Making Classical Machine Learning Pipelines Differentiable: A Neural Translation Approach

Gyeong-In Yu
Seoul National University
gyeongin@snu.ac.kr

Saeed Amizadeh
Microsoft
saamizad@microsoft.com

Byung-Gon Chun
Seoul National University
bgchun@snu.ac.kr

Markus Weimer
Microsoft
mweimer@microsoft.com

Matteo Interlandi
Microsoft
mainterl@microsoft.com

Abstract

Tools such as Scikit-learn allow data scientists to create pipelines composed of data featurizers and machine learning models where models, within a pipeline, are trained in isolation. Conversely, if we look at frameworks such as TensorFlow, layers composing neural models are simultaneously trained using backpropagation. In this paper, we propose a neural translation framework where operators in a machine learning pipeline are mapped into a neural network representation and globally fine-tuned using backpropagation. Our experiments show that fine-tuning pipelines is a promising technique able to increase the final accuracy.

1 Introduction

Recently, deep neural networks (DNNs) have been exceptionally successful in pushing the limits of various fields such as computer vision and natural language processing [21, 12]. Nevertheless, classical machine learning (ML) techniques such as gradient boosting trees and linear models are still very popular among practitioners [4], especially because of their intrinsic efficacy and interpretability. This dichotomy has produced two different sets of system offerings. On one side, frameworks such as TensorFlow [9], PyTorch [18], and Caffe2 [1] are mainly specialized in DNNs. On the other side, tools such as Scikit-learn [19], ML.NET [6] , H2O [2], Spark MLlib [17], or Uber's Michelangelo [5] support the composition of multiple data transforms and (classical) ML models into *machine learning pipelines*, allowing users to capture end-to-end data transformation procedures as Directed Acyclic Graphs (DAGs) of operators. Operators in ML pipelines comprise not only ML models but also data preprocessing and featurization steps, and are often trained *greedily* in topological order.

Internally at Microsoft, we have empirically observed that the best (accuracy-wise) ML pipelines often include one or more *trainable operator*: i.e., machine learning models or data transforms (such as normalizers) requiring the estimation of parameters from the input dataset. In this paper, we argue that the sequential training of ML pipelines' operators is sub-optimal since the parameters are computed in isolation for each operator and are not globally optimized. This approach substantially differs from how DNNs are trained. DNN layers, which can also be seen as multiple cascaded operators, are typically trained *simultaneously* using *backpropagation* by which parameters can be globally estimated end-to-end to reach better (local) minima. Arguably, this is one of the most fundamental features of deep learning, together with the possibility to accelerate computation using specific hardware such as GPUs.

Inspired by these observations, we propose a framework able to *translate* (possibly trained) ML pipelines into neural networks. By doing so, we pursue two distinct goals: (a) enabling backpropagation over ML pipelines in order to bypass the greedy one-operator-at-a-time training model and eventually boosting the accuracy of the entire ML pipeline; and (b) enabling the GPU-acceleration over traditional ML pipelines without reinventing the wheel (i.e., support for hardware acceleration) for classical ML frameworks.

To validate our proposed methodology, we have implemented a framework on top of ML.NET and PyTorch [18]. The framework translates an (possibly trained) ML.NET pipeline into a PyTorch model by generating PyTorch modules out of pipeline operators and composing the modules following the dependencies represented in the original pipeline. After translation, we train the PyTorch model further to fine-tune the parameters globally using backpropagation to unlock the possibility to reach better (local) minima. Additionally, our framework is able to translate the PyTorch model back to its original ML.NET pipeline form by extracting the fine-tuned parameters and mapping them into the original pipeline space. Preliminary results are encouraging: on a regression task, our approach is able to improve the accuracy by 19.4%.

2 Background and Running Example

In this section, we present an example of an ML pipeline with multiple trainable operators and show the simple API of our framework that users can exploit to translate and fine-tune their pipelines. But first, we briefly introduce ML.NET and PyTorch to ease the understanding of the following sections.

ML.NET. ML.NET is Microsoft's machine learning toolkit implemented on .NET, providing both .NET and Python[1] interfaces. ML.NET's runtime provides streaming access to data so that high dimensional data larger than the main memory can be gracefully and efficiently handled. Computation in ML.NET is generally immutable and lazily evaluated (unless forced to be materialized, e.g., during training when multiple passes over the data are requested). In its current implementation, ML.NET has more than 100 (trainable and non-trainable) operators. ML.NET provides an *entry point* API: the recommended way of interacting with ML.NET through other, non-.NET, programming languages. An entry point is a JSON representation of an ML.NET operator. Using the entry point API, an ML.NET pipeline authored in Python can be represented as a DAG of entry point operators with their input/output relationships, which are all serialized into a JSON file. Entry point graphs are parsed in ML.NET by the GraphRunner component that directly executes the parsed pipeline in its .NET runtime.

ML.NET's Python API mirrors exactly Scikit-learn interface (ML.NET Python operators are actually subclasses of Scikit-learn components) and takes advantage of the entry point API. To achieve high-performance interoperability between Python and .NET, however, data residing in Python (for example in Pandas [16] dataframes or NumPy [23] arrays) needs to be accessed from ML.NET. To obtain such behavior in an efficient and scalable way, when ML.NET is imported into a Python project, a .NET Core runtime as well as an instance of ML.NET are spawn within the same Python process. When a user triggers the execution of a pipeline, the call is intercepted on the Python side and an entry point graph is generated for the ML.NET operators and submitted to the GraphRunner component of the ML.NET instance. If the data resides in memory, the C++ reference of the data is passed to the GraphRunner through C#/C++ interop and then used as input for the pipeline.

PyTorch. PyTorch is a deep learning framework that provides a Python interface and NumPy-like programming experience. Users can write their feedforward computation by composing PyTorch operations, while the backpropagation computation is automatically derived from the feedforward counterpart thanks to the automatic differentiation feature [18]. PyTorch also provides a *module* abstraction to represent reusable network building blocks. Developers can use both PyTorch's built-in modules or custom written modules to compose a PyTorch model. PyTorch operations can be executed either on CPUs or GPUs: devices for execution can be easily selected using a simple API.

Running Example. Figure 1 depicts a Python program that declares and trains an ML.NET pipeline implementing a regression task. Within the pipeline, we have several trainable operators: PCA (line 9), scaler (line 10), and three linear models (SDCA at line 11, logistic regression at line 13, and Poisson regression at line 16).

Initially, we divide input features into two classes: numerical features and categorical features where each feature is a value from a discrete set where the order of the elements is not defined. Such categorical features are then converted into one-hot vectors via `OneHotVectorizer` (line 6). After

[1] Python bindings for ML.NET are open-sourced as NimbusML [7].

```
1  # X is accessible via categorical_columns and numerical_columns
2  X, y = prepare_data()
3
4  # set up pipeline
5  mlnet_pipe = Pipeline([
6      OneHotVectorizer(categorical_columns),
7      Expression('x: x != 0 ? 1.0 : 0.0', {'Label01': 'Label'}),
8      ColumnConcatenator({'Features': categorical_columns + numerical_columns}),
9      PcaTransformer('Features'),
10     MinMaxScaler('Features'),
11     Featurizer(SDCABinaryClassifier('Label01','Features')),
12     ColumnConcatenator({'SDCAScore': ['Score', 'Prob']}),
13     Featurizer(LogisticRegression('Label01','Features')),
14     ColumnConcatenator({'LRScore': ['Score', 'Prob']}),
15     ColumnConcatenator({'Features': ['Features', 'SDCAScore', 'LRScore']}),
16     PoissonRegression('Label', 'Features')])
17
18 mlnet_pipe.fit(X, y) # training
```

Figure 1: An example code for an ML pipeline in ML.NET.

that, we create a binary column `Label01` based on the original label column (line 7). Next, all the feature columns are concatenated into a single vector column, `Features` (line 8), and we feed the column into a `PcaTransformer` to train the PCA model and compute the projection of the original feature vector according to the learned PCA model (line 9). We normalize the results of the projection based on the observed minimum and maximum values of the projection (line 10), and train two `Featurizers` using the normalized features in `Features` column and binary label in `Label01` column (line 11 and 13). Finally, outputs of the two `Featurizers` are concatenated with the original features (line 15) and fed into the final Poisson model (line 16). Now we can train the ML.NET pipeline (line 18) and evaluate the accuracy of the pipeline (not shown in the program) using the infrastructure provided by ML.NET.

```
1 # Continuing from Figure 1
2 pytorch_model = mlnet_pipe.to_pytorch() # translation
3
4 optim = lambda params: torch.optim.Adam(params)
5 pytorch_model.fit(X, y, optim) # fine-tuning
6
7 mlnet_pipe.from_pytorch(pytorch_model) # translate back
```

Figure 2: An example code for translating an ML.NET pipeline into PyTorch and fine-tuning the result.

As shown in Figure 2, our framework converts this ML.NET pipeline into a neural network using a simple function call (line 2). We will introduce the detailed translation mechanism in Section 3. Note that we can skip the training of the original pipeline if we want to randomly initialize the parameters of the neural network (although in our preliminary experiments this results in the worst final accuracy). The translated PyTorch model is trained using the usual `fit` method (line 5), with an additional argument `optim` that receives a list of parameters for optimization and returns a PyTorch `optimizer`. Finally, one can also translate the PyTorch model back into the ML.NET pipeline representation using a similar API (line 7).

3 Translating ML Pipelines into Neural Networks

The process for generating neural network programs out of ML pipelines is achieved by:

(a) Determining translation target operators in the ML pipeline;

(b) Generating a neural network module for each target operator;

(c) Composing all the modules into a single neural network by following the dependencies specified in the original pipeline; and finally,

(d) Wiring the neural network with the inputs of the pipeline.

The final output of the translation process is a neural network that provides the same prediction results as the original pipeline.

We further detail this process using the example pipeline of Figure 1. Initially, our framework is able to detect trainable operators with tunable parameters and label them as translation targets, by parsing the pipeline structure and taking advantage of an external mapping table (further described in Section 4). In addition to these operators, we classify the non-trainable operators that follow target operators as translation target as well. This is required in order to avoid problems with automatic differentiation, which is also described in more detail in Section 4. From our running example, the framework labels operators `PcaTransformer`, `MinMaxScaler`, `SDCABinaryClassifier`, `LogisticRegression`, `PoissonRegression`, and the three `ColumnConcatenator` at lines 12, 14, and 15 as targets for translation.

We generate a neural network module out of a target operation by mirroring the transformation procedure of the operation. For example, the `PcaTransformer` learns the column-wise mean of dataset X and the eigenvectors of the empirical covariance matrix of X. After training, the `PcaTransformer` transforms an input vector using a matrix-vector multiplication and addition, which can be represented as the feedforward computation of a fully connected linear layer in neural networks. Hence, in the second step, our system translates the `PcaTransformer` into a fully connected layer, of which the weight matrix and the bias vector are initialized using the eigenvectors and column-wise mean obtained from the training process, respectively. Translation of other linear models such as `LogisticRegression` is similar, except for the fact that they produce two output columns: `Score` and `Prob`. We can compute the `Score` column by applying a fully connected layer with an output size of 1 to the `Features` column, and the `Prob` column can be directly derived from the `Score` column by applying the Sigmoid activation. The `MinMaxScaler` and `ColumnConcatenator` are trivially translated into an element-wise multiplication and tensor concatenation, respectively.

In the third step, the translated neural network modules form a DAG, where vertices represent operations and edges represent data dependencies. We connect the modules according to the input-output relationships between the operators in the original pipeline. For instance, since the output column (`Features`) of the `MinMaxScaler` (line 10) flows into two `Featurizers` (lines 11 and 13), we insert edges between the modules corresponding to the `MinMaxScaler` and `Featurizers`. Details on the final step connecting the original ML pipeline inputs to the neural network are postponed to the next section together with our system's implementation aspects.

Once we get the translated neural network, we employ the backpropagation algorithm to compute the gradients of the final loss with respect to all its parameters. These gradients are then used for updating parameters via stochastic gradient descent or its variants, such as Adam [14]. This algorithm can be different from the original ones used for training each operator in the original ML pipeline, such as L-BFGS or SDCA [15, 22]. The important benefit of using the backpropagation is that we can incorporate training signals from the final loss to fine-tune the parameters of the intermediate operators, such as `PcaTransformer` or `LogisticRegression` in Figure 1. Compared to the original sequential training procedure of these operators that does not directly train toward the final loss, our fine-tuning stage utilizes additional information from the final loss to improve the accuracy.

4 System Implementation

Based on the translation mechanism described in Section 3, we have implemented a prototype framework that translates a ML.NET pipeline into a PyTorch model. Figure 3 illustrates the execution flow of the end-to-end training procedure. The translator in our framework constructs a PyTorch model based on the ML.NET pipeline by composing PyTorch modules that correspond to handwritten templates, one for each supported operator in ML.NET. We maintain a pre-defined mapping table between ML.NET operators and the PyTorch modules. Currently, we support a subset of ML.NET operators (most of the linear models, KMeans, and normalizers), and adding support for more operators such as decision trees is part of our future work.

As described in Section 2, the Python interface of ML.NET is a wrapper library that invokes ML.NET's GraphRunner, such that all the relevant data including trained parameters are directly

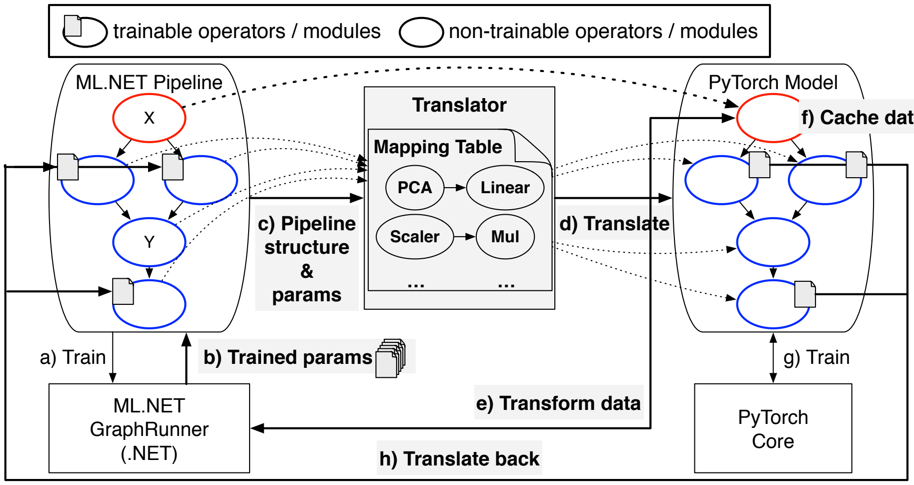

Figure 3: Execution model for training and fine-tuning of an ML pipeline using our framework. Circles in blue line represent pipeline operators translated into PyTorch and their corresponding PyTorch modules. Circles in red represent operators that are not translated but referenced by PyTorch model. Shaded labels highlight steps executed by our framework.

managed by the .NET runtime. Hence, after training the pipeline (Figure 3 a)), we move all the trained parameters from the .NET side to the Python side (Figure 3 b)) and invoke the translator (Figure 3 c)). By doing so, the translator (which is implemented in Python) can see all the already available information such as the entry point graph representing the pipeline structure as well as the trained parameters required for generating the corresponding neural network. The translator generates PyTorch modules for each pipeline operator, and initializes the modules using the pretrained parameters (Figure 3 d)).

Non-trainable ML.NET operators have no parameters to estimate, and therefore they get no accuracy improvement from backpropagation. Moreover, since ML.NET provides optimized operator implementations, it is often more efficient to simply reuse ML.NET's operators instead of substituting them with PyTorch modules at training time. However, when a non-trainable operator follows other trainable operators, we translate the operator into a PyTorch module along with the preceding trainable operators. In fact, if we do not translate the intermediate non-trainable operators, and interleave a chain of PyTorch modules with non-PyTorch components, the automatic differentiation feature [18] cannot propagate gradients across the non-PyTorch components. In Figure 3, the first non-trainable operator (X) in the ML.NET pipeline is not translated into a PyTorch module and instead a reference to it is added in the PyTorch model. Conversely, the second non-trainable operator (Y) that follows trainable operators is translated into a PyTorch module.

Once the translation is completed, we fine-tune the PyTorch model that represents the end-to-end pipeline. The framework first invokes ML.NET GraphRunner to execute the operators not translated into PyTorch modules (Figure 3 e)). A noticeable characteristic of these operators is that their outputs do not change over training epochs. Thus, we can cache the output of these operators (Figure 3 f)) and reuse it over multiple training epochs, instead of re-executing the operators at every iteration. This also can be seen as a separate data preprocessing step, which is typically done before actual training. Since ML pipelines often contain non-trainable operators at the beginning in order to preprocess the training data, our framework's runtime performance benefits from the caching strategy. After fine-tuning the PyTorch model using the cached data (Figure 3 g)), we can translate the model back to the ML.NET pipeline by simply updating the parameters of the pipeline (Figure 3 h)).

5 Preliminary Experiments

We have evaluated our methodology by running a preliminary experiment using the pipeline introduced in Section 2. The input dataset comprises 2.3K examples for training and 0.2K examples for testing. We trained an extended version of the pipeline of Figure 1; we used our framework to translate it into a neural network and fine-tuned the neural network using backpropagation in

	Original ML.NET pipeline	Randomly initialized PyTorch model	Fine-tuned PyTorch model after translation from ML.NET
Poisson Loss	13.67	15.60	**12.01**
L1 Error	17.49	17.64	**14.79**
L2 Error	758.1	760.4	**611.2**

Table 1: Comparison of three different approaches over the test partition of an internal dataset.

PyTorch. We compare the result with two baselines: (a) the original ML pipeline without translation and fine-tuning; and (b) a randomly initialized neural network with the same network structure as the translated pipeline.

As Table 1 shows, our approach outperforms the first baseline, the pipeline trained solely on ML.NET, by up to 19.4% on all evaluation metrics. This means that the fine-tuning stage with backpropagation actually boosts the accuracy of the pipeline. The second baseline, the randomly initialized PyTorch model, converges to a sub-optimal local minimum and shows lower accuracy than the fine-tuned model. This implies that the training of the original ML pipeline provides useful initial parameters for neural networks and helps improving the final accuracy of the model.

6 Related Works

Scikit-learn Compatible Libraries. Skorch [8] is a Scikit-learn compatible neural network library. It provides PyTorch wrapper that has a Scikit-learn interface and can inter-operate with other Scikit-learn transforms and estimators. In contrast, our framework makes users write their ML pipeline using high-level building blocks rather than composing primitive PyTorch operations. Additionally, Skorch cannot apply backpropagation across multiple components hence it optimizes each component separately. HyperLearn [3] provides a Scikit-learn interface with optimized algorithms for linear/ridge regression, PCA, SVD, etc. It provides GPU support by implementing the algorithms using PyTorch's Tensor primitives. Although HyperLearn internally makes use of PyTorch, it focuses on optimizing the algorithms of the components and do not support backpropagation.

Initializing Neural Networks using Decision Trees. There have been early attempts to initialize Multi-Layer Perceptrons (MLPs) from decision trees [10, 13, 20] to make training of MLPs easier. They suggest various ways to transfer information from pretrained decision trees to MLPs and empirically show that the initialization approach provides good initial parameters to start training. Our framework can also incorporate these techniques for translating pipeline operators with tree algorithms such as gradient boosting trees or random forest into neural networks.

Translating Neural Networks into Interpretable Format. Trepan [11] is an algorithm for extracting decision trees from arbitrary trained neural networks. The algorithm does not depend on the network structure; rather, it queries a trained network to determine the split of the decision tree. On the other hand, our framework can only translate back a neural network that was previously translated from an ML pipeline. Nevertheless, unlike Trepan, our framework can preserve the accuracy of the neural network because it knows the ML pipeline structure compatible with the neural network.

7 Conclusion and Future Works

Inspired by the difference in the training of ML pipelines and neural networks, we propose a methodology for fine-tuning ML pipelines by translating them into neural networks. Translation of an ML pipeline is done by generating neural network modules based on the transformation logic of each target operator and composing them into a DAG of neural network modules. We have implemented a prototype framework on top of ML.NET and PyTorch, and our experiments show that the fine-tuning of an ML pipeline using backpropagation improves accuracy.

We are currently implementing various algorithms for translating decision trees into neural networks. Furthermore, we are investigating how to properly do hyperparameter tuning, since our approach doubles the effort for choosing good hyperparameters by introducing an additional training stage. Finally, we are evaluating how hardware accelerators (e.g., GPUs) improve runtime performance of translated ML pipelines. We deem this work as a first step towards a better comprehension of the system trade-offs existing between ML pipelines and DNNs systems.

Acknowledgments

We would like to thank the ML.NET team for suggestions on early drafts of the paper. Gyeong-In Yu and Byung-Gon Chun were partly supported by the Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No.2015-0-00221, Development of a Unified High-Performance Stack for Diverse Big Data Analytics), and by the ICT R&D program of MSIT/IITP (No.2017-0-01772, Development of QA systems for Video Story Understanding to pass the Video Turing Test).

References

[1] Caffe2. `http://caffe2.ai/`.

[2] H2O: Scalable machine learning. `https://github.com/h2oai/h2o-3`.

[3] Hyperlearn. `https://github.com/danielhanchen/hyperlearn`.

[4] Kaggle. `https://www.kaggle.com`.

[5] Michelangelo. `http://eng.uber.com/michelangelo`.

[6] ML.NET. `https://github.com/dotnet/machinelearning`.

[7] NimbusML. `https://github.com/Microsoft/NimbusML`.

[8] skorch. `https://github.com/dnouri/skorch`.

[9] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[10] A. Banerjee. Initializing neural networks using decision trees. In *Proceedings of the International Workshop on Computational Learning and Natural Learning Systems*, pages 3–15. MIT Press, 1994.

[11] M. W. Craven and J. W. Shavlik. Extracting tree-structured representations of trained networks. In *Proceedings of the 8th International Conference on Neural Information Processing Systems*, NIPS'95, pages 24–30, Cambridge, MA, USA, 1995. MIT Press.

[12] H. Hassan, A. Aue, C. Chen, V. Chowdhary, J. Clark, C. Federmann, X. Huang, M. Junczys-Dowmunt, W. Lewis, M. Li, S. Liu, T. Liu, R. Luo, A. Menezes, T. Qin, F. Seide, X. Tan, F. Tian, L. Wu, S. Wu, Y. Xia, D. Zhang, Z. Zhang, and M. Zhou. Achieving human parity on automatic chinese to english news translation. *CoRR*, abs/1803.05567, 2018.

[13] I. Ivanova and M. Kubat. Initialization of neural networks by means of decision trees. *Knowledge-Based Systems*, 8:333–344, 1995.

[14] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[15] Q. V. Le, J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, and A. Y. Ng. On optimization methods for deep learning. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, pages 265–272. Omnipress, 2011.

[16] W. Mckinney. pandas: a foundational python library for data analysis and statistics. 01 2011.

[17] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. Mllib: Machine learning in apache spark. *J. Mach. Learn. Res.*, 17(1):1235–1241, Jan. 2016.

[18] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. 2017.

[19] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, Nov. 2011.

[20] N. Rountree. *Initialising Neural Networks with Prior Knowledge*. PhD thesis, Ph. D. Thesis, Doctor of Philosophy, University of Otago, New Zealand. Accessed as pdf at: http://www.cs.otago.ac.nz/research/publications/OUCS-thesis2-2007.pdf, 2006.

[21] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, Dec 2015.

[22] M. Takáč, P. Richtárik, and N. Srebro. Distributed mini-batch SDCA. *arXiv preprint arXiv:1507.08322*, 2015.

[23] S. van der Walt, S. C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, March 2011.