WOLFE: Strength Reduction and Approximate Programming for Probabilistic Programming

Sebastian Riedel
University College
London, UK
s.riedel@cs.ucl.ac.uk

Sameer Singh
University of Washington,
Seattle, USA
sameer@cs.washington.edu

Vivek Srikumar
Stanford University,
Stanford, USA
svivek@cs.stanford.edu

Tim Rocktäschel
University College
London, UK
t.rocktaschel@cs.ucl.ac.uk

Larysa Visengeriyeva
Technische Universität
Berlin, Germany
visenger@gmail.com

Jan Noessner
University of Mannheim,
Germany
jan@informatik.uni-mannheim.de

Abstract

Existing modeling languages lack the expressiveness or efficiency to support many modern and successful machine learning (ML) models such as structured prediction or matrix factorization. We present WOLFE, a probabilistic programming language that enables practitioners to develop such models. Most ML approaches can be formulated in terms of scalar objectives or scoring functions (such as distributions) and a small set of mathematical operations such as maximization and summation. In WOLFE, the user works within a functional host language to declare scalar functions and invoke mathematical operators. The WOLFE compiler then replaces the operators with equivalent, but more efficient (strength reduction) and/or approximate (approximate programming) versions to generate low-level inference or learning code. This approach can yield very concise programs, high expressiveness and efficient execution.

Introduction

Existing probabilistic programming languages face a trade-off between the expressivity of the models they represent and complexity of the modeling process for the user. Towards one end of the complexity spectrum, we have languages that are limited (by design and often intentionally) in the types of machine learning models and approaches they support. For example, Church (Goodman et al. 2008) is a powerful tool for generative models, but discriminatively trained structured prediction models are difficult to realize. Languages such as Markov Logic Networks (Richardson and Domingos 2006) can train discriminative models, but fail to support paradigms such as matrix or tensor factorization. In contrast, libraries such as FACTORIE (McCallum, Schultz, and Singh 2009) can be used to create arbitrarily rich models, but the burden is often on the user to provide efficient algorithms and data structures to operate such models, requiring machine learning expertise. There is a need for an extensible probabilistic programming language that can express a variety of models and algorithms while still retaining the simplicity and conciseness of existing paradigm-specific, declarative languages.

In this paper, we introduce WOLFE,[1] a functional probabilistic programming language that allows users to define rich models in a concise way (akin to mathematical expressions seen in machine learning papers) which are then complied into efficient implementations, thus combining ease of use with expressiveness and efficient computation. WOLFE is based on a formulation that is common to many machine learning algorithms and models: real-valued/scalar functions (for example corresponding to density functions, energy functions, discriminative scores, and training objectives) and mathematical operators that operate upon them (most prominently maximization, summation, and integration). The user constructs the relevant scalar functions in a functional host programming language,[2] and uses mathematical operators such as `argmax` to operate upon them, thus enabling a rich set of concisely defined ML models. The semantics of a WOLFE program is defined by the mathematical operators, and further, each WOLFE program is an executable program in the host language (albeit with brute force default implementations of the operators that are often intractable).

To ensure efficiency, the WOLFE compiler analyzes the user provided scalar function definitions and generates optimized source code in the host language. WOLFE performs two types of optimizations: *strength reduction* to replace brute-force operator implementations (such as exhaustive search) with equivalent but more efficient versions (such as Max-Product in a junction tree); and *approximate programming* where operator calls are replaced by efficient approximations (for example parallelized stochastic optimization). This separation of program semantics and implementation provides a number of crucial benefits: (1) The user's focus is on the mathematical formulation using the host lan-

[1] http://www.wolfe.ml

[2] Currently, we use Scala (Odersky, Spoon, and Venners 2008).

guage data structures, without distractions from efficiency or implementation details. (2) It allows inspection of the program structure that can lead to significant optimizations, (3) It enables reuse of existing inference and learning implementations from other probabilistic language implementations, for example we use the FACTORIE optimization package (Passos, Vilnis, and McCallum 2013), (4) By generating source code, WOLFE utilizes further specific optimizations from the host language compiler, and last, (5) The WOLFE interface, consisting of a few mathematical operators, is likely to remain fixed, and thus development will focus on further compiler optimizations, resulting in full backwards-compatibility of user programs with future WOLFE versions.

Wolfe

WOLFE models consist of a definition of the sample space and a scalar function that assigns probabilities (or, more generally, unnormalized scores) to each sample. A WOLFE program additionally consists of a set of WOLFE operators that are applied to these two components of the WOLFE model.

Sample Space

A *sample space* in WOLFE is a collection of objects of some type `T`, which represent the set of possible worlds for the model. In Scala we use objects of type `Iterable[T]` to represent this collection. WOLFE provides helper methods to compose rich sample spaces over primitive objects and algebraic data types (ADTs).[3] For example, for the `Sentence` type in listing 1 the function `space` defines the set of all `Sentence` objects as the cross product of all possible word and tag sequences. Notice that this sample space may have word and tag sequences of different lengths. However, using a `filter` on the collection we can constrain sentences to be consistent.

Scalar Function

The second part of a WOLFE model is a scalar function that can be used to define probabilities, densities, loss functions, etc. over the sample space. In Scala, such functions are defined by `T => Double`, allowing use of existing or user-written functions. In listings 1 and 3 the functions denoted by `s` are (parametrized) examples.

Operators

WOLFE provides several operators that take the sample space and the scalar functions as curried arguments, i.e. `(space:Iterable[T])(obj:T=>Dist)`. The `query` operator, similar to that in Church, generates a sample from `space` using the (unnormalized) distribution as defined by `obj`. The `argmax` operator returns an argument in `space` that maximizes `obj`. The `sum` operator applies `obj` to every element in `space` and sums up the results. As we will see in the examples, these operators can be used to represent a variety of inference and learning objectives.

[3] In Scala ADTs are defined using case classes and sealed traits; they can represent complex structures such as maps and lists

Transformations

Each operator comes with a default brute-force implementation in Scala, for example the `argmax` performs a linear search over the `space` to find the object that maximizes `obj`. However, for most models of interest these default implementations will not be tractable. Instead, WOLFE replaces the implementation of these operators with tailor-made algorithms that leverage structure (by examining the definitions of the sample space and the model). We currently perform two types of transformations (implemented using Scala macros).

Factor Graph: For operators and sample spaces that represent inference, WOLFE represents the operator as inference on a factor graph (Kschischang, Frey, and Loeliger 2001). WOLFE first analyzes the Scala Abstract Syntax Tree (AST) that defines the sample space (`space`) to discover the random variables of the graph. It finds a compositional isomorphism between the sample space and a factor graph in which each value in the space corresponds to one assignment of the variables in the factor graph, and vice versa. Next the AST of the `obj` function is divided into *factors* according occurrence of either `sum` or `prod` operators (or quantified versions of these). For each factor we find the nodes that affect the value of the factor, using the aforementioned isomorphism. Given this factor graph that represents the `obj` over the `space`, we can run message passing or sampling algorithms that exploit the graph's structure, providing an efficient (but often approximate) implementation of the original operator.

Gradient Optimizer: Many important scalar functions in machine learning are defined over continuous vector spaces, for example learning the parameters of a model. In WOLFE we provide support for efficient `argmax` execution over such functions, provided that we can analytically derive their gradients. For `argmax` calls that involve continuous `space`, WOLFE takes the AST of the `obj` function and applies derivation rules to find a gradient or sub-gradient. To perform the `argmax` operation WOLFE then uses the efficient FACTORIE optimization library to perform gradient-based search such as SGD, LBFGS or Perceptron.

For both of the above optimizations, WOLFE automatically selects the algorithm to use, and chooses the various hyper-parameters, by performing cursory inspection of the objective and the sample space. However, to allow expert users to inform this selection, WOLFE provides a library of Scala *annotations* that can be used in the WOLFE program to select the algorithm, and to specify the various parameters associated with it, for each call to the operators.

Examples

We give a few examples of WOLFE models and sketch how strength reduction and approximate programming can be lead efficient learning and inference. Note that the code snippets here are primarily for illustration of how a variety

```
case class Sentence(words: Seq[String], tags: Seq[String])
//sample space
def space = all(Sentence){seqs(strings) x seqs(tagSet)}
// feature vector for a sentence
def f(s:Sentence):Vector = {
  val n = s.words.size
  sum(0 until n) { i=> oneHot(s.words(i)->s.tags(i))} +
  sum(0 until n-1) { i => oneHot(s.tags(i)->s.tags(i+1))}
}
// model to score a sentence
def s(w:Vector)(s:Sentence):Double = w dot f(s)
```

Listing 1: **Linear-chain CRF:** `oneHot(key)` is a vector where the component at index `key` is 1 and 0 elsewhere.

```
// MAP inference
def h(w:Vector)(x:Sentence):Sentence =
  argmax(space)(obs(_)==obs(x)){ d=>s_w(d) }
// Loss over training data
def loss(data:Seq[Sentence])(w:Vector):Double =
  sum(data){ d=>s_w(h_w(d))-s_w(d) }
// Parameter estimation
def learn(data:Seq[Sentence]) =
  argmin(vectors){ w => loss_data(w) }
```

Listing 2: **Inference and Learning** for the model in Listing 1. `obs(x)` is the sequence of observations (words) in `x`.

of models may be expressed in WOLFE; the details of the syntax are not crucial for understanding, and may, in fact, be subject to change.

Linear Chains Listing 1 shows the definition of a linear chain Part-of-Speech tagger in WOLFE, together with inference and learning code necessary to use the model in Listing 2. The code is equivalent to the following formulation. We first define a feature function over a sentence using one-hot vectors $\mathbf{e}_i$ (vector where component at index i is 1, 0 elsewhere) with a rich index set:

$$\mathbf{f}(d) = \sum_i^n \mathbf{e}_{d_{word_i}, d_{tag_i}} + \sum_i^{n-1} \mathbf{e}_{d_{tag_i}, d_{tag_{i+1}}}$$

With a weight vector $\mathbf{w}$ this yields a linear model:

$$s_{\mathbf{w}}(d) = \langle \mathbf{w}, \mathbf{f}(d) \rangle .$$

In listing 2 maximum a-posteriori (MAP) inference and learning are defined in terms of `argmax` and `argmin` operators. For example, the MAP predictor `h` corresponds to

$$h_{\mathbf{w}}(x) = \arg\max_{d:d_x=x} s_{\mathbf{w}}(d)$$

i.e. it finds the highest scoring structure d that agrees with the observed part of the evidence x we condition on. Before execution of the program, WOLFE transforms the linear model into a factor graph isomorphic to the `Sentence` data structure, and to the factorization of the model `s`, during compilation time. Then WOLFE replaces each call to `h` with message passing on this factor graph that computes the exact solution, thus demonstrating strength reduction.

```
case class UserItem(user: User, item: Item, rating: Double)
// model
def s(w:Vector)(u:UserItem) =
  sum(0 until k){ i => w(u.item->i)*w(u.user->i) } +
  sum(u.user.items){ i => w(i->u.item) }
// training loss over observed cells
def loss(data:Seq[UserItem])(w:Vector) =
  sum(data){ d => pow2(d.rating - s_w(d))}
```

Listing 3: **Matrix Factorization:** A collaborative filtering model including a neighborhood term.

The function `loss` in Listing 2 defines a perceptron style training loss. The function `learn` finds a minimizer of this loss. In this case WOLFE generates code that calculates the gradient of the loss and then performs gradient descent on it, demonstrating approximate programming. To calculate the gradient at any point of optimization, WOLFE performs MAP inference as described above.

Matrix Factorization Listing 3 shows a recommender system that combines a matrix factorization model (first line of `s`) with a neighborhood model (Koren 2008) that captures item-item correlations directly. WOLFE replaces calls to `argmin(loss(data))` with code that calculates the gradient of the loss and runs stochastic gradient descent.

Generative Models: Latent Dirichlet Analysis Listing 4 illustrates a Wolfe program for a topic model using Latent Dirichlet Analysis (Blei, Ng, and Jordan 2003).[4] Since the underlying model is generative, the model (as defined by `lda`) is easier to write as the probability of the assignment to all the variables, instead of specifying the unnormalized energy functions as in the examples so far. The additional functions used in this example are `dir` for the Dirichlet density, `cat` for the categorical distribution, and `prod` is the product over the arguments (as opposed to the `sum`). WOLFE replaces inference calls code that integrates out the unobserved variables (φ and θ) with an efficient Gibbs sampling implementation.

Related Work

WOLFE shares a number of aspects of different probabilistic programming languages. Church (Goodman et al. 2008), its recently introduced extensions Venture (Mansinghka, Selsam, and Perov 2014) and Fun (Borgström et al. 2011) are similar in that they make the separation between user program and efficient implementations via program (and execution) analysis; however they are limited to generative models defined through generative stories, not scalar functions. Infer.NET (Minka et al. 2010) combines program analysis and host compiler optimizations for efficient sampling and variational inference, but is restricted to generative models that are specified using custom data structures. Work on MLNs (Richardson and Domingos 2006) and ProbLog (Raedt, Kimmig, and Toivonen 2007) provide

[4] Note that generative models are not supported in the current version, but will be included in the future versions of WOLFE.

```
case class Token(topic:Int,word:String)
case class Doc(theta:Map[Int,Double],tokens:Seq[Token])
case class World(phi:Seq[Map[String,Double]],docs:Seq[Doc])

// model computes the probability of the input
def lda(w: Vector)(world:World):Double = {
  import world._
  prod(phi) { dir(_,w('beta))} *
  prod(docs) { d =>
    dir(d.theta, w('alpha)) *
    prod(d.tokens) {t =>
      cat(t.topic, d.theta) *
      cat(t.word, phi(t.topic))
    }
  }
}
```

Listing 4: **Topic Model:** The model here computes the probability given values for all the variables. Inference (not shown) computes distribution over topic assignments (`t.topic`) given the documents and the hyperparameters `w` by summing out `phi` and `theta`.

a useful logic-based declarative language that unfortunately cannot concisely express many of the models used in practice. Learning Based Java (Rizzolo and Roth 2010) provides a declarative language for specifying discriminative models, but restricts the expressiveness of the models that can be defined. Although FACTORIE (McCallum, Schultz, and Singh 2009) is able to express a wide variety of models, it often requires the user to additionally define data structures and implementations for efficient inference. WOLFE, on the other hand, allows the user to specify arbitrarily complex models (concisely by using host language syntax) while still providing efficient implementation via strength reduction and approximate programming.

Conclusions

We have presented WOLFE, a rich language for declarative ML modeling. WOLFE uses two ingredients: a functional host language to define real-valued functions in, and a compiler that transforms mathematical operations on these functions into efficient inference and learning code by strength reduction and approximate programming. WOLFE thus uses mathematics as the unifying machine learning language, and aims to make code look similar to the "model" section of papers that use machine learning. This approach enables users to define expressive models declaratively and shields them from implementation details.

Acknowledgments

This work was supported in part by Microsoft Research through its PhD Scholarship Programme, and in part by the TerraSwarm Research Center, one of six centers supported by the STARnet phase of the Focus Center Research Program (FCRP) a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

Blei, D. M.; Ng, A. Y.; and Jordan, M. I. 2003. Latent dirichlet allocation. *the Journal of machine Learning research* 3:993–1022.

Borgström, J.; Gordon, A. D.; Greenberg, M.; Margetson, J.; and Van Gael, J. 2011. Measure transformer semantics for bayesian machine learning. In *Programming Languages and Systems*. Springer. 77–96.

Goodman, N. D.; Mansinghka, V. K.; Roy, D.; Bonawitz, K.; and Tenenbaum, J. B. 2008. Church: a language for generative models. In *Uncertainty in Artificial Intelligence*.

Koren, Y. 2008. Factorization meets the neighborhood: a multifaceted collaborative filtering model. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '08, 426–434. New York, NY, USA: ACM.

Kschischang, F. R.; Frey, B. J.; and Loeliger, H. A. 2001. Factor graphs and the sum-product algorithm. *IEEE Transactions of Information Theory* 47(2):498–519.

Mansinghka, V.; Selsam, D.; and Perov, Y. 2014. Venture: a higher-order probabilistic programming platform with programmable inference. *ArXiv e-prints*.

McCallum, A.; Schultz, K.; and Singh, S. 2009. Factorie: Probabilistic programming via imperatively defined factor graphs. In Bengio, Y.; Schuurmans, D.; Lafferty, J.; Williams, C. K. I.; and Culotta, A., eds., *Advances in Neural Information Processing Systems 22*. 1249–1257.

Minka, T.; Winn, J. M.; Guiver, J. P.; and Knowles, D. A. 2010. Infer.NET 2.4. Microsoft Research Cambridge. http://research.microsoft.com/infernet.

Odersky, M.; Spoon, L.; and Venners, B. 2008. *Programming in Scala*. artima.

Passos, A.; Vilnis, L.; and McCallum, A. 2013. Optimization and learning in factorie. In *NIPS Workshop on Optimization for Machine Learning (OPT)*.

Raedt, L. D.; Kimmig, A.; and Toivonen, H. 2007. Problog: A probabilistic prolog and its application in link discovery. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2462–2467.

Richardson, M., and Domingos, P. 2006. Markov logic networks. *Machine Learning* 62:107–136.

Rizzolo, N., and Roth, D. 2010. Learning based java for rapid development of nlp systems. In *Language Resources and Evaluation Conference (LREC)*.