Machine Learning in Python with No Strings Attached

Guillaume Baudart
IBM Research
USA

Martin Hirzel
IBM Research
USA

Kiran Kate
IBM Research
USA

Louis Mandel
IBM Research
USA

Avraham Shinnar
IBM Research
USA

Abstract

Machine-learning frameworks in Python, such as scikit-learn, Keras, Spark, or Pyro, use embedded domain specific languages (EDSLs) to assemble a computational graph. Unfortunately, these EDSLs make heavy use of strings as names for computational graph nodes and other entities, leading to repetitive and hard-to-maintain code that does not benefit from standard Python tooling. This paper proposes eliminating strings where possible, reusing Python variable names instead. We demonstrate this on two examples from opposite ends of the design space: Keras.na, a light-weight wrapper around the Keras library, and Yaps, a new embedding of Stan into Python. Our techniques do not require modifications to the underlying library. Avoiding strings removes redundancy, simplifies maintenance, and enables Python tooling to better reason about the code and assist users.

CCS Concepts • **Software and its engineering** → *Domain specific languages.*

Keywords Machine Learning, Programming Languages

ACM Reference Format:
Guillaume Baudart, Martin Hirzel, Kiran Kate, Louis Mandel, and Avraham Shinnar. 2019. Machine Learning in Python with No Strings Attached. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL '19), June 22, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3315508.3329972

1 Introduction

Many machine-learning (ML) frameworks use Python, both because of its well-designed language features and for interoperability with other Python libraries. Most machine-learning frameworks in Python use *lazy evaluation*: they first build a computational graph, which they evaluate only later on training or test datasets. There are good reasons for this lazy-evaluation approach: it means the same computational graph can be used to both train and predict; it simplifies shifting some of the computation to GPUs or to clusters of multiple computers; it helps save a model for later use; and the framework can interpret or rewrite the computational graph for automatic differentiation, back-propagation, optimization, or Monte-Carlo sampling.

Since the lazy-evaluation approach makes ML frameworks look and feel like languages, they are often referred to as embedded domain specific languages (EDSLs) [15]. Most ML frameworks provide a way to attach strings to specific nodes in the computational graph, so they can be referenced either from elsewhere in the code (e.g., to set a hyperparameter) or from summaries or visualizations (e.g., to help users debug their code). Consider the following Keras [9] code snippet:

```
output = Dense(N_LABELS, activation='softmax',
               name='output')(dropped2)
```

We immediately notice one problem: the code *stutters* due to spurious repetition of the word `output`. But there are also deeper issues. When a framework attaches strings, users will want error-checking for how those strings are being used; they will want a scoping facility; they will want their editor to support navigation, code completion, refactoring, etc.; and they will want to see the strings in summaries or visualizations. This causes either extra framework implementation effort or simply missing features. While this example happens to be in Keras, Section 2 gives several other examples.

Since implementing all of the above-mentioned name-related features for a machine-learning framework is difficult, it would be nice if the host language Python could help somehow. Besides highlighting the problem, the above example also hints at a possible solution. Python already has a well-designed naming facility for variable names. The framework should be able to pick up the name `output` from:

```
output = Dense(N_LABELS, activation='softmax')
             (dropped2)
```

This requires *name reflection*, a way for the framework to obtain variable names to attach to the computational graph. Fortunately, name reflection turns out to be easy to implement in Python. So this leaves us with two ways to design EDSLs: *strings attached* or *names attached*. We advocate following the Zen of Python: "There should be one – and preferably

MAPL '19, June 22, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6719-6/19/06…$15.00
https://doi.org/10.1145/3315508.3329972

only one – obvious way to do it." When it comes to Python EDSLs, strings are a code smell, and the obvious way to go is to keep source variable names attached. Ideally, the other string in this code, `'softmax'`, should also be replaced by a name, perhaps using an enum.

We illustrate how to avoid attached strings with two examples, one lightweight wrapper around an existing library and one new from-scratch EDSL design and implementation. For the first example, we picked Keras, a deep learning framework that is popular for its ease-of-use [9]. Our lightweight wrapper, Keras.na (Keras retrofitted to keep names attached), improves programmability, visualization, and error reporting for Keras. A similar approach could improve assorted other existing machine-learning frameworks. For the second example, we picked Stan, a popular probabilistic programming language (PPL) that lets users express probabilistic models and train them with Monte-Carlo sampling, among other techniques [8]. There are existing libraries PyStan [8] and PyCmdStan [16] that use Stan syntax in Python strings or loaded from files. In contrast, our new EDSL, Yaps (Yet Another Pythonic Stan), reinterprets Python syntax with Stan semantics. We have open-sourced Yaps at https://github.com/ibm/yaps and put a description of Yaps on arXiv [2]. Besides avoiding attached strings, Yaps also prevents the host and guest language from leaking into each other, thus providing watertight abstractions for probabilistic programming in Python.

This paper makes the following contributions:

- Making the case that strings attached are trouble and names attached are a simple but effective remedy.
- Keras.na, a thin layer over the Keras deep-learning library using name reflection to keep names attached.
- Yaps, a new front-end for the Stan probabilistic programming language using reinterpreted Python.

We demonstrate both Keras.na and Yaps with screen-shots that show reduced stuttering in the code, better visualization, and better error reporting. We successfully test the Yaps compiler on 1,282 Stan programs. We hope that this paper influences EDSL designers to use our techniques for machine learning with no strings attached.

2 Strings Considered Harmful

This section makes the case for having no strings attached in Python-based machine learning frameworks. Unfortunately, there is no dearth of examples to draw from for frameworks with strings attached and their pitfalls. The issue occurs in both traditional machine learning (e.g., with scikit-learn [7]) and in deep learning (e.g., with Keras [9]). It is not limited to the core modeling stage of machine learning, but also manifests in data preparation (e.g., with Spark SQL [1]). And it also occurs in exciting new frameworks that combine deep learning with probabilistic programming (e.g., with Pyro [3]).

```
1 filter = SelectKBest(f_regression, k=5)
2 clf = svm.SVC(kernel='linear')
3 anv_svm = Pipeline([('anv', filter), ('svc', clf)])
4 anv_svm.set_params(anv__k=10, svc__C=.1).fit(X, y)
```

Figure 1. Example scikit-learn pipeline with strings attached, adapted from the official documentation at http://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html

This section uses idiomatic code examples in each of these frameworks to illustrate that strings are both pervasive and harmful. However, we do not want readers to come away with a negative impression. Each of the frameworks is widely-adopted, which is testament for its usability. We believe these frameworks are already great, and want to point out simple constructive changes to make them even better.

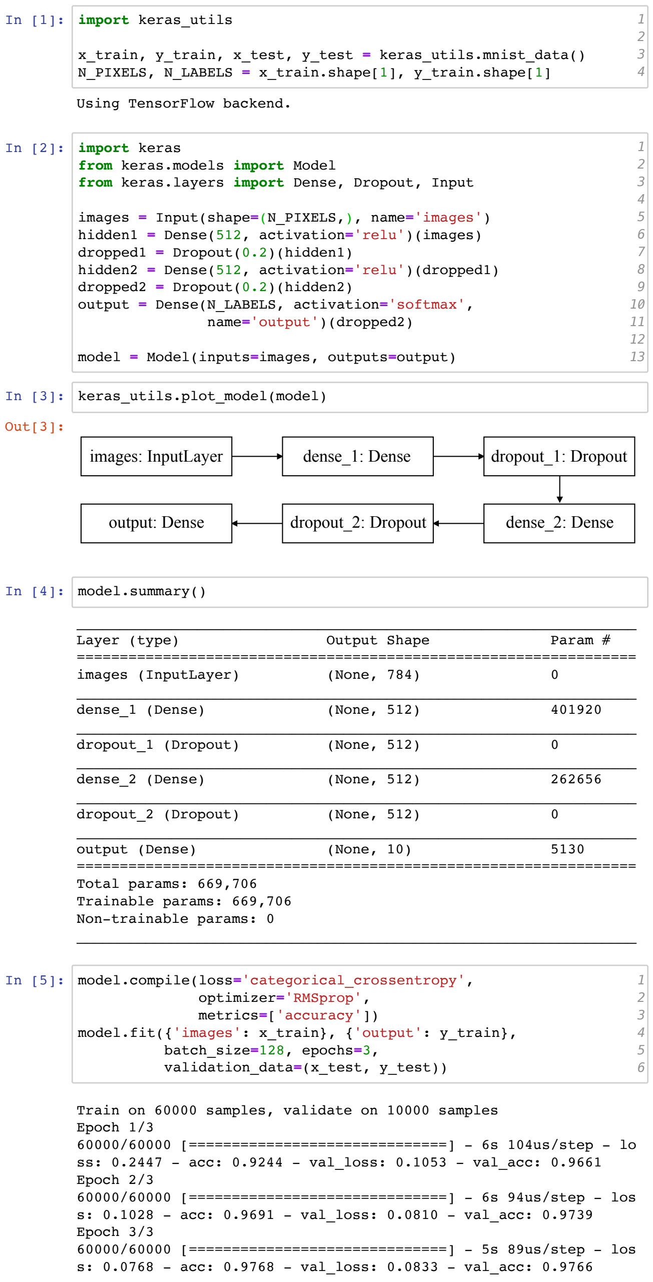
2.1 Strings in Scikit-Learn

The popular scikit-learn library is a comprehensive collection of off-the-shelf machine learning models [7]. Its `pipeline` module lets users compose ML models into a bigger computational graph. Figure 1 shows an idiomatic example.

Line 1 instantiates a feature-selector model and assigns it to the Python variable `filter`. It also sets two hyperparameters: `f_regression`, the statistical measure to use, and `k=5`, the number of features to select. Line 2 instantiates a support vector machine classifier and assigns it to Python variable `clf`. It also sets one hyperparameter, `kernel='linear'`. The `svm.SVC` class also has other hyperparameters, which have default values and are thus optional. At this point, both models, `filter` and `clf`, are trainable but not yet trained.

Line 3 creates a computational graph consisting of the two trainable models `filter` and `clf`, and assigns the entire graph to variable `anv_svm`. The graph is constructed with lazy evaluation: it is itself a trainable model, storing the hyperparameter configuration and the graph topology, but not yet trained on a dataset. In programming-languages terminology, this code uses *point-free style*, since rather than introducing intermediate variables for intermediate results (points), it defines the function composition `filter` ∘ `clf`. Line 3 explicitly attaches strings to the graph nodes, so they can be referenced from elsewhere in the code. The code would be shorter and more consistent if it did not attach strings to nodes that already have names.

Line 4 changes the hyperparameters on some graph nodes, then calls `fit` to evaluate the entire computational graph, training its constituent models on dataset `X,y`. This code is literally taken from the official scikit-learn documentation modulo minor edits for space. Even though it would have been possible to set the hyperparameters earlier already, the example emulates the typical case where hyperparameters are adjusted later, for instance to search their valid ranges for good values. The code uses Python's keywords parameters feature, along with a name mangling scheme that is



```
In [1]:  1 import keras_utils
         2
         3 x_train, y_train, x_test, y_test = keras_utils.mnist_data()
         4 N_PIXELS, N_LABELS = x_train.shape[1], y_train.shape[1]

Using TensorFlow backend.

In [2]:  1 import keras
         2 from keras.models import Model
         3 from keras.layers import Dense, Dropout, Input
         4
         5 images = Input(shape=(N_PIXELS,), name='images')
         6 hidden1 = Dense(512, activation='relu')(images)
         7 dropped1 = Dropout(0.2)(hidden1)
         8 hidden2 = Dense(512, activation='relu')(dropped1)
         9 dropped2 = Dropout(0.2)(hidden2)
        10 output = Dense(N_LABELS, activation='softmax',
        11                name='output')(dropped2)
        12
        13 model = Model(inputs=images, outputs=output)

In [3]: keras_utils.plot_model(model)

Out[3]:
```

```
In [4]: model.summary()

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
images (InputLayer)          (None, 784)               0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               401920
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 512)               262656
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0
_________________________________________________________________
output (Dense)               (None, 10)                5130
=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
_________________________________________________________________

In [5]:  1 model.compile(loss='categorical_crossentropy',
         2               optimizer='RMSprop',
         3               metrics=['accuracy'])
         4 model.fit({'images': x_train}, {'output': y_train},
         5           batch_size=128, epochs=3,
         6           validation_data=(x_test, y_test))

Train on 60000 samples, validate on 10000 samples
Epoch 1/3
60000/60000 [==============================] - 6s 104us/step - lo
ss: 0.2447 - acc: 0.9244 - val_loss: 0.1053 - val_acc: 0.9661
Epoch 2/3
60000/60000 [==============================] - 6s 94us/step - los
s: 0.1028 - acc: 0.9691 - val_loss: 0.0810 - val_acc: 0.9739
Epoch 3/3
60000/60000 [==============================] - 5s 89us/step - los
s: 0.0768 - acc: 0.9768 - val_loss: 0.0833 - val_acc: 0.9766
```

Figure 2. Example Keras model with strings attached.

specific to scikit-learn, using a double underscore to connect a graph node name with a hyperparameter name. This name mangling is not immediately obvious to novice users, can be brittle, and lacks support from development tooling.

2.2 Strings in Keras

Whereas scikit-learn is a library of classic machine learning models, Keras is a library of deep-learning layers [9]. Its main value is making it easy to compose layers into neural network architectures, i.e., computational graphs. Keras offers two APIs: a point-free style API akin to scikit-learn pipelines, and an API that lets users create layers one at a time and assign them to Python variables. Figure 2 shows an idiomatic example of the second API in a Jupyter notebook [17].

Cell [1] loads the MNIST dataset of handwritten digits via a small utility library we wrote to reduce boilerplate.

Cell [2] creates a computational graph using lazy evaluation. Each of Lines 5–10 instantiates a layer for a neural network with a call of the form

```
output_tensor = Layer(hyperparams)(input_tensor)
```

At this point, each of the layers is trainable but not yet trained. Two of the calls explicitly attach strings `'images'` and `'output'` to graph nodes, so they can be referenced from elsewhere in the code. In both cases, the code is stuttering, with spurious repetition of the output tensor variable name in a string. Section 3 will show how to make this code shorter by using name reflection. Cell [2] Line 13 puts the entire computational graph into a deep learning model.

Cell [3] visualizes the computational graph in the notebook. The visualization shows each node as label: LayerType. For nodes with strings attached, the label is the string, while for nodes without strings attached, the label is based on a name mangling scheme, using an underscore to connect the layer type with a unique integer. To figure out which node in the visualization corresponds to which line in the source code, one has to count. The more obvious way would be to show Python variable names in the visualization.

Cell [4] shows a summary of the computational graph. It uses the same labels as the visualization but shows additional shape information. The `None` dimensions will be bound to the minibatch size later during training, and the `Param #` indicates the number of floating point numbers for weights and biases to be learned during training.

Cell [5] evaluates the computational graph, training its constituent layers on the MNIST dataset. Line 4 uses the strings `'images'` and `'output'` to reference the computational graph nodes with these strings attached. Using Python variables instead of strings would enable existing Python tooling to help with scoping, code navigation, auto-completion, and checking for name-related errors.

2.3 Strings in Spark

Both of the examples so far, from scikit-learn and from Keras, focused on the core modeling and training stage of machine learning. However, quoting Domingos [10]:

> Often, the raw data it not in a form that is amenable for learning, but you can construct features from it that are. This is typically where most of the effort in a machine learning project goes.

Spark is a framework for data processing at scale. Spark SQL can prepare raw data to make it digestible by a downstream Spark ML model [1]. Figure 3 shows an idiomatic example.

```
1 people = sqlContext.read.parquet("people.parquet")
2 dept = sqlContext.read.parquet("dept.parquet")
3 people.filter(people.age > 30) \
4   .join(dept, people.deptId == dept.id) \
5   .groupBy(dept.name, "gender") \
6   .agg({"salary": "avg", "age": "max"})
```

Figure 3. Example Spark SQL program with strings attached.

Lines 1 and 2 use lazy evaluation to instantiate computational graph nodes that will later load the data from files.

Lines 3–6 express a query in point-free style, where each line uses lazy evaluation to instantiate a separate computational graph node. The nodes correspond roughly to logical operators from relational algebra, except that classic relational algebra would fuse the `groupBy` and `agg` operators to avoid violating normal form with nested tables.

Line 3 uses `people.age > 30` as the filtering predicate. This predicate itself uses lazy evaluation: Spark SQL overloads the > operator to build a computational graph to be evaluated later one-at-a-time for each row of the input data. Line 4 performs an equi-join, this time using a predicate by overloading the == operator.

Line 5 instantiates a group-by node in the computational graph. It groups the data by two features, `name` and `gender`. Since both input tables, `people` and `department`, have a `name` feature, the code disambiguates it to `department.name`. On the other hand, the `gender` feature is unique, so the code can reference it with a string. This code is literally taken from the Spark Dataframe documentation modulo minor edits for completeness[1]. Using two different approaches to reference features is inconsistent and the string-based approach lacks source code editor support. Line 6 instantiates an aggregation node in the computational graph. This code refers to features by strings `"salary"` and `"age"`, and it also refers to aggregation functions by strings `"avg"` and `"max"`.

2.4 Strings in Pyro

Pyro is a Python-based framework for deep probabilistic programming [3]. With probabilistic programming, one can define a model, fit it to data, and then use the trained model not just for predictions but also to quantify the uncertainty or to generate new data [11]. Pyro combines probabilistic programming with deep learning, so the probabilistic models can incorporate neural networks, for instance, with weight uncertainty [5]. Figure 4 shows an idiomatic example.

Deep probabilistic models are usually trained with explicit variational guides for variational inference (VI) [4]. A guide is a family of probability distributions and VI finds a member of that family that minimizes the divergence from the correct distribution according to the training data. In Figure 4, `coin` defines the model and `guide` defines the guide.

```
1 def coin():
2   bias = pyro.sample("bias", Uniform(0, 1))
3   pyro.sample("tosses", Bernoulli(bias))
4 def guide():
5   qalpha = pyro.param("qalpha", torch.randn(()))
6   qbeta = pyro.param("qbeta", torch.randn(()))
7   pyro.sample("bias", Beta(qalpha, qbeta))
```

Figure 4. Example Pyro model with strings attached.

Line 2 defines the computational graph node `bias`, representing a latent probabilistic variable to be sampled from a uniform probability distribution between 0 and 1. The code attaches a string `"bias"` to the node so it can be referenced from elsewhere. The code stutters, spuriously repeating the variable name in a string. Line 3 defines the node for an observed probabilistic variable `tosses`, which will be associated with the training data, and will be fit to a Bernoulli distribution parameterized by `bias`. Lines 5–6 create computational graph nodes for the parameters q_α and q_β of the guide. Again, the code attaches strings and stutters. Finally, Line 7 creates a node indicating that VI should minimize the divergence between a Beta($\mathrm{q}_\alpha, \mathrm{q}_\beta$) distribution and the true posterior. Corresponding computational graph nodes in the model and the guide must have the same string attached, in this case `"bias"`. Among other things, using strings instead of variables means that there is a single global shared namespace; this is a known and acknowledged Pyro issue [20].

3 Name Reflection: Keras.na

This section shows a lightweight wrapper for an existing machine-learning framework to attach variable names instead of strings. We picked Keras [9] for this demonstration, because it focuses on being user-friendly, which is exactly the quality our wrapper enhances further. Our wrapper is called Keras.na, where ".na" stands for "with names attached".

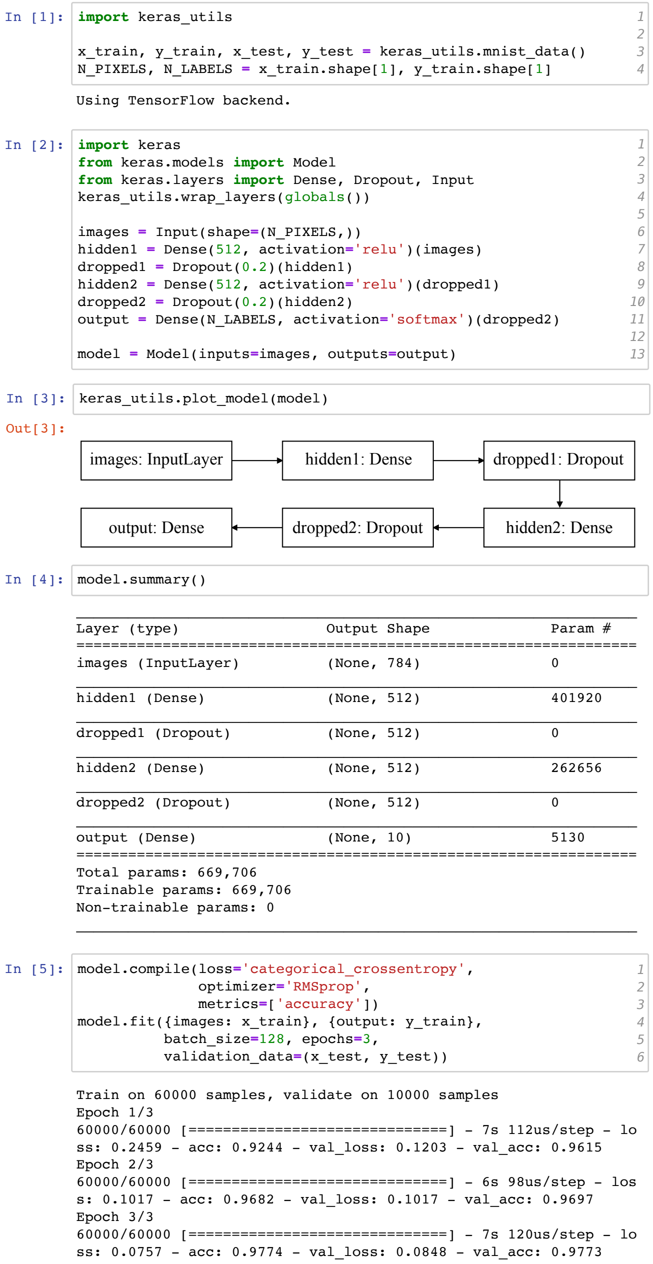
Figure 5 shows an idiomatic example of using Keras.na in a Jupyter notebook [17]. This is derived by changing the example from Figure 2 to use name reflection.

Cell `[1]` is unchanged, loading the data exactly as before.

Cell `[2]` Line 4 calls a utility function to install our Keras.na wrappers. The `globals()` argument is the current symbol table after imports. The `wrap_layers` function modifies the symbol table, replacing each Keras layer (including `Input`, `Dense`, and `Dropout`) by a wrapper. With these wrappers in place, the code for instantiating layers in Lines 6–11 can omit explicit `name='string'` arguments, avoiding the stuttering of the corresponding classic Keras code in Figure 2. One could also eliminate the remaining strings in this code with enum constants; we leave this to future work.

Cell `[3]` visualizes the model and Cell `[4]` summarizes the model. Both views demonstrate that computational graph nodes are labeled by Python variable names. This is accomplished by name reflection in the wrappers for `Input`, `Dense`, etc. Implementing name reflection turned out to be easy: the

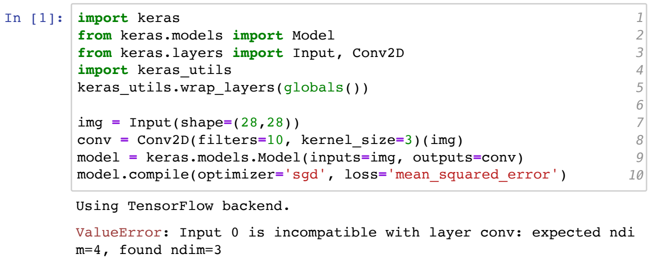
[1] https://spark.apache.org/docs/2.2.0/api/python/pyspark.sql.html#pyspark.sql.DataFrame



```
In [1]: import keras_utils

        x_train, y_train, x_test, y_test = keras_utils.mnist_data()
        N_PIXELS, N_LABELS = x_train.shape[1], y_train.shape[1]

        Using TensorFlow backend.

In [2]: import keras
        from keras.models import Model
        from keras.layers import Dense, Dropout, Input
        keras_utils.wrap_layers(globals())

        images = Input(shape=(N_PIXELS,))
        hidden1 = Dense(512, activation='relu')(images)
        dropped1 = Dropout(0.2)(hidden1)
        hidden2 = Dense(512, activation='relu')(dropped1)
        dropped2 = Dropout(0.2)(hidden2)
        output = Dense(N_LABELS, activation='softmax')(dropped2)

        model = Model(inputs=images, outputs=output)

In [3]: keras_utils.plot_model(model)

Out[3]:
```

```
In [4]: model.summary()

        _________________________________________________________________
        Layer (type)                 Output Shape              Param #
        =================================================================
        images (InputLayer)          (None, 784)               0
        _________________________________________________________________
        hidden1 (Dense)              (None, 512)               401920
        _________________________________________________________________
        dropped1 (Dropout)           (None, 512)               0
        _________________________________________________________________
        hidden2 (Dense)              (None, 512)               262656
        _________________________________________________________________
        dropped2 (Dropout)           (None, 512)               0
        _________________________________________________________________
        output (Dense)               (None, 10)                5130
        =================================================================
        Total params: 669,706
        Trainable params: 669,706
        Non-trainable params: 0
        _________________________________________________________________

In [5]: model.compile(loss='categorical_crossentropy',
                      optimizer='RMSprop',
                      metrics=['accuracy'])
        model.fit({images: x_train}, {output: y_train},
                  batch_size=128, epochs=3,
                  validation_data=(x_test, y_test))

        Train on 60000 samples, validate on 10000 samples
        Epoch 1/3
        60000/60000 [==============================] - 7s 112us/step - lo
        ss: 0.2459 - acc: 0.9244 - val_loss: 0.1203 - val_acc: 0.9615
        Epoch 2/3
        60000/60000 [==============================] - 6s 98us/step - los
        s: 0.1017 - acc: 0.9682 - val_loss: 0.1017 - val_acc: 0.9697
        Epoch 3/3
        60000/60000 [==============================] - 7s 120us/step - lo
        ss: 0.0757 - acc: 0.9774 - val_loss: 0.0848 - val_acc: 0.9773
```

Figure 5. Example Keras.na model (cf. Figure 2).



```
In [1]:  1  import keras
         2  from keras.models import Model
         3  from keras.layers import Input, Conv2D
         4  import keras_utils
         5  keras_utils.wrap_layers(globals())
         6
         7  img = Input(shape=(28,28))
         8  conv = Conv2D(filters=10, kernel_size=3)(img)
         9  model = keras.models.Model(inputs=img, outputs=conv)
        10  model.compile(optimizer='sgd', loss='mean_squared_error')

        Using TensorFlow backend.

        ValueError: Input 0 is incompatible with layer conv: expected ndi
        m=4, found ndim=3
```

Figure 6. Example Keras.na model with names preserved in error message from built-in Keras error checking.

Python traceback module returns a call trace including the source code of the call site, and the Python ast module can parse that source code to extract the variable name on the left-hand-side of the assignment. Putting Python variable names into the visualization and summary makes it easier for users to map them back to source code.

Cell [5] trains the model on the MNIST dataset. But whereas the corresponding classic Keras code in Figure 2 used strings to refer to computational graph nodes

```
{'images': x_train}, {'output': y_train},
```

the Keras.na code in Figure 5 uses variable names that the Python interpreter resolves to definitions from Cell [2].

```
{images: x_train}, {output: y_train}.
```

Getting this to work required modest additional implementation effort. In addition to wrapping Keras layers, wrap_layers also wraps the Keras Model class, overriding the fit method to remap the keys of the data dictionaries.

For backward compatibility, Keras.na uses the following heuristics to attach labels to computational graph nodes:

- If the user specifies an explicit name argument, use that. For example, the code
  ```
  aux_output = Dense(.., name='output2')(input)
  ```
  instantiates a layer with the label output2.
- Otherwise, if the output is assigned to a variable with a unique name (first occurrence wins), use that. For example,
  ```
  aux_output = Dense(..)(input)
  ```
  instantiates a layer with the label aux_output.
- Otherwise, fall back upon the Keras built-in automatic name mangling. Especially legacy code might make that necessary for several reason, e.g., non-unique names:
  ```
  h = Dense(..)(input)
  h = Dense(..)(h)
  ```
  Another reason is chaining multiple computational graph node creations in a single line of code:
  ```
  output = Dense(..)(Dense(..)(input))
  ```
  Another reason is using the point-free API of Keras:
  ```
  model = Sequential([Dense(..), Dense(..)])
  ```
  Finally, Keras even supports separating the layer instantiation from creating a computational graph node:
  ```
  d = Dense(..)
  y1 = d(x1)
  y2 = d(x2)
  ```

If backward compatibility is less important, one could disallow some of these cases with informative error messages.

To test whether Keras.na improves error messages, we created a model where we intentionally used a wrong tensor shape. Figure 6 shows what happens when we try to run that from a Jupyter notebook. Line 8 of the code assigns a computational graph node to Python variable conv. The Keras.na wrappers arrange for the node to be labeled conv. And the error message correctly mentions that variable name.

Section 2 argued that the problem of attached strings is pervasive in ML frameworks. While this section focused on Keras, name reflection is a general technique that could also be retrofitted to other frameworks discussed in Section 2.

4 Reinterpreted Python: Yaps

This section shows an EDSL designed from the start to keep strings attached. Our EDSL, Yaps, embeds the popular Stan language [8] for probabilistic programming into Python.

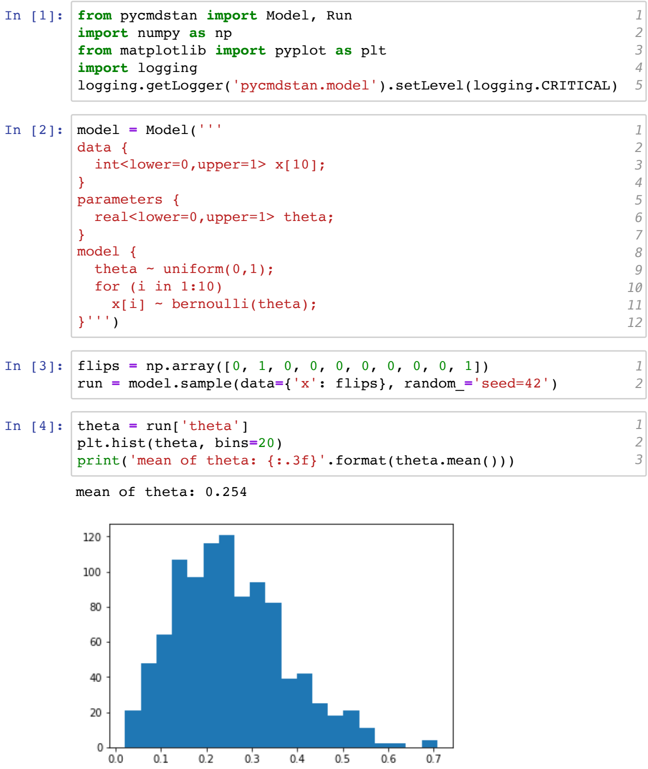
A *probabilistic model* is a mathematical model for explaining real-world observations as being generated from latent distributions [11]. Probabilistic models can be used for machine learning, and compared to alternative approaches, have the potential to make uncertainty more overt, require less labeled training data, and improve interpretability. The key *abstractions* for writing probabilistic models are *sampling* of latent variables and observations and *inference* of latent variables [12]. There are several stand-alone probabilistic programming languages, e.g., Stan [8]. To capitalize on Python's familiarity and for interoperability with other ML frameworks, PyStan [8], PyCmdStan [16], and other efforts [3, 21, 23] embed probabilistic abstractions into Python.

In programming, a *watertight abstraction* provides a basis for coding or debugging without *leaking* information about lower-level abstractions that it builds upon. Unfortunately, probabilistic abstractions in Python offered by existing efforts [3, 21, 23] are not watertight. To code with those packages, one must also use lower-level packages such as NumPy, PyTorch, or TensorFlow. Furthermore, bugs such as tensor dimension mismatches often manifest at those lower levels and cannot be reasoned about at the probabilistic level alone.

4.1 Yaps Example

Figure 7 shows an example program in PyCmdStan [16], an existing Python interface from prior work for which our new Yaps language offers an alternative. Cell `[1]` imports the library. Cell `[2]` puts the Stan code as a multi-line string into a Python variable `code`. Line 3 declares one observed variable `x`, representing ten coin tosses, each either tails (0) or heads (1). Line 6 declares a latent variable `theta` for the unknown bias of the coin. Line 9 uses `~` to sample `theta` with the prior that any bias is equally likely. Lines 10–11 indicate that each of the coin tosses `x[i]` is sampled from a Bernoulli distribution with the same latent `theta` parameter.

Cell `[3]` Line 1 contains concrete observed coin flips. Line 2 calls inference, attaching the Python string `'x'` to the observed data and passing the Stan code of the model to Stan to infer a joint posterior distribution. PyCmdStan creates a `run` object from which latent variables such as `'theta'` can be extracted. Cell `[4]` Line 1 performs the extraction. Since the Stan code is a Python string, the extraction also uses a string to refer to posterior variables. Line 2 prints the results: the inferred posterior belief is that the coin is biased towards tails (the mean of `theta` is 0.254).

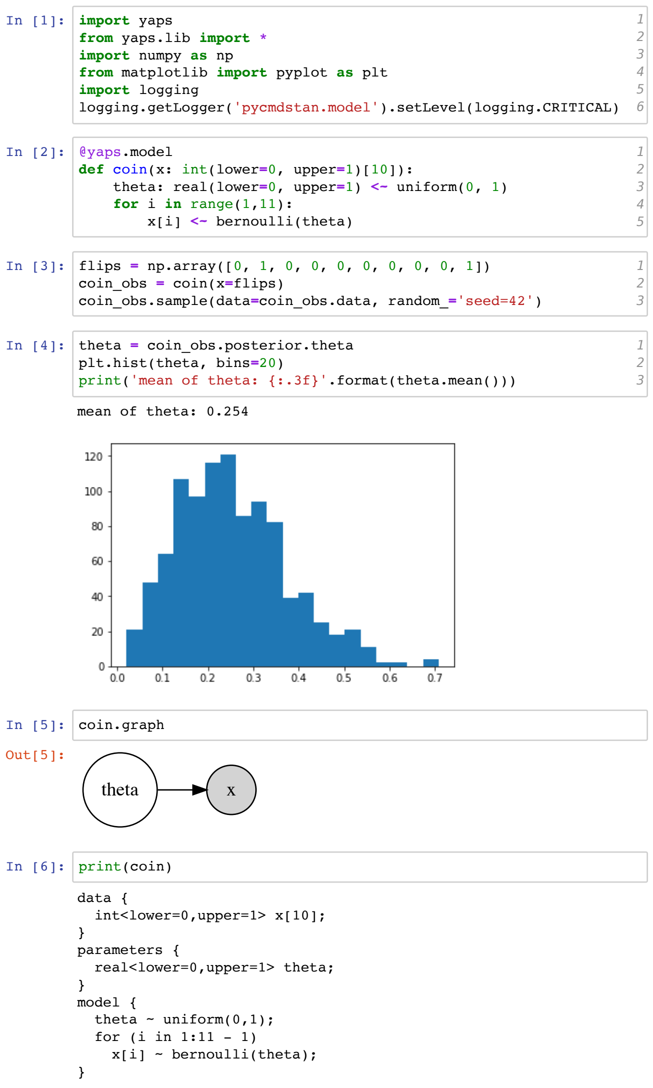


```
In [1]: from pycmdstan import Model, Run                                              1
        import numpy as np                                                            2
        from matplotlib import pyplot as plt                                          3
        import logging                                                                4
        logging.getLogger('pycmdstan.model').setLevel(logging.CRITICAL)               5

In [2]: model = Model('''                                                             1
        data {                                                                        2
          int<lower=0,upper=1> x[10];                                                 3
        }                                                                             4
        parameters {                                                                  5
          real<lower=0,upper=1> theta;                                                6
        }                                                                             7
        model {                                                                       8
          theta ~ uniform(0,1);                                                       9
          for (i in 1:10)                                                            10
            x[i] ~ bernoulli(theta);                                                 11
        }''')                                                                        12

In [3]: flips = np.array([0, 1, 0, 0, 0, 0, 0, 0, 0, 1])                              1
        run = model.sample(data={'x': flips}, random_='seed=42')                      2

In [4]: theta = run['theta']                                                          1
        plt.hist(theta, bins=20)                                                      2
        print('mean of theta: {:.3f}'.format(theta.mean()))                          3

        mean of theta: 0.254
```

Figure 7. Example PyCmdStan model with strings attached.

Figure 8 shows the same example program in Yaps. Cell `[1]` imports our `yaps` library. Cell `[2]` shows code in Python syntax that corresponds directly to the corresponding Stan code in Figure 7. Whereas Stan has explicit blocks, Yaps provides a more concise (but equally watertight) syntax inspired by SlicStan [13]. The `@yaps.model` decorator indicates that the following function, while being syntactically Python, should be semantically reinterpreted as Stan when the Python interpreter first encounters the function definition. Since the code is reinterpreted, its original Python interpretation is no longer available, and thus, does not leak abstractions. Line 2 declares `coin` as a probabilistic model with one observed variable `x`, using Python type syntax to encode the corresponding Stan type. Yaps attaches the name `x`, as well as other variable names in the model, to its intermediate representation. Line 3 declares the latent variable `theta` and establishes its prior. The loop in Lines 4–5 samples the observed coin tosses from a Bernoulli distribution.

Cell `[3]` Line 1 contains concrete observed coin flips. Instead of attaching a string, Line 2 associates the actual `flips` with the formal argument `x` of the `coin` model. This Python code invokes the Yaps compiler to translates the model to Stan code and the Stan compiler to compile it to C++ code, before training it on the training data. Yaps adds an attribute



```
In [1]: import yaps                                                          1
        from yaps.lib import *                                               2
        import numpy as np                                                   3
        from matplotlib import pyplot as plt                                 4
        import logging                                                       5
        logging.getLogger('pycmdstan.model').setLevel(logging.CRITICAL)      6

In [2]: @yaps.model                                                          1
        def coin(x: int(lower=0, upper=1)[10]):                              2
            theta: real(lower=0, upper=1) <~ uniform(0, 1)                   3
            for i in range(1,11):                                            4
                x[i] <~ bernoulli(theta)                                     5

In [3]: flips = np.array([0, 1, 0, 0, 0, 0, 0, 0, 0, 1])                     1
        coin_obs = coin(x=flips)                                             2
        coin_obs.sample(data=coin_obs.data, random_='seed=42')               3

In [4]: theta = coin_obs.posterior.theta                                     1
        plt.hist(theta, bins=20)                                             2
        print('mean of theta: {:.3f}'.format(theta.mean()))                  3

        mean of theta: 0.254
```

```
In [5]: coin.graph

Out[5]:
```

```
In [6]: print(coin)

        data {
          int<lower=0,upper=1> x[10];
        }
        parameters {
          real<lower=0,upper=1> theta;
        }
        model {
          theta ~ uniform(0,1);
          for (i in 1:11 - 1)
            x[i] ~ bernoulli(theta);
        }
```

Figure 8. Example Yaps model (cf. Figure 7).

posterior with fields for latent variables such as theta. Therefore, Cell [4] can refer to theta by name instead of using a string. The inferred posterior belief is the same (modulo minor numerical differences) as in classic PyCmdStan.

To reinterpret Python syntax, we must first parse it, and once parsed, we can easily visualize its dependencies. Cell [6] visually renders the graphical model. A user study has demonstrated that this kind of graphical representation is helpful while writing probabilistic programs [14]. Of course, the nodes in our visualization have variable names attached. Cell [5] shows the Stan code generated by Yaps.

4.2 Yaps Design

The design of Yaps follows the principle "Stan-like for probabilistic features, Python-like for everything else". It uses familiar Python syntax for non-probabilistic features such as for loops, type declarations, or function declarations. Stan-specific features are expressed with syntactically-valid Python that resembles the original syntax: <~ for sampling and *D*.T[a,b] for truncated distribution.

Since observed variables are free during modeling but bound during inference, Yaps make them formal arguments of the model (Figure 8 Cell [2] Line 2) and actual arguments of the inference call (Cell [3] Line 2).

Writing our own embedded Python parser would have been cumbersome, but fortunately, the Python standard library modules inspect and ast solved that for us. Our decorator @yaps.model uses those modules to replace the Python function by an intermediate representation suitable for visualization, compilation to Stan, and inference. One difficulty was to implement a suitable syntax for sampling. We first tried =~, based on Python's assignment, but that was insufficient, because the left-hand side of Stan's sampling is more expressive than that of Python's assignment. Hence we settled on <~ instead, which we substitute with is before parsing, since that does not occur in Stan and has a low precedence in Python.

Another difficulty was name resolution. Yaps identifiers refer to Stan types, functions, and distributions; for watertight abstractions, they should not be resolved to Python entities. The Python interpreter does not attempt to resolve names in Yaps function bodies, but it does in function signatures, such as def model2(N: int, y: real[N]). To prevent spurious error messages, Yaps provides stubs (e.g., int, real) and lets users to declare other names (e.g. N = yaps.dependent_type_var()).

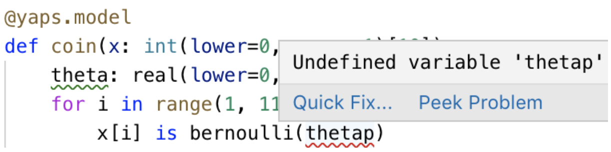
Finally, building a robust interface between Python and the Stan compiler would also have been cumbersome, but fortunately, PyCmdStan solved that for us. The only difficulty was that PyCmdStan error messages refer to locations in generated Stan code. We implemented a reverse source location mapping and used that to make error messages refer to source locations in Yaps code instead.

4.3 Yaps Evaluation

We evaluated Yaps using two kinds of tests. First, we tested our compiler on many programs; and second, we tested whether Yaps yields good error messages.

To obtain many realistic programs to test our compiler on, we built a second compiler that goes in the opposite direction. In other words, for testing purposes, we implemented a helper compiler from Stan to Yaps, so we can then use our main compiler from Yaps to Stan to complete the round-trip. This enabled the following experiments:

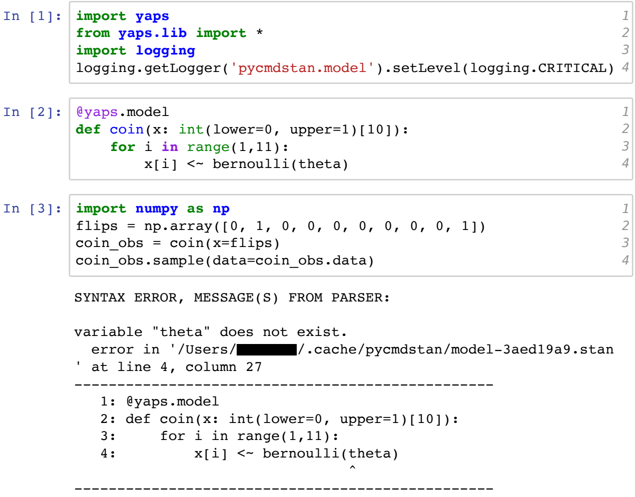
- We extracted 61 examples from the official Stan manual. The round-trip test succeeded for all of them (100%).
- We tried the round-trip test on 721 programs from the Stan dev repository. It succeeded for 700 of them (97%). The failed tests used deprecated syntax.



```
@yaps.model
def coin(x: int(lower=0,
    theta: real(lower=0,
    for i in range(1, 11
        x[i] is bernoulli(thetap)
```

Undefined variable 'thetap'

Figure 9. Example Yaps model with error message from common Python tooling.



```
In [1]: import yaps
        from yaps.lib import *
        import logging
        logging.getLogger('pycmdstan.model').setLevel(logging.CRITICAL)

In [2]: @yaps.model
        def coin(x: int(lower=0, upper=1)[10]):
            for i in range(1,11):
                x[i] <~ bernoulli(theta)

In [3]: import numpy as np
        flips = np.array([0, 1, 0, 0, 0, 0, 0, 0, 0, 1])
        coin_obs = coin(x=flips)
        coin_obs.sample(data=coin_obs.data)

        SYNTAX ERROR, MESSAGE(S) FROM PARSER:

        variable "theta" does not exist.
          error in '/Users/[redacted]/.cache/pycmdstan/model-3aed19a9.stan
        ' at line 4, column 27
        -------------------------------------------------
           1: @yaps.model
           2: def coin(x: int(lower=0, upper=1)[10]):
           3:     for i in range(1,11):
           4:         x[i] <~ bernoulli(theta)
                                          ^
        -------------------------------------------------
```

Figure 10. Example Yaps model with error message from Yaps compiler.

- We tried the round-trip test on 500 programs from the Stan examples repository. It succeeded for 411 of them (82%). Again, the failed tests used deprecated syntax.

To test whether the output of Pystan.na code matches the output of equivalent Stan code, we performed runtime tests as follows: we picked 13 Stan models with corresponding datasets and used our round-trip setup to generate Stan code through Yaps. We then compared the output of the inference run on the original Stan model code and the generated Stan model code. It matched for all 13 models.

To test whether Yaps improves error messages, we conducted two experiments. First, we intentionally misspelled a variable name and looked at the buggy program with popular Python tools. Figure 9 shows that the VSCode editor puts a red squiggly line under variable `thetap` and PyLint explains that it is undefined. Similarly, a green squiggly line warns that variable `theta` is unused. If the code had used a string instead of a variable, VSCode and PyLint would not have been able to detect and explain these mistakes.

For the second error-message experiment, we again injected an undefined variable, then used Yaps and Stan to compile the code, see Figure 10. Stan finds the error in the compiled code, and Yaps intercepts the error message from the Stan compiler, mapping it back to the Python code. As usual in notebooks, the error message appears in cell output.

While this section focused on Stan, reinterpreted Python is a general technique. It could also be used to define other EDSLs embedded in Python that have watertight abstractions with no strings attached.

5 Related Work

As Section 2 illustrates, it is common for ML frameworks in Python to construct a computational graph via lazy evaluation. This means those frameworks have the flavor of embedded domain specific languages (EDSLs) [15]. In essence, these EDSLs support a form of *staging*: in a first stage, the host-language code builds a computational graph, and in a second stage, the guest-language interpreter evaluates that graph, for instance, to train an ML model. In contrast to most popular Python EDSLs that have strings attached, this paper advocates using name reflection to have no strings attached.

Of course, embedding a language that constructs a computational graph is possible with other host languages besides Python. For instance, lightweight modular staging is a disciplined way for accomplishing the same in Scala [19]. Some programming languages take the idea to its extreme; for instance, the Racket language from the Lisp/Scheme family can implement languages as libraries [22]. In contrast, this paper focuses on Python and on the issue of attached names.

Like our work, several other recent efforts also recognize the potential to reinterpret Python code by parsing it. Both Tangent and Myia build a computational graph from which they derive derivatives that are crucial for gradient-descent based ML [6, 24]. And Relay builds a computational graph both for automatic differentiation and for mapping to heterogeneous hardware [18]. In contrast to these papers, our paper identifies strings attached as a common problem, and demonstrates solutions to this problem for Keras and PyStan.

Compared to PyStan [8] or PyCmdStan [16], which read Stan code from a file or from multi-line Python strings, Yaps is more deeply embedded. Compared to PyMC3 [21], Edward [23], or Pyro [3], Yaps is more watertight. While all of these efforts work in pure Python without any separate preprocessor, unlike Yaps, they do not track local variable names, do not cleanly isolate the embedded language, and their verbose syntax differs from stand-alone PPLs.

6 Conclusion

Python-based ML frameworks are very successful, but most frameworks popular today share a common flaw. They depend heavily on strings for naming things in the source code. This paper advocates using name reflection instead to keep variable names attached. We demonstrate that name reflection can be retrofitted on an existing ML framework with modest effort. We also show an extensive case study of designing a new ML framework that embeds Stan in Python with no strings attached. Our vision is that in the future, Python-based ML frameworks will be free of strings, making it easier to write, read, and maintain code in them.

References

[1] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *International Conference on Management of Data (SIGMOD)*. 1383–1394. https://doi.org/10.1145/2723372.2742797

[2] Guillaume Baudart, Martin Hirzel, Kiran Kate, Louis Mandel, and Avraham Shinnar. [n. d.]. Yaps: Python Frontend to Stan. https://arxiv.org/abs/1812.04125

[3] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. [n. d.]. Pyro: Deep Universal Probabilistic Programming. https://arxiv.org/abs/1810.09538

[4] David M. Blei, Alp Kucukelbir, and Jon D. McAuliffe. 2017. Variational Inference: A Review for Statisticians. *J. Amer. Statist. Assoc.* 112 (2017), 859–877. Issue 518. https://arxiv.org/abs/1601.00670

[5] Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. 2015. Weight Uncertainty in Neural Network. In *International Conference on Machine Learning (ICML)*. 1613–1622. http://proceedings.mlr.press/v37/blundell15.html

[6] Olivier Breuleux and Bart van Merriënboer. 2017. Automatic Differentiation in Myia. In *Autodiff Workshop*. https://openreview.net/forum?id=S1hcluzAb

[7] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. 2013. API Design for Machine Learning Software: Experiences from the scikit-learn Project. https://arxiv.org/abs/1309.0238

[8] Bob Carpenter, Andrew Gelman, Matt Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Michael A. Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A probabilistic programming language. *Journal of Statistical Software* 76, 1 (2017), 1–37. https://www.jstatsoft.org/article/view/v076i01

[9] Fran cois Chollet. 2015. Keras: The Python Deep Learning library. https://keras.io/ (Retrieved September 2018).

[10] Pedro Domingos. 2012. A Few Useful Things to Know About Machine Learning. *Communications of the ACM (CACM)* 55, 10 (Oct. 2012), 78–87. https://doi.org/10.1145/2347736.2347755

[11] Zoubin Ghahramani. 2015. Probabilistic machine learning and artificial intelligence. *Nature* 521, 7553 (May 2015), 452–459. https://www.nature.com/articles/nature14541

[12] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. 2014. Probabilistic Programming. In *ICSE track on Future of Software Engineering (FOSE)*. 167–181. https://doi.org/10.1145/2593882.2593900

[13] Maria I. Gorinova, Andrew D. Gordon, and Charles Sutton. 2018. SlicStan: Improving Probabilistic Programming using Information Flow Analysis. In *Workshop on Probabilistic Programming Languages, Semantics, and Systems (PPS)*. https://pps2018.soic.indiana.edu/files/2017/12/SlicStanPPS.pdf

[14] Maria I. Gorinova, Advait Sarkar, Alan F. Blackwell, and Don Syme. 2016. A Live, Multiple-Representation Probabilistic Programming Environment for Novices. In *Conference on Human Factors in Computing Systems (CHI)*. 2533–2537. https://doi.org/10.1145/2858036.2858221

[15] Paul Hudak. 1998. Modular domain specific languages and tools. In *International Conference on Software Reuse (ICSR)*. 134–142. https://doi.org/10.1109/ICSR.1998.685738

[16] marmaduke. 2017. PyCmdStan. https://gitlab.thevirtualbrain.org/tvb/pycmdstan (Retrieved September 2018).

[17] Fernando Pérez. 2014. Project Jupyter. http://jupyter.org/ (Retrieved September 2018).

[18] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. 2018. Relay: A New IR for Machine Learning Frameworks. In *Workshop on Machine Learning and Programming Languages (MAPL)*. 58–68. http://doi.acm.org/10.1145/3211346.3211348

[19] Tiark Rompf and Martin Odersky. 2012. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. *Communications of the ACM (CACM)* 55 (2012), 121–130. Issue 6. https://doi.org/10.1145/2184319.2184345

[20] Alexander M. Rush. 2017. Pyro Issue #502: Alternative Design of Variable Scoping and Naming. https://github.com/uber/pyro/issues/502 (Retrieved September 2018).

[21] John Salvatier, Thomas V. Wiecki, and Christopher Fonnesbeck. 2015. Probabilistic Programming in Python Using PyMC3. https://arxiv.org/abs/1507.08050

[22] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. Languages as Libraries. In *Conference on Programming Language Design and Implementation (PLDI)*. 132–141. https://doi.org/10.1145/1993498.1993514

[23] Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. 2017. Deep Probabilistic Programming. In *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/1701.03757

[24] Alex Wiltschko, Bart van Merriënboer, and Dan Moldovan. 2017. Tangent: Source-to-Source Debuggable Derivatives in Pure Python. https://github.com/google/tangent (Retrieved September 2018).