DEFIne: A Fluent Interface DSL for Deep Learning Applications

Nina Dethlefs
The Digital Centre
School of Engineering and Computer Science
University of Hull, UK
n.dethlefs@hull.ac.uk

Ken Hawick
The Digital Centre
School of Engineering and Computer Science
University of Hull, UK
k.a.hawick@hull.ac.uk

ABSTRACT

Recent years have seen a surge of interest in deep learning models that outperform other machine learning algorithms on benchmarks across many disciplines. Most existing deep learning libraries facilitate the development of neural nets by providing a mathematical framework that helps users implement their models more efficiently. This still represents a substantial investment of time and effort, however, when the intention is to compare a range of competing models quickly for a specific task. We present DEFIne, a fluent interface DSL for the specification, optimisation and evaluation of deep learning models. The fluent interface is implemented through method chaining. DEFIne is embedded in Python and is build on top of its most popular deep learning libraries, Keras and Theano. It extends these with common operations for data pre-processing and representation as well as visualisation of datasets and results. We test our framework on three benchmark tasks from different domains: heart disease diagnosis, hand-written digit recognition and weather forecast generation. Results in terms of accuracy, runtime and lines of code show that our DSL achieves equivalent accuracy and runtime to state-of-the-art models, while requiring only about 10 lines of code per application.

CCS Concepts

•**Software engineering** → *Reusable software;* •**Artificial Intelligence** → *Learning;*

Keywords

Domain-specific languages, deep learning

1. INTRODUCTION

Deep learning has received a lot of interest in recent years in both academic and industrial contexts. It is increasingly used in applications such as stock market prediction [17] or medical imaging [23], natural language processing [42], computer vision [18], and artificial intelligence in general [5], in some tasks even exceeding human performance [53].

Despite the great success of deep learning and an increasing number of libraries and frameworks that are available, a substantial amount of parameter tuning is required to develop an effective model for a new domain, see e.g. the random search solution in [8]. There are few guidelines on how many layers a deep learning model should use, or what activation, loss function and optimiser will lead to an adequate representation of the input data, as these details depend crucially on the target data [6]. Developers will typically start from a basic set of rules of thumb and then experiment to find the best setup.

In this paper, we present a fluent interface DSL that aims to facilitate the process of developing deep learning models for new domains. A fluent interface [19] is one where syntactic features of the hosting language are used to good effect to construct an DSL that is embedded in a host language and that captures the jargon, the commands and other notions of the requisite application domain [20, 31]. Our fluent interface is implemented through method chaining as explained in Section 4. The general idea is to abstract away from implementational details and integrate standard operations in data representation or model definition into a DSL that is embedded in Python and built directly on top of the most popular deep learning libraries, Keras [10] and Theano [58]. We present experiments in three different domains that are relevant to real-world applications: heart disease diagnosis, hand-written digit recognition and weather forecast generation. Our results in terms of accuracy, runtime and lines of code suggest that our DSL is able to provide sufficient flexibility to programmers to express the representational and mathematical peculiarities of individual domains, while at the same time enhancing the readability and maintainability of code. The research contributions of this paper are:

1. A DSL framework that performs automatic data analysis, pre-processing and choice of hyper-parameters for deep learning applications; see Section 4.
2. An evaluation in three different application domains to demonstrate the flexibility of the DSL across datasets; see Section 5 for the datasets and Section 6 for results.
3. A reduction in code size by up to a factor of 5 at equivalent performance to recent state-of-the-art results; see Section 6.

RWDSL '17, February 04 2017, Austin, TX, USA

ISBN 978-1-4503-4845-4/17/02...$15.00

DOI: http://dx.doi.org/10.1145/3039895.3039898

2. BACKGROUND

Computer programming language designers have a major goal of helping programmers express ideas and algorithms concisely and clearly. Even the most elegant programming languages however sometimes falter in this goal, when programs become bigger and more complex. The standard computer science "divide and conquer" approach is to abstractify ideas and component parts of a large program into a framework or software library that can be separately developed, tested and hidden away, and invoked only when needed. This can considerably lower the amount of source code and hence concepts that the programmer needs to hold on their screen or in their mind all at once and is a key to managing large-scale complex software development [62].

Many different techniques and tools have been introduced in modern programming languages to help this abstraction and lowering of code complexity including: subroutines and functions; modules and packages; and classes and objects where data structures and operational code are combined together to form abstract data types. While these help, application developers are still often easily overwhelmed by the size and sheer complexity of programs [36]. These distractions take away effort from addressing the application problems - particularly when carrying out work in computational experiments when each experiment must be carefully and repeatably programmed.

Different programming languages [52] have varied relative strengths, advantages and specialities appropriate to various sorts of applications. General-purpose programming languages often have a lot of legacy operational and idiomatic features that can obscure the essence of an application and sometimes make it unnecessarily complex for developers.

A relatively new technique is to create a domain-specific programming language (DSL) [19, 63, 21, 14] that provides the programmer with a very high-level vehicle to formulate application ideas. The goal is for the language to focus concisely on only those concepts and aspects that are directly relevant to the particular application problem domain and to hide away any "boiler plate" source code that is just an artefact of the underpinning programming language. The application domain is often a business problem using business jargon and language or for the discussion below it could use the scientific terminology particular to complex deep learning systems.

A DSL can be implemented as either a full-blown language using all the necessary compiler [2] and language builder environment apparatus to aid the programmer [35]. However, this approach takes a lot of development effort and requires the implementation of a lot of other apparatus such as normal arithmetic, logic, text and string handling features to make a seamless programming environment. This approach is known as implementing an "external DSL" since the DSL is external to the programming language that the DSL system itself is implemented in. A considerably more light-weight approach is to use the constructs of an existing programming language to add on high-level language features so that the DSL features are effectively superposed onto the conventional language. This approach is known as implementing an "internal DSL" and is the one we employ in this paper, where we use the Python programming language as the substrate for our deep learning DSL.

The DSL approach [43] is particularly powerful when one can abstract a major set of operations and data structures together into a back-end framework or library and allow them to be invoked by the application user through appropriate compact programming language features in a spanning DSL. DSLs are particularly effective when a whole family of problems [4] can be identified. Once a subset of special cases have been solved, it is often feasible and efficient to inductively design a DSL framework [32] that then addresses the whole family of problems rather than continuing to solve each individually. This is a very productive approach.

Spinellis [54] described some well known usage patterns of DSLs but although ideas such as language-oriented programming [66] have been reported in the programming literature since 1994, the development and deployment of DSLs is still a relatively new area with most activity reported only over the last 15 years [63, 19, 43, 39]. Many application domains including: simulations [25]; business applications [51]; image processing [55]; database systems [41]; materials physics problems [26], or other complex systems [27] could be supported in this manner [63, 54]. In this present paper we develop a DSL for deep-learning applications.

DSLs are used in generating programming languages and tools themselves [16, 59], but other areas of reported successful DSL use to date include: communications and telephony [50, 15]; real-time- embedded systems [24] and field programmable gate array device deployment applications [13]; distributed and computational grid applications [33]; and mathematical [9] and equation-based problem formulation [40, 28, 57]. Parallel computing [38] is also a promising area for use of DSLs, whereby multiple versions of a program suited to different parallel architectures could be generated by a single DSL specification.

DSLs approaches are thus now being employed in many application areas [12]. At the time of writing, their use is still not completely widespread although this appears to be accelerating as better development tools become available and as more positive user and programmer experiences are reported.

3. DEEP LEARNING FRAMEWORKS

Deep Learning can be approached in a number of ways. In this section we present some background on principle approaches, followed by a explanation of how existing libraries can be used.

3.1 Overview of deep learning models

Two aspects that are particular relevant to our present work are artificial neural network models and more specifically - recurrent neural nets. We present some background and terminology.

3.1.1 Artificial neural nets

An artificial neural network learns a hidden representation $\mathbf{h}$ of an input $\mathbf{x}$, and a mapping from $\mathbf{h}$ to an output $\mathbf{y}$. Input $\mathbf{x}$ is typically a sequence $\mathbf{x} = (x_1, \ldots, x_N)$, and output $\mathbf{y}$ can be a single value from a pre-specified set (in a classification task), a continuous numeric value (in a regression task) or a sequence $\mathbf{y} = (y_1, \ldots, y_M)$. The hidden representation $\mathbf{h}$ is defined as $\mathbf{h} = f(x)$, where f is an activation function, such as sigmoid, tangent or relu. During training, the goal is to minimise the loss L between the input and output:

$$L(x, y) = -\frac{1}{N} \sum_{n \in N} x_n \log y_n, \tag{1}$$

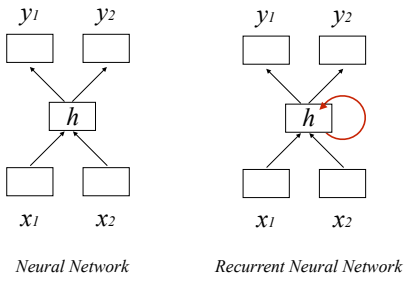

Figure 1: On the left, an artificial neural network. On the right, a recurrent neural network. Both architectures have two inputs, two outputs and a single hidden unit.

using e.g. cross entropy as a loss function. Figure 1 (left) shows a simple artificial neural net for illustration. It has two input symbols, one hidden node and two outputs.

3.1.2 *Recurrent neural nets*

A Recurrent neural net (RNN) is a type of neural network that learns a hidden representation $\mathbf{h}$ of an input sequence $\mathbf{x} = (x_1, \ldots, x_N)$ by learning an increasingly abstract encoding of the inputs. An RNN can also have an output sequence $\mathbf{y} = (y_1, \ldots, y_M)$, which is reconstructed from $\mathbf{h}$. Again, output $\mathbf{y}$ can take different forms depending on the learning task. The hidden representation $\mathbf{h}$ can be found through updates at time step t:

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, x_t), \tag{2}$$

where each update to $\mathbf{h}$ takes the context of the previous time step into account so that dependencies are learnt across input sequences. RNNs are suitable for tasks such as time-series data or natural language, where one input symbol can rely on the previous symbol(s). Figure 1 (right) shows a simple illustration of an RNN, where $\mathbf{h}$ is updated recursively.

Conventional update functions, such as sigmoid or tangent, have been associated with the problem of vanishing or exploding gradients [7]. A type of RNN that mitigates these problems is the long short-term memory (LSTM) [29]. In contrast to a conventional RNN, an LSTM has three gates, which control the loss and addition of information for the current "cell state". Each gate has the same shape as the hidden state. The "input gate" i is a sigmoid function which determines how much new available information to add to the cell state at the current time step. It first identifies for each member of the cell state vector whether it should be updated or not, and then chooses an update from a set of candidates. The "forget gate" f is a sigmoid function that determines for each member in the cell state vector whether it should be forgotten or retained. Finally, the "output gate" o determines what the output of the cell state should be.

In an LSTM, following [22], we update the hidden state $\mathbf{h}$ at each time step t using the following steps:

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i) \tag{3}$$

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f) \tag{4}$$

$$c_t = f_t c_{t-1} + i_t \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \tag{5}$$

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o) \tag{6}$$

$$h_t = o_t \tanh(c_t) \tag{7}$$

In Equations 3 - 7, σ refers to the logistic sigmoid function, and i, f, o and c refer to the input gate, forget gate, output gate and cell state vectors, respectively.

3.2 Use of existing libraries

A number of existing software libraries facilitate the development of deep learning models. Most of them focus on providing a mathematical framework that allows faster development of deep learning architectures, including e.g. efficient implementations of multi-dimensional arrays and tensors, gradients and activation functions. All common libraries also support code generation for CPU and GPU processing. Perhaps the most popular deep learning library is Theano [58], a Python library that allows users to define, optimise and evaluate mathematical expressions, and dynamically generate C code for faster expression evaluation. Torch [11] is similar to Theano in its functionality, but is implemented in Lua and OpenMP/SEE and CUDA for low-level operations. TensorFlow [1] is a further alternative for Python and C++ that is becoming increasingly popular in commercial applications. Finally, Caffe [30] is a library for convolutional neural nets and was developed primarily for computer vision applications.

All of the above libraries provide an extensive mathematical framework around deep learning, but do not provide actual implementations of neural networks. Keras [10] is a Python library that comes with a Theano and a TensorFlow backend. It contains a well-developed library of deep learning architectures, optimisers, and support for training and evaluating a wide range of models, illustrated with example implementations. Keras is becoming increasingly popular as it allows developers to compare a range of deep learning models and parameter settings in a short time frame. Despite Keras' comprehensiveness, the development of new deep learning models always involves code for data preprocessing. This includes representing data in the correct shape anticipated by the learning algorithm, encoding and decoding inputs and outputs, preparation of data splits for training and testing, etc. These are relatively repetitive operations that involve a lot of boilerplate code that hardly changes from one application to the next. For example, outputs for binary classification tasks (see Section 5.1) are often represented as binary matrices of boolean values, while sequence-to-sequence learning tasks (see Section 5.3) require 3-dimensional representations of their inputs and outputs.

4. A DEEP LEARNING DSL

The general idea of our DSL is to integrate common operations involved in training, optimising, evaluating and visualising deep learning models into a set of high-level commands in order to save time, avoid rewriting code, introducing bugs, and generally providing a useable interface to deep learning.

4.1 Deep learning in DEFIne

DEFIne (DEep learning Fluent Interface language) is a domain-specific language internal to Python, using particularly its numpy, keras and theano libraries for fast and efficient evaluation of multi-dimensional arrays and C code generation that is GPU-compatible. It is implemented as a

Figure 2: Basic operations involved in preparing a dataset for deep learning (top); a list of possible parameters that define the learning model, optimisation process and evaluation (middle); and common operations involved in training a deep learning model (bottom).

```
# Operations on dataset
data = DataSet(X_set, Y_set)
data.representData()
data.shuffleAndSplit()

# Parameter list for deep learner
parameters = {
modelString : MLP,
layers : 2,
batch_size : 32,
epochs : 20,
hidden_size : 50,
embedding_size : 1000,
learning_rate : 0.01,
momentum : 0.9,
optimiser : adam,
loss : categorical_crossentropy,
eval_metrics : [accuracy]
}

# Methods for deep learner
dl = defineDL(parameters)
dl.designModel(data)
dl.compileModel()
dl.loadWeights(weights_in_file)
dl.trainModel(data, out_file="weights.h5",
    verbose=False)
```

fluent interface using method chaining for all methods that refer to a common object, such as `DataSet` or `DeepLearner`. A fluent interface [19] is one where syntactic features of the hosting language are used to good effect to construct an internal DSL that captures the jargon, the commands and other notions of the requisite application domain. This approach has been widely used with the Java language [20] employing language features such as for-each iteration [31]. The use of Python [48] as a base for fluent interface programming is still in its infancy. Python lends itself well to the development of a fluent interface and we show in this section how the use of Python's *self* object self-reference feature allows this to be implemented. Python is an effective scripting language that is now quite widely used as a "glue" like language to build libraries and frameworks for particular application domains and to embed non native Python codes.

Our DSL implementation currently focuses on two main object types and their methods, shown in Figure 2. All methods were designed so as to correspond to a sequence of actions that are typically carried out when pre-processing a dataset and training a deep learning model. For example, a `DeepLearner` object is created based on a set of parameters. The general model architecture is then determined based on the parameters and the data. This model is compiled to contain an optimiser and loss function. Optionally, a set of pre-trained weights can be loaded as prior knowledge to the domain. Finally, the deep learner is trained and evaluated. As each of these operations rely (to an extent) on the previous operations, method chaining is a convenient way to achieve more readability and make the model easy to configure. While Figure 2 presented all operations as a sequence, we can chain them together as shown in Figure 3. Method chaining is implemented in Python by each method returning the `self` keyword as a return value.

4.1.1 Data representation

The `DataSet` object receives a multi-dimensional array X as input which contains the input features, and an array Y which contains the output labels. Y can either be one-dimensional (for single output scenarios) or multi-dimensional for sequence outputs. The method `representData()` will analyse the shape of both datasets and determine the best way to represent them, either using a 2-dimensional or a 3-dimensional representation. It will also analyse whether the data is numeric or symbolic, and in the latter case create a mapping dictionary from symbols to indices. This allows symbolic data (such as words) to be represented as numpy arrays during training, which is much faster than lists. Finally, the `shuffleAndSplitData()` method, as its name suggests, shuffles X and Y in unison and prepares training and test sets.

4.1.2 Parameters and model definition

Figure 2 in the middle shows a list of parameters that a user can set regarding the deep learning model itself (modelString, layers, hidden_size, embedding_size), the training setup (batch_size, epochs), optimisation (learning_rate, momentum, optimiser, loss) and evaluation (eval_metrics). All parameters except modelString are optional and will default to pre-specified standard values if not set.

The method `defineMode()` receives these parameters as input and will create a new instance of a `DeepLearner`. The method `designModel()` will add layers to the model, an input layer, a specified number of hidden layers, an output layer and an activation. `compileModel()` then adds an optimiser, a loss function and evaluation metrics against which to check the model's progress. The method `loadWeights()` can optionally load pre-trained weights to the model if training is not to start from scratch. Finally, `trainModel()` trains the model in a verbose on non-verbose fashion and saves its output weights to a file.

4.2 Visualisation

Unlike other machine learning frameworks such as supervised, unsupervised or reinforcement learning, which normally provide some insights to their users on the rationales for their decisions, deep learning models operate as a black-box. They do not allow for an easy inspection of the features and patterns that give rise to specific network decisions. This aspect can be an important limitation in safety-critical applications in health care or security, where it is important to understand the network's reasoning in order to trust its decisions. While this is an active area of ongoing research [45, 37], DEFIne provides a basic set of visualisation options to shed some light on the learning process and outcomes.

Scatterplots of data point embeddings.

With a large enough data set, it is possible to find patterns of statistical co-occurrence between data points in a

Figure 3: Method chaining implementing a fluent interface for preparation of a dataset and specification of a deep learner.

```
# Operations on dataset
data = DataSet(X_set, Y_set)
data.representData().shuffleAndSplit()

# Method chaining for deep learner
dl = defineDL(parameters).designModel(data).compileModel().loadWeights(weights_in_file).
    trainModel(data, out_file="weights.h5", verbose=False)
```

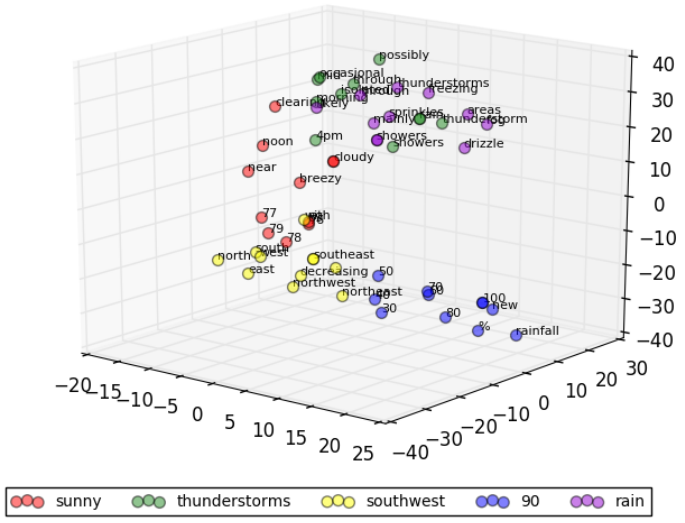

Figure 4: Illustration of learnt word embeddings and respective clusters in the weather forecast domain—words that are close together occur in similar contexts in the data.

k-dimensional space, where k is often between 50 and 100, but depends on the size of the data set and number of data points. Using dimensionality reduction [61], these multi-dimensional embeddings can then be mapped into a 2-dimensional or 3-dimensional space for visualisation. Figure 4 shows an example from the weather forecast domain, where data points correspond to words. In particular, words that are shown together in the plot tends to occur in similar contexts in the data. Neural networks are generally able to learn such embeddings from data, see e.g. [60] for an overview.

Image reconstructions.

As neural networks can generally learn a mapping from an input to an output, it is also possible to learn to reconstruct inputs from features. Possible applications are the generation of text that mimics the style and word sequences found in the input data, but without being semantically controlled [44, 56] or the reconstruction of images from learnt features. The latter is a frequent application of autoencoders that can be used for data compression [64]. Figure 6 shows example reconstructions of each of the digits [0-9] in the MNIST dataset for hand-written digit recognition, see Section 5.2.

Heat maps.

Heat maps can illustrate the strength of some input-output mappings learnt during training. While it is not possible in deep learning models to inspect the exact input features that are most predictive of a certain output, we can visualise the model's confidence in its output decisions. An example of this is shown in Figure 9, where the learning task was to learn correspondences between weather measurements (the input features, shown on the left) and word sequences "sunny", "cloudy", "clear", "mostly sunny", ..., "partly sunny", etc. We can see that some mappings are learnt with high confidence while others contain uncertainty.

Table 1: Feature set for heart disease diagnosis

Feature	Description	Max
age	age in years	77
sex	patient's gender (0 or 1)	1
cp	chest pain type	4
trestbps	resting blood pressure	200
chol	serum cholestorol	564
fbs	fasting blood sugar	1
restecg	resting electrocardiograph	2
thalach	max heart rate achieved	202
exang	exercise induced angina	1
oldpeak	ST depression induced by exercise relative to rest	6.2
slope	slope of peak exercise ST segment	3
ca	no of major vessels coloured by fluoroscopy	3
thal	3=normal, 6= fixed effect, 7=reversible defect	7
num	diagnosis of heart disease (0 or 1)	1

5. BENCHMARKS TASKS

We evaluate our DSL on three benchmark tasks in different areas that are relevant to real world applications of deep learning—medical diagnosis, image recognition and data-to-text generation.

5.1 Heart Disease Diagnosis

The Heart Disease dataset is available from the UCI Machine Learning Repository[1]. Table 1 shows the feature set for this classification task. All features except male/female were normalised, and the potential outputs were reduced to 0 (no disease present) and 1 (disease present).

The learning task is a classical classification task where we

[1]http://archive.ics.uci.edu/ml/machine-learning-databases/heart-disease/ (we used the processed Cleveland data)

Figure 5: Code implementing a neural net for heart disease diagnosis. Note that we are able to flexibly define new optimisers, such as Stochastic Gradient Descent with tailored parameters within the same framework.

```
data = DataSet(X_set, Y_set)
data.representData().shuffleAndSplit()

parameters = {
modelString : MLP,
layers : 1,
epochs : 200,
hidden_size : 6,
optimiser : SGD(lr=0.32, momentum=0.73)
}

defineDL(parameters).designModel(data).
    compileModel().trainModel(data)
```

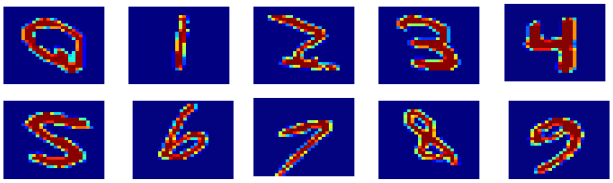

Figure 6: Examples of reconstructed hand-written digits from MNIST dataset.

want to find a mapping between a set of 13 numeric input features and one of two output features. Using our DSL, we define the model as shown in Figure 5. As can be seen, we are able to specify a number of learning parameters, including the learning rate and momentum, while leaving others unspecified. As the heart disease dataset is a comparatively small dataset for deep learning experiments, we can include human prior knowledge by setting the learning rate high to start with. This will accelerate learning in comparison to the low default learning rate of 0.01. The number of inputs and outputs as well as their dimensionality (1D, 2D, 3D) will be determined automatically by the DSL. The dataset contains 303 examples, which we split into training and test data in a 90%-10% ratio. Results in Section 6 are averaged over 10 runs.

5.2 Hand-written Digit Recognition

The MNIST hand-written digit recognition dataset[2] is a classical example of an image recognition task. Images are represented as 28*28 pixel matrices and there are 10 discrete output labels in the range of 0-9. Figure 6 shows sample reconstructions of each digit. MNIST contains 60,000 training examples and 10,000 test examples. We use the original split that the dataset is provided with. We can specify a deep learning model for the MNIST task as shown in Figure 7. This time we need to pre-process our data in a slightly different way as the MNIST dataset is already partitioned into training and test sets, and we need to flatten matrices into (784,)-shaped arrays. As our framework is build on Python and Keras, we can still combine our DSL with other library operations in situations where it is needed.

[2]http://www.iro.umontreal.ca/~lisa/deep/data/mnist/mnist.pkl.gz

Figure 7: Code implementing a neural net for hand-written digit recognition. Note the more elaborated data pre-processing steps required due to the MNIST data coming pre-split into training and test sets.

```
(X_train, Y_train), (X_test, Y_test) =
    mnist.load_data()
X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)
data = DataSet(X_train, Y_train)

data.Y_train = np_utils.to_categorical(
    Y_train, data.outputs)
data.Y_val = np_utils.to_categorical(Y_test
    , data.outputs)

parameters = {
modelString : MLP,
layers : 2,
epochs : 100,
hidden_size : 500
}

defineDL(parameters).designModel(data).
    compileModel().trainModel(data)
```

5.3 Weather Forecast Generation

Our final benchmark is the generation of weather forecasts[3] from meteorological measurements. This task differs from the previous ones in that we require a sequence-to-sequence model. We want to learn a mapping from an input sequence $\mathbf{x} = (x_1, \ldots, x_N)$ to an output sequence $\mathbf{y} = (y_1, \ldots, y_M)$, where we assume that $\mathbf{x}$ and $\mathbf{y}$ can have different lengths. The goal is to learn a probability distribution that conditions a target sequence (i.e. a sequence of words representing a weather forecast) on a source se-

[3]The dataset is available from http://cs.stanford.edu/~pliang/papers/weather-data.zip.

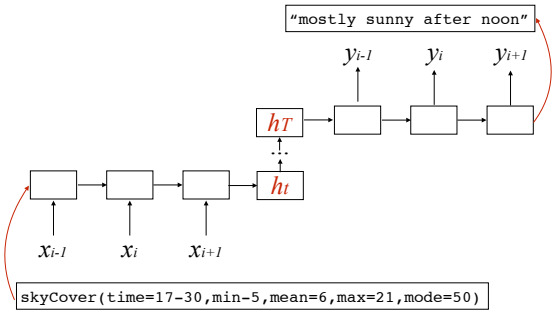

Figure 8: Sequence-to-sequence LSTM learning a mapping from an input sequence $\mathbf{x}$ of meteorological measurements to an output sequence $\mathbf{y}$ of words.

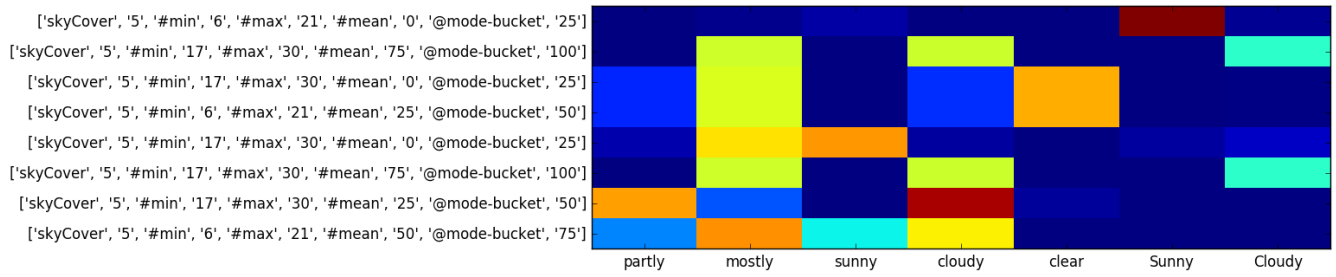

Figure 9: Heat map illustrating the model's confidence in its output decisions. Dark red indicates high confidence, dark blue indicates low confidence. In this example, the model learns to map "skyCover" measurements to word sequences.

Table 2: Feature set for weather forecast generation. Output features are individual words.

Feature	Description
id	a unique ID assigned to the weather record
type	type of weather described, e.g. *skyCover*, *temperature*, *rainChance*, etc.
time	time of the measurement
min	minimum value of measurement
max	maximum value of measurement
mean	mean value of measurement
mode	alternative value of measurement, e.g. for *windDirection*: south, southeast, etc.

quence (i.e. a sequence of measurements). An illustration of a sequence-to-sequence model is shown in Figure 8, where the hidden representation **h** can have multiple layers. For our example, we choose 4 layers and an LSTM architecture.

Figure 10 shows the code for this model. We require a recursive loop that defines one deep learning model per weather phenomenon, e.g. *skyCover*, *precipPotential*, etc. over 15 phenomena. This is needed because each weather phenomenon is expressed differently, so that each model has the same input features (measurements) but different output features (words). Input features are shown in Table 2. The sequence-to-sequence task requires a 3-dimensional representation of both input and output sequences, which will be determined automatically. We split our dataset of 29,528 examples into training and test instances in a 90%-10% ratio and average results over 10 runs.

6. EVALUATION

We evaluate the implementations of our three benchmark tasks in terms of accuracy, runtime and lines of code. We compare accuracy achieved by our model against its corresponding Keras implementation as well as against state-of-the-art results. This comparison is intended to make sure that our DSL does not compromise performance and achieves the same results as competing frameworks. The comparison in terms of runtime (in seconds) will give an indication of how much time we sacrifice for the benefit of less / shorter code before starting the learning process. Finally, we compare the lines of code that we need to define a model in comparison to Keras. All Keras comparisons use its Theano backend.

Figure 10: Code implementing an LSTM that learns to map a sequence of measurements to a sequence of words. This example iterates through a number of models (corresponding to subsets of data) and trains a model for each.

```
# Requires a dictionary "models" of tuples
    (X, Y) per deep learner to be created.

parameters = {
modelString : LSTM,
layers : 4,
epochs : 2000,
hidden_size : 20
}

# Create a deep learner per weather task.
for key in models:
  data = DataSet(models[key][0], models[key
      ][1])
  data.representData().shuffleAndSplit()
  dl = defineDL(parameters).designModel(
      data).compileModel().trainModel(data)
```

Accuracy / Similarity with gold standard.

Table 3 shows the training accuracy achieved by our models. This measure is intended to show that our DSL does not compromise the quality of deep learning models in any way and achieves comparable performance to an equivalent implementation in Keras. This is confirmed for all three benchmarks where small differences in the accuracy perceived are negligible and could be down to different data splits in training and testing. The last column provides a comparison with state of the art results (SOA). For the latter, we compare particularly with deep learning approaches.

For the heart disease diagnosis task, we compare with [46] who achieve 85%—a slightly better performance than our model even when using the same learning parameters on a neural net as us. [49] claims an accuracy of 90% but unfortunately does not provide her network parameters for replicability. Most other approaches use supervised learning.

For the MNIST dataset, the highest accuracy achieved that we are aware of is 99.9% [65] with a 2-layered convolutional neural net and DropConnect, a technique similar to drop-out in neural nets. Note that MNIST is not a difficult dataset to model and is mostly used as a benchmark to

Table 3: Accuracy (in %) / BLEU*

Domain	DEFIne	Keras	SOA
Heart Disease	80.7%	79.3%	85%
Digit recognition	98.2%	98.4%	99.9
Weather	0.65*	0.65*	0.52*

Table 4: Runtimes in seconds

Domain	DEFIne	Keras	% improvement
Heart Disease	1.82	1.79	1.68%
Digit recognition	2.77	2.71	2.21%
Weather	26.62	45.08	-69.34%

confirm that an algorithm has been implemented correctly, rather than pushing the state of the art.

Finally for the weather forecast experiments, we compare our generated output against related work using the BLEU score [47] instead of accuracy. BLEU measures the similarity against a gold standard data set (the human examples in the training data in our case) in terms of 4-grams. Language outputs are conventionally not evaluated based on accuracy because there can be more than one legitimate way of expressing the same thing. BLEU scores are measured in the range of 0-1, and related work reports scores of 0.52 [3] and 0.34 [34] for the weather task. None of these methods use deep learning, however, but semi-supervised learning from aligned data and hypergraphs, respectively.

Runtime.

Table 4 shows runtime results (in seconds) for each of the benchmark tasks using our DSL and its equivalent Keras implementation. Measurements include data processing and model definition, but not the training times of the neural nets. All results were computed on a 2015 Macbook with 2.7GHz Intel Core i5 processor and 8GB in RAM.

We can see that our DSL is slightly slower for the heart disease and MNIST tasks but substantially faster for the recursive definition of weather models. The former trend is to be expected as additional code needs to be executed in our DSL before Keras is invoked. Invoking Keras directly is therefore expected to be faster. The latter case (weather forecast) likely leads to a longer execution time for Keras code because our Keras implementation does not contain objects for datasets and deep learners. We kept our implementation as close as possible to the Keras example models[4], which require methods for data encoding/decoding, model definition and compilation, etc. to be defined for each model separately. While this makes sense if a single model is defined, it might well lead to slower runtime results when defining multiple models recursively. Our DSL then saves time over this as objects (`dataSet`, `deepLearner`) come with certain operations pre-defined.

Lines of code.

Table 5 shows a comparison of the lines of code required by our DSL and by Keras. We can observe that between 50% and 80% of code can be hidden away in our DSL, thus substantially reducing the lines of code required for the same programmes.

[4]https://github.com/fchollet/keras/tree/master/examples

Table 5: Lines of code

Domain	DEFIne	Keras	% improvement
Heart Disease	9	57	533.3%
Digit recognition	13	28	115.4%
Weather	10	58	480.0%

7. CONCLUSIONS

We have described some preliminary investigations into the use of Python as a host language for an embedded DSL that facilitates the implementation of deep learning code in comparison to existing libraries, such as Keras. Our DSL DEFIne summarises important operations for data preprocessing and model definition for deep learning into a fluent interface of common operations. This avoids the duplication of boilerplate code and reduces the introduction of errors. It also includes operations for frequent visualisation options. Our main research contributions are: (1) a fluent DSL framework for automatic data analysis and preprocessing and corresponding choice of hyper-parameters for deep learning (Section 4), (2) an evaluation in three different application domains to test the DSL's flexibility across datasets (Sections 5 and 6), and (3) the reduction of programming code by a factor of 5 (Section 6). Results from three benchmark tasks in heart disease diagnosis, handwritten digit recognition and weather forecast generation are encouraging. In terms of model accuracy, we observe that our framework achieves equivalent performance to state-of-the-art baselines implemented in other libraries. The trade-off of runtime efficiency versus lowered program source code complexity seems a well worthwhile one, as it simplifies and compactifies the codes for the computational experiments in deep learning that we report, as manifested by a reduced number of lines of application domain programmer code.

8. ACKNOWLEDGMENTS

We acknowledge the VIPER high-performance computing facility of the University of Hull and its support team.

9. REFERENCES

[1] M. Abadi, A. Agarwal, P. Barham, and *et al.* TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques and Tools.* Addison-Wesley, second edition, 2007. ISBN 0-321-48681-1.

[3] G. Angeli, P. Liang, and D. Klein. A Simple Domain-Independent Probabilistic Approach to Generation. In *Proc. of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Cambridge, Massachusetts, 2010.

[4] D. Batory, C. Johnson, B. MacDonald, and D. von Heeder. Achieving extensibility through product-lines and domain-specific languages: A case study. *ACM Trans. Software Engineering and Methodology*, 11(2):191–214, April 2002.

[5] Y. Bengio. Learning Deep Architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–127, 2009.

[6] Y. Bengio. Practical recommendations for gradient-based training of deep architectures. *Neural Networks: Tricks of the Trade*, 7700:437–478, 2012.

[7] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[8] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, pages 281–305, 2012.

[9] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, D. W.-F. J. Turian, and Y. Bengio. Theano: A CPU and GPU Math Expression Compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy) 2010*, Austin, TX, USA., June 2010.

[10] F. Chollet. Keras. https://github.com/fchollet/keras, 2016.

[11] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

[12] J. Cong. Overview of center for domain-specific computing. *Journal of Computer Science and Technology*, 26:632–635, 2011.

[13] J. Cong, V. Sarkar, G. Reinman, and A. Bui. Customizable domain-specific computing. *IEEE Design & Test of Computers*, March/April:6–14, 2011.

[14] C. Consel. Domain specific languages: What, why, how. *Electronic Notes in Theoretical Computer Science*, 65:1, 2002.

[15] C. Consel and L. Réveillère. A dsl paradigm for domains of services: A study of communication services. In *Domain-Specific Program Generation*, pages 165–179, 2003.

[16] K. Czarnecki, J. T. O'Donnell, J. Striegnitz, and W. Taha. Dsl implementation in metaocaml, template haskell, and c++. In *Domain-Specific Program Generation*, pages 51–72, 2003.

[17] X. Ding, Y. Zhang, T. Liu, and J. Duan. Deep Learning for Event-Driven Stock Prediction. In *Proc. of the 24th Joint International Conference on Artificial Intelligence (IJCAI)*, Beijing, China, 2015.

[18] P. Druzhkov and V. Kustikova. A survey of deep learning methods and software tools for image classification and object detection. *Pattern Recognition and Image Analysis*, 26(1):9–15, 2016.

[19] M. Fowler. *Domain-Specific Languages*. Number ISBN 0-321-71294-3. Addison Wesley, 2011.

[20] S. Freeman and N. Pryce. Evolving an embedded domain-specific language in java. In *Proc. OOPSLA'06*, pages 855–865, Portland, Oregon, USA, 22-26 October 2006.

[21] D. Ghosh. Dsl for the uninitiated - domain-specific languages bridge the semantic gap in programming. *Communications of the ACM*, 54(7):44–50, 2011.

[22] A. Graves. Generating Sequences With Recurrent Neural Networks. *CoRR*, abs/1308.0850, 2013.

[23] H. Greenspan, B. vanGinneken, and R. Summers. Guest Editorial Deep Learning in Medical Imaging: Overview and Future Promise of an Exciting New Technique. *IEEE Transactions on Medical Imaging*, 35(5), 2016.

[24] K. Hammond and G. Michaelson. The design of hume: A high-level language for the real-time embedded systems domain. In *Domain-Specific Program Generation*, pages 127–142, 2003.

[25] K. A. Hawick. Engineering internal domain-specific language software for lattice-based simulations. In *Proc. Int. Conf. on Software Engineering and Applications*, pages 314–321, Las Vegas, USA, 12-14 November 2012. IASTED.

[26] K. A. Hawick. Fluent interfaces and domain-specific languages for graph generation and network analysis calculations. In *Proc. Int. Conf. on Software Engineering (SE'13)*, pages 752–759, Innsbruck, Austria, 11-13 February 2013. IASTED.

[27] K. A. Hawick and H. A. James. Performance, scalability and object-orientation in discrete graph-based simulation models. In *Int. Conf. on Modeling, Simulation and Visualization Methods (MSV'05)*, pages 25–31, Las Vegas, USA, 27-30 June 2005. CSREA. ISBN 1-932415-70-X.

[28] K. A. Hawick and D. P. Playne. Automatically Generating Efficient Simulation Codes on GPUs from Partial Differential Equations. Technical Report CSTN-087, Computer Science, Massey University, Albany, North Shore 102-904, Auckland, New Zealand, July 2010.

[29] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, Nov. 1997.

[30] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[31] J. Kabanov, M. Hunger, and R. Raudjarv. On designing safe and flexible embedded dsls with java 5. *Science of Computer Programming*, 76:970–991, 2011.

[32] M. Karlsch. model-driven framework for domain specific languages demonstrated on a test automation language. Master's thesis, Hasso-Platner-Institute of Software Systems Engineering, Potsdam, Germany, 2007.

[33] K. Kennedy, B. Broom, K. Cooper, J. Dongarra, R. Fowler, D. Gannon, L. Johnsson, J. Mellor-crummey, and L. Torczon. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel and Distributed Computing*, 61:1803–1826, 2001.

[34] I. Konstas and M. Lapata. Unsupervised Concept-to-Text Generation with Hypergraphs. In *Proc. of the North American Chapter of the Association for Computational Linguistics (NAACL)*, Montreal, Canada, 2012.

[35] T. Kosar, N. Oliveira, M. Mernik, M. J. V. Pereira, matej Crepinsek, D. da Cruz, and P. R. Henriques. Comparing general-purpose and domain-specific languages: An empirical study. *Computer Science and Information Systems*, 7(2):247–264, May 2010.

[36] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E.

Perry, and W. M. Turski. Metrics and laws of software evolution - the nineties view. In *Proc. 4th Int. Symp. on Software Metrics (METRICS'97)*, Albuquerque, NM, USA, 5-7 November 1997. ISBN:0-8186-8093-8.

[37] T. Lei, R. Barzilay, and T. Jaakkola. Rationalizing Neural Predictions. In *Proc. of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Austin, Texas, 2016.

[38] C. Lengauer. Program optimization in the domain of high-performance parallelism. In *Domain-Specific Program Generation*, pages 73–91, 2003.

[39] C. Lengauer, D. Batory, C. Consel, and M. Odersky, editors. *Domain-Specific Program Generation*. Number 3016 in LNCS. Springer, 2003. ISBN 3-540-22119-0.

[40] A. Logg and G. N. Wells. Dolfin: Automated finite element computing. *ACM Trans. Math. Soft.*, 37(2):1–28, April 2010.

[41] C. A. Maddra and K. A. Hawick. Domain modelling and language issues for family history and near-tree graph data applications. In H. Arabnia, editor, *Proc. 14th Int. Conf. Software Engineering Research and Practice*, number SER3911, pages 10–16, Las Vegas, USA, 25-28 July 2016. WorldComp, CSREA Press. ISBN: 1-60132-446-4.

[42] C. Manning. Last Words: Computational Linguistics and Deep Learning. *Computational Linguistics*, 41(4):701–707, 2015.

[43] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, December 2005.

[44] T. Mikolov, M. Karafiát, L. Burget, J. Cernocký, and S. Khudanpur. Recurrent neural network based language model. In *INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September 26-30, 2010*, pages 1045–1048, 2010.

[45] V. Mnih, N. Heess, A. Graves, and K. Kavukcuoglu. Recurrent Models of Visual Attention. In *Proceedings of NIPS*, 2014.

[46] E. Olaniyi, O. Oyedotun, and K. Adnan. Heart Diseases Diagnosis Using Neural Networks Arbitration. *International Journal of Intelligent Systems and Applications*, 7 (12), 2015.

[47] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. BLEU: A Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics (ACL)*, pages 311–318. Association for Computational Linguistics, 2001.

[48] Python Software Foundation. The python programming language, 2007.

[49] U. Rani. Analysis of Heart Disease Dataset Using Neural Network Approach. *International Journal of Data Mining & Knowledge Management Process (IJDKP)*, 1 (5), 2011.

[50] D. A. Sadilek. Prototyping and simulating domain-specific languages for wireless sensor networks. Technical report, Humboldt-Universitat zu Berlin, Institute for Computer Science, 2007.

[51] M. Schuts and J. Hooman. Industrial Application of Domain Specific Languages Combined with Formal Techniques. In *Proc. of the 1st Workshop on Real World DSLs*, Barcelona, Spain, 2016.

[52] R. W. Sebesta. *Concepts of Programming Languages*. Pearson, ninth edition edition, 2009. ISBN 978-0-13-607347-5.

[53] D. Silver, A. Huang, and *et al.* Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[54] D. Spinellis. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56:91–99, 2001.

[55] R. Steward. An Image Processing Language: External and Shallow/Deep Embeddings. In *Proc. of the 1st Workshop on Real World DSLs*, Barcelona, Spain, 2016.

[56] I. Sutskever, J. Martens, and G. Hinton. Generating text with recurrent neural networks. In L. Getoor and T. Scheffer, editors, *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 1017–1024. ACM, June 2011.

[57] A. R. Terrel. From equations to code: Automated scientific computing. *Computing in Science & Engineering*, March/April:78–82, 2011.

[58] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[59] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *Proc. ACM PLDI'11*, pages 132–141, San Jose, California, 4-8 June 2011.

[60] P. Turney and P. Pantel. From Frequency to Meaning: Vector Space Models of Semantics. *Journal of Artificial Intelligence Research*, 37:141–188, 2010.

[61] L. van der Maaten and G. Hinton. Visualizing High-Dimensional Data Using t-SNE. *Journal of Machine Learning Research*, 9):2579–2605, 2008.

[62] A. van Deursen and P. Klint. Little languages: little maintenance. *Journal of Software Maintenance: Research and Practice*, 10(2):75–92, 1998.

[63] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Notices*, 35:26–36, June 2000.

[64] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol. Extracting and Composing Robust Features with Denoising Autoencoders. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 1096–1103, 2008.

[65] L. Wan, M. Zeiler, S. Zhang, Y. L. Cun, and R. Fergus. Regularization of Neural Networks using DropConnect. In S. Dasgupta and D. Mcallester, editors, *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, volume 28 (3), pages 1058–1066. JMLR Workshop and Conference Proceedings, May 2013.

[66] M. Ward. Language oriented programming. *Software - Concepts and Tools*, 15(4):147–161, 1994.