Inference Compilation and Universal Probabilistic Programming

Tuan Anh Le  Atılım Güneş Baydin  Frank Wood
Department of Engineering Science, University of Oxford
{tuananh, gunes, fwood}@robots.ox.ac.uk

Abstract

We introduce a method for using deep neural networks to amortize the cost of inference in models from the family induced by universal probabilistic programming languages, establishing a framework that combines the strengths of probabilistic programming and deep learning methods. We call what we do “compilation of inference” because our method transforms a denotational specification of an inference problem in the form of a probabilistic program written in a universal programming language into a trained neural network denoted in a neural network specification language. When at test time this neural network is fed observational data and executed, it performs approximate inference in the original model specified by the probabilistic program. Our training objective and learning procedure are designed to allow the trained neural network to be used as a proposal distribution in a sequential importance sampling inference engine. We illustrate our method on mixture models and Captcha solving and show significant speedups in the efficiency of inference.

1 INTRODUCTION

Probabilistic programming uses computer programs to represent probabilistic models (Gordon et al., 2014). Probabilistic programming systems such as STAN (Carpenter et al., 2015), BUGS (Lunn et al., 2000), and Infer.NET (Minka et al., 2014) allow efficient inference in a restricted space of generative models, while systems such as Church (Goodman et al., 2012), Venture (Mansinghka et al., 2014), and Anglican (Wood et al., 2014)—which we call universal—allow inference in unrestricted models. Universal probabilistic programming systems are built upon Turing complete programming languages which support constructs such as higher order functions, stochastic recursion, and control flow.

There has been a spate of recent work addressing the production of artifacts via “compiling away” or “amortizing” inference (in the sense of Gershman and Goodman (2014)). This body of work is roughly organized into two camps. The one in which this work lives, arguably the camp organized around “wake-sleep” (Hinton et al., 1995), is about offline unsupervised learning of observation-parameterized importance-sampling distributions for Monte Carlo inference algorithms. In this camp, the approach of Paige and Wood (2016) is closest to ours in spirit; they propose learning autoregressive neural density estimation networks offline that approximate inverse factorizations of graphical models so that at test time, the trained “inference network” starts with the values of all observed quantities and progressively proposes parameters for latent nodes in the original structured model. However, inversion of the dependency structure is impossible in the universal probabilistic program model family, so our approach
hews instead towards learning proposals for “forward” inference methods in which no model dependency inversion is performed. In this sense, our work can be seen as being inspired by that of Kulkarni et al. (2015) and Ritchie et al. (2016b) where program-specific neural proposal networks are trained to guide forward inference. Our aim, though, is to be significantly less model-specific. At a high level what characterizes this camp is the fact that the artifacts are trained to suggest sensible yet varied parameters for a given, explicitly structured and therefore potentially interpretable model.

The other related camp, emerging around the variational autoencoder (Kingma and Welling, 2013; Burda et al., 2015), also amortizes inference in the manner we describe, but additionally also simultaneously learns the generative model, within the structural regularization framework of a parameterized non-linear transformation of the latent variables. Approaches in this camp generally produce recognition networks that non-linearly transform observational data at test time into parameters of a variational posterior approximation, albeit one with less conditional structure, excepting the recent work of Johnson et al. (2016). A chief advantage of this approach is that the learned model, as opposed to the recognition network, is simultaneously regularized both towards being simple to perform inference in and towards explaining the data well.

In this work, we concern ourselves with performing inference in generative models specified as probabilistic programs while recognizing that alternative methods exist for amortizing inference while simultaneously learning model structure. Our contributions are twofold: (1) We work out ways to handle the complexities introduced when compiling inference for the class of generative models induced by universal probabilistic programming languages and establish a technique to embed neural networks in forward probabilistic programming inference methods such as sequential importance sampling (Doucet and Johansen, 2009). (2) We develop an adaptive neural network architecture, comprising a recurrent neural network core and embedding and proposal layers specified by the probabilistic program, that is reconfigured on-the-fly for each execution trace and trained with an “infinite” stream of training data sampled from the generative model. This establishes a framework combining deep neural networks and generative modeling with universal probabilistic programs (Figure 1).

We begin by providing background information and reviewing related work in Section 2. In Section 3 we introduce our compiled sequential importance sampling approach, the objective function, and the neural network architecture. Section 4 demonstrates our approach on two examples, mixture models and Captcha solving, followed by the discussion in Section 5.

2 BACKGROUND

2.1 Probabilistic Programming

Probabilistic programs denote probabilistic generative models as programs that include sample and observe statements. Both sample and observe are functions that specify random variables in this generative model using probability distribution objects as an argument, while observe, in addition, specifies the conditioning of this random variable upon a particular observed value in a second argument. These observed values induce a conditional probability distribution over the execution traces whose approximations and expected values we want to characterize by performing inference.

An execution trace of a probabilistic program is obtained by successively executing the program deterministically, except when encountering sample statements at which point a value is generated according to the specified probability distribution and appended to the execution trace. We assume the order in which the observe statements are encountered is constant. Hence, we denote the observed values by $y := (y_n)_{n=1}^N$ for a fixed $N$ in all possible traces.

Depending on the probabilistic program and the values generated at sample statements, the order in which the execution encounters sample statements as well as the number of encountered sample statements may be different from one trace to another. Therefore, given a scheme which assigns a unique address to each sample statement according to its lexical position in the probabilistic program, we represent an execution trace of a probabilistic program as a sequence of length $T$ which is itself a program-execution dependent variable

$$ (x_t, a_t, i_t)_{t=1}^T, \quad (1) $$

where $x_t$, $a_t$, and $i_t$ are respectively the sample value, address, and instance (call number) of the $t$th entry in a given trace. Instance values $i_t = \sum_{j=1}^t 1(a_t = a_j)$ count the number of sample values obtained from the specific sample statement at address $a_t$ up to time step $t$. For each trace, a sequence $x := (x_j)_{j=1}^T$ holds the $T$ sampled values from the sample statements.

The joint probability density of an execution trace is

$$ p(x, y) := \prod_{t=1}^T f_{a_t} (x_t \mid x_{1:t-1}) \prod_{n=1}^N g_n(y_n \mid x_{1:\tau(n)}), \quad (2) $$

where $f_{a_t}$ is the probability distribution specified by the sample statement at address $a_t$ and $g_n$ is the probability distribution specified by the $n$th observe statement. $f_{a_t}(\cdot \mid x_{1:t-1})$ is called the prior conditional density given the sample values $x_{1:t-1}$ obtained before encountering the $t$th sample statement. $g_n(\cdot \mid x_{1:\tau(n)})$
is called the likelihood density given the sample values $x_{1:t}(n)$ obtained before encountering the $n$th `observe` statement, where $\tau$ is a mapping from the index $n$ of the `observe` statement to the index of the last `sample` statement encountered before this `observe` statement during the execution of the program.

Inference in such models amounts to computing an approximation of $p(x \mid y)$ and its expected values $\hat{I}_\zeta = \int \zeta(x)p(x \mid y)\,dx$ over chosen functions $\zeta$.

While there are many inference algorithms for universal probabilistic programming languages (Wingate et al., 2011; Ritchie et al., 2016a; Wood et al., 2014; Paige et al., 2014; Rainforth et al., 2016), we focus on algorithms in the importance sampling family in the context of which we will develop our scheme for amortized inference.

2.2 Sequential Importance Sampling

Sequential importance sampling (SIS) (Arlulampalam et al., 2002; Doucet and Johansen, 2009) is a method for performing inference over execution traces of a probabilistic program (Wood et al., 2014) whereby a weighted set of samples $\{(a^K, x^K)\}_{k=1}^K$ is used to approximate the posterior and the expectations of functions as

$$\hat{p}(x \mid y) = \frac{K}{\sum_{k=1}^K w^k} \delta(x^k - x) / \sum_{j=1}^K w^j$$  \hspace{1cm} (3)

$$\hat{I}_\zeta = \frac{\sum_{k=1}^K w^k \zeta(x^k)}{\sum_{j=1}^K w^j},$$  \hspace{1cm} (4)

where $\delta$ is the Dirac delta function.

SIS requires designing proposal distributions $q_{a,i}$ corresponding to the addresses $a$ of all `sample` statements in the probabilistic program and their instance values $i$. A proposal execution trace $x^k_{1:T_k}$ is built by executing the program as usual, except when a `sample` statement at address $a_t$ is encountered at time $t$, a proposal sample value $x^k_t$ is sampled from the proposal distribution $q_{a_t,i_t}(\cdot \mid x^k_{1:t-1})$ given the proposal sample values until that point. We obtain $K$ proposal execution traces $x^k := x^k_{1:T_k}$ (possibly in parallel) to which we assign weights

$$w^k = \prod_{n=1}^N g_n(y_n \mid x^k_{1:tn(n)}) \cdot \prod_{t=1}^{T_k} f_{a_t,i_t}(x^k_t \mid x^k_{1:t-1})$$  \hspace{1cm} (5)

for $k = 1, \ldots, K$ with $T_k$ denoting the length of the $k$th proposal execution trace.

3 APPROACH

We describe our approach to compiled inference, which we call compiled sequential importance sampling (CSIS), in Section 3.1. In CSIS, compilation amounts to minimizing a function, specifically the loss function of a neural network architecture, which makes the proposal distributions good in the sense that we specify in Section 3.2. The process of generating training data for this neural network architecture from the generative model is described in Section 3.3. At the end of training, we obtain a compilation artifact comprising the neural network components—the recurrent neural network core and the embedding and proposal layers corresponding to the original model denoted by the probabilistic program—and the set of trained weights, as described in Section 3.4.

3.1 Compiled Sequential Importance Sampling

As a first step towards compilation of inference in universal probabilistic programming systems, we present proposal distribution adaptation via approximating $p(x \mid y)$ in the framework of sequential importance sampling (SIS).

Assuming we have a set of adapted proposals $q_{a_t,i_t}(x_t \mid x_{1:t-1}, y)$ such that their joint $q(x \mid y) \approx p(x \mid y)$ the resulting CSIS algorithm remains unchanged from the one described in Section 2.2 except the replacement of $q_{a_t,i_t}(x_t \mid x_{1:t-1})$ by $q_{a_t,i_t}(x_t \mid x_{1:t-1}, y)$.
3.2 Objective Function

We can think of our neural network as a function \( \eta(\cdot) \) parameterized by \( \phi \), mapping to the parameters \( \eta_t \) of the individual adapted proposals \( q_{a_t,i_t}(x_t \mid x_{1:t-1}, y, \phi) := q_{a_t,i_t}(x_t \mid x_{1:t-1}, y) \) for \( t = 1, \ldots, T \).

We use the Kullback–Leibler divergence \( D_{KL}(p(x \mid y) \| q(x \mid y; \phi)) \) as our measure of closeness between \( p(x \mid y) \) and \( q(x \mid y; \phi) \). To achieve closeness over many possible \( y \)'s, we take the expectation of this quantity under the distribution of \( p(y) \) and ignore the terms excluding \( \phi \) in the second equality:

\[
E_{p(y)} [D_{KL}(p(x \mid y) \| q(x \mid y; \phi))] = \int p(y) \int p(x \mid y) \log \frac{p(x \mid y)}{q(x \mid y; \phi)} \, dx \, dy
\]

\[
= -E_{p(x,y)} [\log q(x \mid y; \phi)] + \text{const.} \tag{7}
\]

This objective function corresponds to the negative entropy criterion.

Considering the factorization

\[
q(x \mid y; \phi) = \prod_{t=1}^{T} q_{a_t,i_t}(x_t \mid \eta_t(x_{1:t-1}, y, \phi)), \tag{8}
\]

the neural network architecture must have the capacity to map to a variable number of outputs, and incorporate sampled values in a sequential manner, in conjunction with the running of the inference engine. We describe our neural network architecture in detail in Section 3.4.

3.3 Training Data

Since Eq. 7 is an expectation over the joint distribution, we can use its Monte Carlo estimate as the objective function to minimize:

\[
\mathcal{L}(\phi) = -\frac{1}{M} \sum_{m=1}^{M} \log q(x^{(m)} \mid y^{(m)}; \phi) \tag{9}
\]

\[
(x^{(m)}, y^{(m)}) \sim p(x, y), \quad m = 1, \ldots, M. \tag{10}
\]

Here, \((x^{(m)}, y^{(m)})\) is the \( m \)th training (probabilistic program execution) trace generated by running an unconstrained probabilistic program corresponding to the original one. This unconstrained probabilistic program is obtained by a program transformation which replaces each \texttt{observe} statement in the original program by \texttt{sample} and ignores its second argument.

Universal probabilistic programming languages support stochastic branching and can generate execution traces with a changing (and possibly unbounded) number of random choices. We must, therefore, keep track of information about the addresses and instances of the samples \( x_t^{(m)} \) in the execution trace, as introduced in Eq. 1. Specifically, we generate our training data in the form of minibatches (Cort et al., 2011) sampled from the generative model \( p(x, y) \):

\[
D_{\text{train}} = \left\{ (x_t^{(m)}, a_t^{(m)}, i_t^{(m)})_{t=1}^{T_{(m)}}, (y_n^{(m)})_{n=1}^{N} \right\}_{m=1}^{M}, \tag{11}
\]

where \( M \) is the minibatch size, and, for a given trace \( m \), the sample values, addresses, and instances are respectively denoted \( x_t^{(m)}, a_t^{(m)}, i_t^{(m)} \), and the values sampled from the distributions in \texttt{observe} statements are denoted \( y_n^{(m)} \).

During compilation, training minibatches are generated on-the-fly from the probabilistic generative model and streamed to a stochastic gradient descent (SGD) procedure for optimizing the neural network weights \( \phi \). In our experiments we use the adaptive moment estimation (Adam) method of Kingma and Ba (2014), which computes individual adaptive learning rates for the parameters of a stochastic objective function with only first-order gradients and favorable memory requirements.

Minibatches of this “infinite” stream of training data are discarded after each SGD update; we therefore have no notion of a finite training set and associated issues such as overfitting to a set of training data and early stopping using a validation set (Jehleit, 1998). We do sample a validation set to compute validation losses for tracking the progress of training in a less noisy way than that admitted by the training loss.

3.4 Neural Network Architecture

Our compilation artifact is a collection of neural network components and their trained weights, specialized in performing inference in the model specified by a given probabilistic program. The neural network architecture comprises a non-domain-specific recurrent neural network (RNN) core and domain-specific observation embedding and proposal layers specified by the given program. We denote the set of the combined parameters of all neural network components \( \phi \).

RNNs are a popular class of neural network architecture which are well-suited for sequence-to-sequence modeling (Sutskever et al., 2014) with a wide spectrum of state-of-the-art results in domains including machine translation (Bahdanau et al., 2014), video captioning (Venugopalan et al., 2014), and learning execution traces (Reed and de Freitas, 2015). We use RNNs in this work owing to their ability to encode dependencies over time in the hidden state. In particular, we use a long short-term memory (LSTM) architecture which helps mitigate the vanishing and exploding gradient problems of RNNs (Hochreiter and Schmidhuber,
In an LSTM, for a given input sequence \( \{ \rho_t : \rho_t \in \mathbb{R}^{d_{\text{in}}}, t = 1 : T \} \), an output sequence \( \{ h_t : h_t \in \mathbb{R}^{d_{\text{out}}}, t = 1 : T \} \) can be produced via the following forward propagation equations (for \( t = 1, \ldots, T \)):

\[
\begin{align*}
    i_t &= \sigma(\theta_{pi}\rho_t + \theta_{ih}h_{t-1} + b_i), \\
    f_t &= \sigma(\theta_{pf}\rho_t + \theta_{hf}h_{t-1} + b_f), \\
    o_t &= \sigma(\theta_{po}\rho_t + \theta_{ho}h_{t-1} + b_o), \\
    g_t &= \tanh(\theta_{pg}\rho_t + \theta_{hg}h_{t-1} + b_g), \\
    c_t &= f_t \odot c_{t-1} + i_t \odot g_t, \\
    h_t &= o_t \odot \tanh(c_t).
\end{align*}
\]

Here, \( i_t, f_t, o_t \in \mathbb{R}^{d_{\text{out}}} \) are the input, forget, and output gates; \( g_t \in \mathbb{R}^{d_{\text{out}}} \) is the gated signal; \( c_t, h_t \in \mathbb{R}^{d_{\text{out}}} \) are the cell and hidden states; \( \sigma \) is an elementwise logistic sigmoid function; \( \tanh \) is an elementwise hyperbolic tangent; \( \odot \) is elementwise multiplication; the \( \theta, b \) are trainable parameters of the LSTM which are matrices or vectors of appropriate dimensions.

The overall architecture is formed by combining the LSTM core with a domain-specific observe embedding layer \( f_{\text{obs}} \), and several sample embedding layers \( f_{\text{sam}} \) and proposal layers \( f_{\text{prop}} \) that are distinct for each address-instance pair \( (a, i) \) (Figure 3). As described in Section 3.3, each probabilistic program execution trace can be of different length and composed of a different sequence of addresses and instances. To handle this complexity, we define an adaptive neural network architecture that is reconfigured for each encountered trace by attaching the corresponding embedding and proposal layers to the LSTM core, creating new layers on-the-fly on the first encounter with each \( (a, i) \) pair.

Evaluation starts by computing the observe embedding \( f_{\text{obs}}(y) \). This embedding is computed once per trace and repeatedly supplied as an input to the LSTM at each time step. Another alternative is to supply this embedding only once in the first time step, an approach preferred by Karpathy and Fei-Fei (2015) and Vinyals et al. (2015) to prevent overfitting (also see Section 4.2).

At each time step \( t \), the input \( \rho_t \) of the LSTM is constructed as a concatenation of

1. the observe embedding \( f_{\text{obs}}(y) \),
2. the embedding of the previous sample \( f_{\text{sam}}(a_{t-1}, i_{t-1}) \), and
3. the one-hot encodings of the current address \( a_t \), instance \( i_t \), and proposal type \( \text{type}(a_t) \) of the sample statement

for which the artifact will generate the parameter \( \eta_t \) of the proposal distribution \( q_{a_t,i_t}(: \mid \eta_t) \). The parameter \( \eta_t \) is obtained via the proposal layer \( f_{\text{prop}}(h_t) \), mapping the output of the LSTM through the corresponding proposal layer. The LSTM network has the capacity to incorporate inputs in its hidden state. This allows the parametric proposal \( q_{a_t,i_t}(x_t \mid \eta_t(x_{1:t-1}, y, \phi)) \) to take in all previous samples and all of the observes.

During training (compilation), we supply the actual sample values \( x_{1:t-1} \) to the embedding \( f_{\text{sam}} \), and we are interested in the parameter \( \eta_t \) in order to calculate the per-sample loss \( \mathcal{L}_t(\phi) = -\log q_{a_t,i_t}(x_t \mid \eta_t(x_{1:t-1}, y, \phi)) \) and its derivative with respect to \( \phi \) to use in SGD updates.

During inference, the evaluation proceeds by requesting proposal parameters \( \eta_t \) from the artifact for specific address-instance instance pairs \( (a_t, i_t) \) as these are encountered, and the sample values \( x_{1:t-1} \) are generated by the CSIS procedure.

The neural network artifact is implemented in Torch (Collobert et al., 2011), and it uses the ZeroMQ\(^1\) framework for interfacing with our probabilistic programming system.\(^2\) This setup allows distributed training (e.g., Dean et al. (2012)) and inference with GPU support across many machines, which is beyond the scope of this paper.

\(^1\)http://zeromq.org/
\(^2\)Kept anonymous for this double-blind review.
4 EXPERIMENTS

We demonstrate our CSIS scheme on two examples. In our first example we demonstrate compiled inference for an open universe mixture model. In our second, we demonstrate Captcha solving via probabilistic inference (Mansinghka et al., 2013).

4.1 Mixture Models

Mixture modeling, e.g. the Gaussian mixture model (GMM) shown in Figure 5, is about density estimation, clustering, and counting. The inference problems posed by a GMM, given a set of vector observations, are to figure out how many clusters there are, where they are, how big they are, and, optionally, which data points came from which cluster.

We investigate compiled inference for a GMM in which the number of clusters is unknown. Inference arises from observing the values of $y_n$ (Figure 5, line 8) and inferring the posterior number of clusters $K$ and the set of cluster mean and covariance parameters $\{\mu_k, \Sigma_k\}_{k=1}^K$. We assume that the input data to this model has been translated to the origin and constrained to lie within $[-1, 1]$ in both directions.

In order for CSIS to make good proposals for such inference, the encoder artifact must be able to count, i.e., extract and represent information about how many clusters there are and, conditioned on that, to localize the clusters. Towards that end our encoder includes a deep convolutional neural network whose input is a $D \times D$ two-dimensional histogram image of binned observed data $y$.

In presenting observational data $y$ assumed to arise from a mixture model to the neural network, there are some important considerations that must be accounted for. In particular there are symmetries in mixture models (Nishihara et al., 2013) that must be broken in order for training and inference to work. First, there are $K!$ (factorial) ways to label the classes. Second, there are $N!$ ways the individual data points could be permuted. Even in experiments like ours with $K < 6$ and $N \approx 100$ this presents a major challenge for neural network training. We break the first symmetry by, at training time, sorting the clusters by Euclidian distance from the origin and relabeling all points with a permutation that labels points from the cluster nearest the original as coming from the first cluster, next closest the second, and so on. This is only approximately symmetry breaking as many different clusters may be very nearly the same distance away from the origin. Second, we avoid the $N!$ symmetry by only predicting the number, means, and covariances of the clusters, not the individual cluster assignments. The net effect of the sorting is that the proposal mechanism will learn to propose the nearest cluster to the origin as it receives training data always sorted in this manner.

Figure 4 shows that we are able to learn a proposal that makes inference dramatically more efficient. Figure 2 shows one kind of application such an efficient inference engine can do: object counting (Lempitsky and Zisserman, 2010) and localization.

4.2 Captcha Solving

Here we also demonstrate our CSIS framework by writing generative probabilistic models for Captchas (von Ahn et al., 2003) and comparing our results with the literature. Captcha solving is well suited for a generative probabilistic programming approach because its latent parameterization is low-dimensional and interpretable by design. Using conventional computer vision techniques, the problem has been previously approached using segment-and-classify pipelines (Starostenko et al., 2015; Bursztein et al., 2014; Gao et al., 2014, 2013), and state-of-the-art results have been obtained by using deep convolutional neural networks (CNNs) (Goodfellow et al., 2013; Stark et al., 2015), at the cost of requiring very large (in the order of millions) labeled training sets for supervised learning.

We start by writing generative models for each of the types surveyed by Bursztein et al. (2014), namely Baidu 2011 (Chinese), Baidu 2013 (English), eBay (45899), Yahoo (45899), reCaptcha (45899), and Wikipedia (1030). Figure 6 provides an overall summary of our modeling approach. The actual models include domain-specific letter dictionaries, font styles, and various types of renderer noise for matching each Captcha style. In particular, implementing the displacement fields introduced by Simard et al. (2003) proved instrumental in achieving our results. Note that the steps of stochastic renderer noise are not included in the example trace in Figure 6. Our experiments have shown that we can successfully train artifacts that also extract renderer noise parameters, but excluding these from the list of addresses for which we learn proposal distributions improves robustness when testing with data not sampled from the same model. This corresponds to the well-known technique of adding synthetic variations to training data for transformation invariance, as used by Simard et al. (2003), Varga and Bunke (2003), Jaderberg et al. (2014), and many others.

For the compilation artifacts we use a stack of two LSTMs of 512 hidden units each, an observe-embedding CNN consisting of six convolutions and two linear layers organized as $[2 \times \text{Convolution}]$-$\text{MaxPooling}$-$[3 \times \text{Convolution}]$-$\text{MaxPooling}$-$\text{Convolution}$-$\text{MaxPooling}$-$\text{Linear}$-$\text{Linear}$, where convolutions are $3 \times 3$ with successively 64, 64,
We subsequently investigated how the trained models would perform on Captcha images collected from the web. We identified Wikipedia and Facebook as two major services still making use of textual Captchas, with step size 2, and the resulting embedding vector values of the means \( \{ \max_{\mu_k} p(\mu_k | y) \}_{k=1}^3 \) have been reported with blind review. Facebook Captchas are collected from a page for accessing groups. Wikipedia Captchas appear on the account creation page.

Table 1 reports inference results with test images sampled from the model, where we achieve very high recognition rates across the board. The reported results are obtained after approximately 16M training traces. With the resulting artifacts, running inference on a test Captcha takes < 100 ms, whereas durations ranging from 500 ms (Starostenko et al., 2015) to 7.95 s (Bursztein et al., 2014) have been observed on a test Captcha takes

Initially obtaining low recognition rates (\(< 10\%\)), with several iterations of model modifications (involving tuning of the prior distributions for font size and renderer noise), we were able to achieve 81% and 42% recognition rates with real Wikipedia and Facebook datasets, considerably higher than the threshold of 1% needed to deem a Captcha scheme broken (Bursztein et al., 2011). The fact that we had to tune our priors highlights the issues of model bias and “synthetic gap” (Zhang et al., 2015) when training models with synthetic data and testing with real data.\(^5\)

In our experiments we also investigated feeding the observe embeddings to the LSTM at all time steps versus only in the first time step. We empirically verified that both methods produce equivalent results, but the latter takes significantly \((\sim 3\times)\) longer to train. This is because we are training end-to-end from scratch, and the former setup results in more frequent gradient updates for \(f_{\text{obs}}\) per training trace. (Both Karpathy and Fei-Fei (2015) and Vinyals et al. (2015), who feed CNN output to an RNN only once, use pre-trained embedding layers.)

In summary, we only need to write a probabilistic generative model that produces Captchas sufficiently similar to those that we would like to solve. Using our CSIS framework, we get the inference neural network architecture, training data, and labels for free. If you

\[^{4}\text{Data sets are available, kept anonymous for this double-blind review. Facebook Captchas are collected from a page for accessing groups. Wikipedia Captchas appear on the account creation page.}\]

\[^{5}\text{Note that the synthetic/real boundary is not always clear: for instance, we assume that the Captcha results in Goodfellow et al. (2013) closely correspond to our results with synthetic test data because the authors have access to Google’s true generative process of reCaptcha images for their synthetic training data. Stark et al. (2015) both train and test their model with synthetic data.}\]
We have explored making use of deep neural networks for amortizing the cost of inference in probabilistic programming. In particular, we transform an inference problem given in the form of a probabilistic program into a trained neural network architecture that parameterizes proposal distributions during sequential importance sampling. The amortized inference technique presented here provides a framework within which to integrate the expressiveness of universal probabilistic programming languages for generative modeling and the processing speed of deep neural networks for inference. This merger addresses several fundamental challenges associated with its constituents: fast and scalable inference on probabilistic programs, interpretability of the generative model, an “infinite” stream of labeled training data, and the ability to correctly represent and handle uncertainty.

Our experimental results show that, for the family of models on which we focused, the proposed neural network architecture can be successfully trained to approximate the parameters of the posterior distribution in the sample space with nonlinear regression from the observe space. There are two aspects of this architecture that we are currently working on refining. Firstly, the structure of the neural network is not wholly determined by the given probabilistic program: the invariant LSTM core maintains long-term dependencies and acts as the glue between the embedding and proposal layers that are automatically configured for the address–instance pairs \((a_t, i_t)\) in the program traces. We would like to explore architectures where there is a tight correspondence between the neural artifact and the computational graph of the probabilistic program. Secondly, domain-specific observe embeddings such as the convolutional neural network that we designed for the Captcha-solving task are hand picked from a range of models on which we focused, the proposed neural network itself—as is done by the models organized around the variational autoencoder (Kingma and Welling, 2013; Burda et al., 2015)—is the possibility of model misspecification (Shalizi et al., 2009; Gelman and Shalizi, 2013). Section 3.3 explains that our generative model is ideally as close as possible to the true data generation process and remember the selection of potentially pretrained embeddings.

A limitation that comes with not learning the generative model itself—as is done by the models organized around the variational autoencoder (Kingma and Welling, 2013; Burda et al., 2015)—is the possibility of model misspecification (Shalizi et al., 2009; Gelman and Shalizi, 2013). Section 3.3 explains that our training setup is exempt from the common problem of overfitting to the training set. But as demonstrated by the fact that we needed alterations in our Captcha model priors for handling real data, we do have a risk of overfitting to the model. Therefore we need to ensure that our generative model is ideally as close as possible to the true data generation process and remember that misspecification in terms of broadness is preferable to a misspecification where we have a narrow, but uncalibrated, model.

5 DISCUSSION

We have explored making use of deep neural networks for amortizing the cost of inference in probabilistic programming. In particular, we transform an inference problem given in the form of a probabilistic program into a trained neural network architecture that parameterizes proposal distributions during sequential importance sampling. The amortized inference technique presented here provides a framework within which to integrate the expressiveness of universal probabilistic programming languages for generative modeling and the processing speed of deep neural networks for

---

**Table 1: Captcha recognition rates.**

<table>
<thead>
<tr>
<th></th>
<th>Baidu 2011</th>
<th>Baidu 2013</th>
<th>eBay</th>
<th>Yahoo</th>
<th>reCaptcha</th>
<th>Wikipedia</th>
<th>Facebook</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our method</td>
<td>99.8%</td>
<td>99.9%</td>
<td>99.2%</td>
<td>98.4%</td>
<td>96.4%</td>
<td>93.6%</td>
<td>91.0%</td>
</tr>
<tr>
<td>Bursztein et al. (2014)</td>
<td>38.68%</td>
<td>55.22%</td>
<td>51.39%</td>
<td>5.33%</td>
<td>22.67%</td>
<td>28.29%</td>
<td></td>
</tr>
<tr>
<td>Starostenko et al. (2015)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gao et al. (2014)</td>
<td>34%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gao et al. (2013)</td>
<td></td>
<td>51%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goodfellow et al. (2013)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stark et al. (2015)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Our experimental results show that, for the family of models on which we focused, the proposed neural network architecture can be successfully trained to approximate the parameters of the posterior distribution in the sample space with nonlinear regression from the observe space. There are two aspects of this architecture that we are currently working on refining. Firstly, the structure of the neural network is not wholly determined by the given probabilistic program: the invariant LSTM core maintains long-term dependencies and acts as the glue between the embedding and proposal layers that are automatically configured for the address–instance pairs \((a_t, i_t)\) in the program traces. We would like to explore architectures where there is a tight correspondence between the neural artifact and the computational graph of the probabilistic program. Secondly, domain-specific observe embeddings such as the convolutional neural network that we designed for the Captcha-solving task are hand picked from a range of fully-connected, convolutional, and recurrent architectures and trained end-to-end together with the rest of the architecture. Future work will explore automating the selection of potentially pretrained embeddings.

---

Figure 6: Pseudo algorithm and a sample trace of the Facebook Captcha generative process. Variations include sampling font styles, image coordinates for letter placement, and language-model-like letter ID distributions \(p(\lambda | \lambda_{1:t-1})\) (e.g., for meaningful Captchas). Noise parameters \(p(\pi)\) may or may not be a part of inference. At test time an observe statement that compares the generated Captcha with the ground truth is added after line 12.

can create instances of a Captcha, you can break it.
References


