int, iteration: int, total_it: int):
        return sess.run([train_step, loss],
                        feed_dict={tensor_x: image_batch, tensor_y: target_batch,
                                   self.mask_tensor: self.mask_value})
```

Fig. 4 Source code of RMSPropTrainer

***Step 4 – Defining the Experiment**:* In this module the link with the other modules is made and all aspects required to execute the experiment are defined. The experiment must inherit from the *Experiment* class, implementing the following methods: 1) _prepare_data_pipeline(), where the pipeline that supplies the data is created; 2) _prepare_neural_network(), where an instance of the model to be trained is created; 3) _prepare_trainer(), where the trainer object is created; and 4) _prepare_config(), where the specific configurations for training and validation are created. A single global configuration object (*GeneralConfig*) and as many local configuration objects as there are megabatches in the dataset (*MegabatchConfig*) should be instantiated. It is important to highlight that if, for example, it is sought to run a test using a neural network other than LeNet, or using the Adagrad training algorithm

rather than RMSProp, it is only necessary to define it in a new experiment, without modifying the other components of the framework.

Step 5 - Executing the Experiment: To execute an experiment, it is first necessary to have a folder with the data that will be used for the training (i.e. the dataset). Then, the class constructor and the prepare_all and execute_experiment methods are used, providing the directory addresses, intervals at which the validations will be carried out, and training mode. While the experiment is running, the training results are saved in real time into the summaries folder and can be viewed by accessing them with TensorBoard.

Fig. 5 shows the results of multiple incremental training experiments (30 executions per experiment) using two algorithms and four datasets (MNIST, Fashion-MNIST, CIFAR-10 and Caltech 101), with different configurations. The datasets are used in two scenarios: incremental classes (i.e. each class appears in a single megabatch) and unbalanced classes (i.e. classes appear in multiple megabatches in different proportions). The training algorithms used are RMSProp and Naive Incremental Learning (NIL), an incremental proposal by the authors that uses RMSProp alongside a set of randomly selected representatives (1% or 10% of the total dataset data for these experiments). RMSProp is used incrementally (i.e. using only the data of the new megabatch) and in an accumulated way in which it is trained with all data seen so far. Using RMSProp in the accumulated scenario allows a goal or upper-limit to be set that is expected to be achieved or surpassed using incremental algorithms, and in less time. Overall, it can be seen that the best results are obtained with NIL 10%, then NIL 1% and finally incremental RMSProp, which catastrophically forgets the information learned from the megabatches previously used in training the DNN. In addition, it can be observed that the most complex scenario to be faced by the algorithms is that of incremental classes.

4. *Impact*

Current research on AI and IL is broad and vibrant. This framework seeks to assist researchers in these fields by providing a tool that abstracts at a medium level the most common functionalities of TensorFlow, one of the libraries most widely used in DL, thereby facilitating implementation of new solutions, so that researchers can focus on the core of their research, letting the framework take care of necessary but not central aspects of the research: how to load data, or result validation. A clear and flexible architecture is also provided that facilitates the creation and reading of the source code so that other researchers can easily understand it, meaning that researchers no longer require to invest time in reimplementing complex algorithms. Thus, adoption of the developed algorithms is facilitated for the purposes of comparison as well as those of extension and modification. DILF enables the comparison of multiple algorithms, using different neural network architectures, and on various datasets. This is achieved because each module has a clear responsibility and is independent of any other (low coupling), so each component is implemented only once but can be used in as many different types of experiments as is required. This is not currently the case in many code implementations released by a number of researchers, since their components are highly coupled and it is difficult to modify them to be used in experiments different from that proposed by the authors. DILF is currently being used by the authors to develop a new incremental learning algorithm and compare its results with other state-of-the-art algorithms. The framework presented here has facilitated the development of the research.

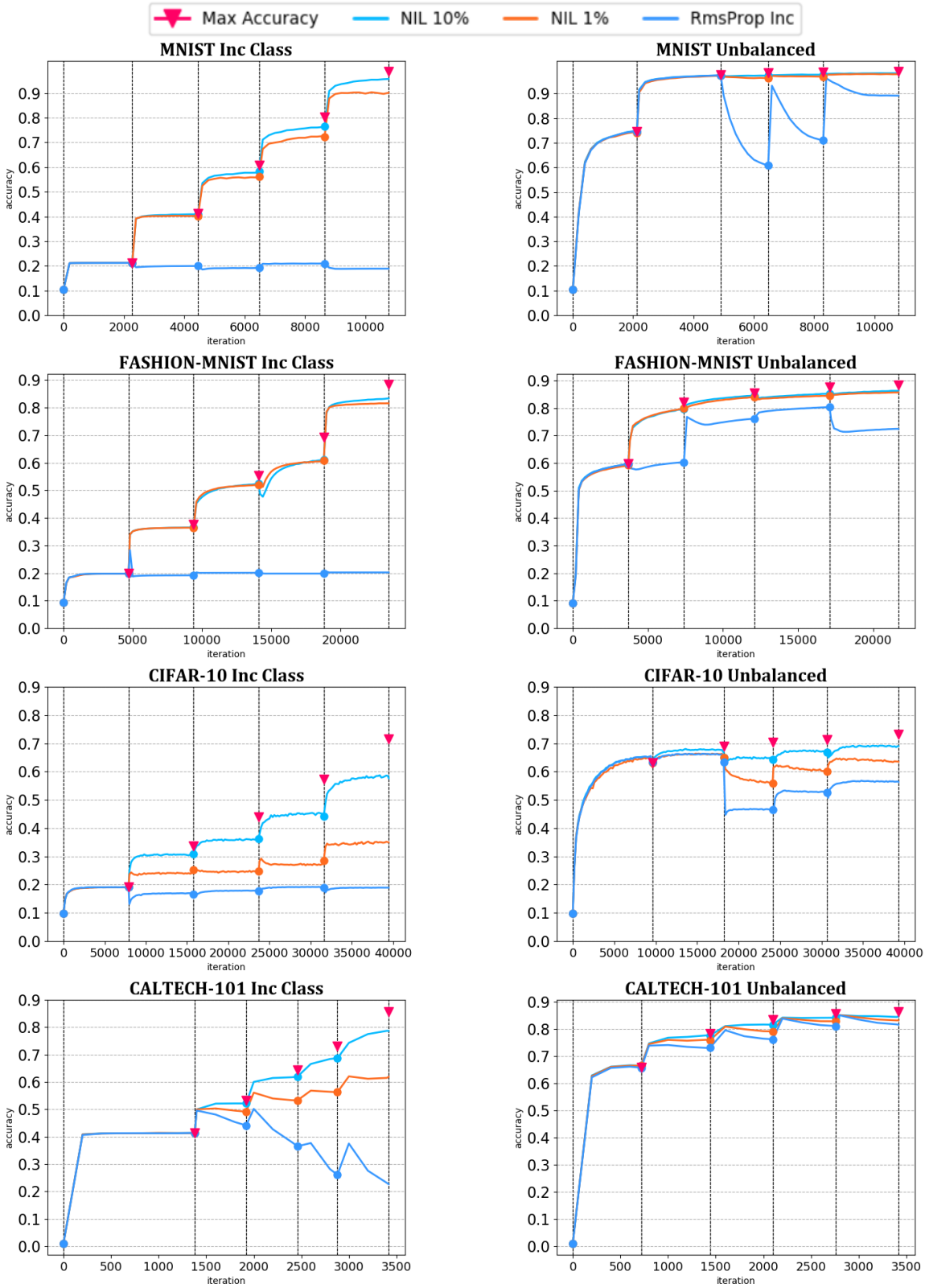

Fig. 5 Sample tests using four algorithms and four datasets. The vertical lines represent the change of megabatch. The graphs on the left show the results when each megabatch includes new classes (incremental) are used and the graphs on the right show the results when unbalanced megabatches are used.

5. Conclusions

The modular structure of the proposed framework facilitates the development, evaluation and comparison of incremental learning algorithms in Deep Learning. The ETL module allows easy addition of datasets to conduct tests and experiments without modifying the code of existing algorithms, while the Networks module allows easy addition of new neural network architectures. In the Training module - the core of the framework - multiple common training tasks are abstracted, facilitating the extension and design of new training algorithms and the Experimentation module allows the conditions of an experiment to be clearly specified so that other researchers can replicate the results obtained. Taking into account the low adoption of modularized and decoupled designs in implementations of incremental algorithm, using DILF is expected to encourage these good practices and ensure that the implemented algorithms will be used by other researchers, increasing their visibility. DILF is available as open source software, and we hope it will be used and expanded, adding support for more ample range of datasets and training algorithms in the future.

References

[1] A. Gepperth and B. Hammer, "Incremental learning algorithms and applications," in *European Symposium on Artificial Neural Networks (ESANN)*, 2016.

[2] J. Schmidhuber, "Deep Learning in Neural Networks: An Overview," *Neural Networks*, vol. 61, pp. 85–117, 2015.

[3] F. Chollet and others, "Keras." 2015.

[4] M. Abadi *et al.*, "TensorFlow : A System for Large-Scale Machine Learning This paper is included in the Proceedings of the TensorFlow : A system for large-scale machine learning," *Proc 12th USENIX Conf. Oper. Syst. Des. Implement.*, pp. 272–283, 2016.

[5] Y. Jia *et al.*, "Caffe: Convolutional Architecture for Fast Feature Embedding," *arXiv Prepr. arXiv1408.5093*, 2014.

[6] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A Matlab-like Environment for Machine Learning," in *BigLearn, NIPS Workshop*, 2011.

[7] R. F. and P. P. L. Fei-Fei, "Caltech 101."

[8] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," *Ttechnical Rreport, Dep. Comput. Sci. Univ. Toronto*, pp. 1–60, 2009.

[9] Y. LeCun and C. Cortes, "{MNIST} handwritten digit database," 2010.

[10] "The cifar-10 dataset." [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html.

[11] Ethereon, "Caffe-Tensorflow." GitHub, 2016.

[12] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.

[14] G. E. Hinton, N. Srivastava, and K. Swersky, "Lecture 6.5- Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural Networks for Machine Learning*. p. 31, 2012.

Required Metadata

Current code version

Table 1 – Code metadata (mandatory)

Nr	Code metadata description	
C1	Current code version	*v1.0.0*
C2	Permanent link to code/repository used of this code version	https://github.com/camilonar/DILF/
C3	Legal Code License	GNU General Public License (GPL) 3.0
C4	Code versioning system used	Git
C5	Software code languages, tools, and services used	Python 3.6 TensorFlow TensorBoard
C6	Compilation requirements, operating environments & dependencies	S.O. independent, tested on Windows 10 TensorFlow >= 1.9.0 Numpy >= 1.14.5
C7	If available Link to developer documentation/manual	https://camilonar.github.io/DILF/
C8	Support email for questions	camilonar@unicauca.edu.co