

---

# Attention for Inference Compilation

---

**William Harvey**<sup>†</sup>

Dept. of Computer Science  
University of British Columbia  
Vancouver, B.C.  
Canada  
wsgh@cs.ubc.ca

**Andreas Munk**<sup>†</sup>

Dept. of Computer Science  
University of British Columbia  
Vancouver, B.C.  
Canada  
amunk@cs.ubc.ca

**Atılım Güneş Baydin**

Dept. of Engineering Science  
University of Oxford  
Oxford, OX2 6PN  
United Kingdom  
gunes@robots.ox.ac.uk

**Alexander Bergholm**

Dept. of Computer Science  
University of British Columbia  
Vancouver, B.C.  
Canada  
bergholm.alexander@gmail.com

**Frank Wood**

Dept. of Computer Science  
University of British Columbia  
Vancouver, B.C.  
Canada  
fwood@cs.ubc.ca

## Abstract

We present a new approach to automatic amortized inference in universal probabilistic programs which improves performance compared to current methods. Our approach is a variation of inference compilation (IC) which leverages deep neural networks to approximate a posterior distribution over latent variables in a probabilistic program. A challenge with existing IC network architectures is that they can fail to model long-range dependencies between latent variables. To address this, we introduce an attention mechanism that attends to the most salient variables previously sampled in the execution of a probabilistic program. We demonstrate that the addition of attention allows the proposal distributions to better match the true posterior, enhancing inference about latent variables in simulators.

generative models written as programs. Conditions on these random variables are imposed through **observe** statements, while the **sample** statements define latent variables we seek to draw inference about. Common to the different languages is the existence of an inference backend, which contains one or more general inference methods.

Recent research has addressed the task of making repeated inference less computationally expensive, by using up-front computation to reduce the cost of later executions, an approach known as amortized inference (Gershman and Goodman, 2014). One new method called inference compilation (IC) (Le et al., 2017) enables fast inference on arbitrarily complex and non-differentiable generative models. The approximate posterior distribution it learns can be combined with importance sampling at inference time, so that inference is asymptotically correct. It has been successfully used for Captcha solving (Le et al., 2017) and inference in particle physics simulators (Baydin et al., 2018).

## 1 INTRODUCTION

Probabilistic programming languages (van de Meent et al., 2018; Mansinghka et al., 2014; Milch et al., 2005; Wood et al., 2014; Pfeffer, 2009; Minka et al., 2018; Goodman et al., 2008; Gordon et al., 2014) allow for automatic inference about random variables in

<sup>†</sup>Equal contribution

The neural network used in IC is trained to approximate the joint posterior given the observed variables by sequentially proposing a distribution for each latent variable generated during an execution of a program. As such, capturing the possible dependencies on previously sampled variables is essential to achieve good performance. IC uses a Long Short Term Memory (LSTM)-based architecture (Hochreiter and Schmidhuber, 1997) to encapsulate these dependencies. However, this architecture fails to learn the dependency be-

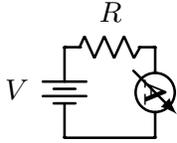


Figure 1: The electric circuit modelled by the probabilistic program in Figure 2.

tween highly dependent random variables when they are sampled far apart (with several other variables sampled in-between). This motivates allowing the neural network which parameterizes the proposal distribution for each latent variable to explicitly access any previously sampled variables. Inspired by the promising results of attention for tasks involving long-range dependencies (Jaderberg et al., 2015; Vaswani et al., 2017; Seo et al., 2016), we implemented an attention mechanism over previously sampled values. This enables the network to selectively attend to any combination of previously sampled values, regardless of their order and the trace length. We show that our approach significantly improves the approximation of the posterior, and hence facilitates faster inference.

The principle contributions of this paper are two-fold: we show that attention improves the performance of IC, and we show that we are able to use IC to perform fault detection via amortized inference in complex simulators. Section 2 introduces the concepts of probabilistic programming, inference compilation and attention for neural networks. We then describe our approach in Section 3 and our experiments in Section 4.

## 2 BACKGROUND

### 2.1 Probabilistic Programming

Probabilistic programming languages (PPLs) allow the specification of probabilistic generative models (and therefore probability distributions) as computer programs. Universal PPLs, which are based on Turing complete languages, may express models with an unbounded number of random variables. To this end, they combine traditional programming languages with the ability to sample a latent random variable (using syntax which we denote as a `sample` statement) and to condition these latent variables on the values of other, observed, random variables (using an `observe` statement). More formally, following (Le et al., 2017), we will operate on higher-order probabilistic programs in which we discuss the joint distribution of variables in an execution “trace”  $(x_t, a_t, i_t)$ , where  $t = 1, \dots, T$ , with  $T$  being the trace length (which may vary between executions).  $x_t$  denotes the value sampled at the

---

```
def circuit(measured_current):
    V = sample(Normal(mean=5, std=0.01))
    F = sample(Bernoulli(0.1)) # faulty or not
    if F:
        R = sample(Uniform(low=0, high=10))
    else:
        R = sample(Normal(mean=5, std=0.1))
    I = V/R # simulate circuit
    observe(measured_current,
            likelihood=Normal(mean=I, std=0.001))
    return R
```

---

Figure 2: Probabilistic program modeling the circuit in Figure 1 with a possibly faulty resistor. First the voltage,  $V$ , of the battery is sampled from a Gaussian prior centered on 5V. We then sample whether or not the resistor is faulty. If it is, its value is sampled from a broad uniform distribution. Otherwise, its value is sampled from a tightly peaked Gaussian. A noisy measurement of the current is then sampled from a Gaussian prior centered on the true value.

$t$ th `sample` statement encountered,  $a_t$  is the address of this `sample` statement and  $i_t$  represents the instance: the number of times the same address has been encountered previously, i.e.  $i_t = \sum_{j=i}^t \mathbb{1}(a_t = a_j)$ . We shall assume that there is a fixed number of observations,  $N$ , and these are denoted by  $\mathbf{y} = (y_1, \dots, y_N)$ , and we denote the latent variables as  $\mathbf{x} = (x_1, \dots, x_T)$ . Using this formalism, we express the joint distribution of a trace and observations as,

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

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. A mapping from the index,  $n$ , of the `observe` statement to the index of the most recent `sample` statement before the  $n$ th `observe` statement, is denoted by  $\tau$ .

As an example, consider the simple circuit as well as the probabilistic program shown in Figure 1, which expresses the joint distribution over the battery voltage,  $V$ , whether the resistor is faulty,  $F$ , the resistance of the resistor,  $R$ , and the measured current,  $I$ , as  $p(V, F, R, I) = p(I|V, R)p(R|F)p(F)p(V)$ .

Traces will have the form  $(x_t, a_t, i_t)_{t=1}^{T=3}$  where there are two trace “types,” one corresponding to the sequence of addresses of random variables generated if the resistor is faulty, and the other the opposite. In other words  $a_1$  is the address where  $V$  is sampled,  $a_2$  is the address where  $F$  is sampled, and  $a_3$  is the address from which  $R$  is sampled, which depends on  $F$ . The instance

counts in this program are always  $i_1 = i_2 = i_3 = 1$ , and the observation, `measured_current`  $\sim \mathcal{N}(I, 0.001)$ , with  $N = 1$ .

This generative model allows posterior inference to be performed over the joint distribution of the input voltage  $V$ , current  $I$ , and “faulty” variable  $F$  given the observed `measured_current`. Estimates of the marginal posterior distribution over  $F$  make it possible to directly answer questions such as whether the resistor is faulty or not. We will return to a more complex version of this problem in section 4.2. Generally, PPLs are designed to infer posterior distributions over the latent variables given the observations. Inference in probabilistic programs is carried out with algorithms such as Sequential Importance Sampling (SIS) (Arulampalam et al., 2002), Lightweight Metropolis-Hastings (Wingate et al., 2011), and Sequential Monte Carlo (Del Moral et al., 2006). However, these algorithms are too computationally expensive for use in real-time applications. Therefore, recent research (Le et al., 2017; Kulkarni et al., 2015) has considered amortizing the computational cost by performing up-front computation (for a given model) to allow faster inference later (given this model and any observed values).

## 2.2 Inference Compilation

IC (Le et al., 2017) is a generalized method for performing amortized inference in the framework of universal probabilistic programming. It involves training neural networks, which we describe as “inference networks,” whose outputs parameterize proposal distributions used for SIS.

### 2.2.1 Objective Function

In IC, we desire to match the proposal distribution,  $q(\mathbf{x}|\mathbf{y}; \phi) = \prod_{t=1}^T q_{a_t, i_t}(x_t | \eta_t(x_{1:t-1}, \mathbf{y}, \phi))$ , closely to the true posterior,  $p(\mathbf{x}|\mathbf{y})$ . The Kullback-Leibler divergence,  $D_{\text{KL}}(p(\mathbf{x}|\mathbf{y}) || q(\mathbf{x}|\mathbf{y}; \phi))$ , is used as a measure of this “closeness”. In order to ensure over any observed  $\mathbf{y}$ , an expectation of this divergence is taken with respect to  $p(\mathbf{y})$ ,

$$\begin{aligned} \mathcal{L}(\phi) &= \mathbb{E}_{p(\mathbf{y})}[D_{\text{KL}}(p(\mathbf{x}|\mathbf{y}) || q(\mathbf{x}|\mathbf{y}; \phi))] \\ &= \mathbb{E}_{p(\mathbf{x}, \mathbf{y})}[-\log q(\mathbf{x}|\mathbf{y}, \phi)] + \text{const} \end{aligned} \quad (2)$$

where the joint distribution takes the form of Eq. (1).

The parameters,  $\phi$ , are updated using gradient descent using the following estimate of the gradient of (2),

$$\nabla_{\phi} \mathcal{L}(\phi) \approx \frac{1}{M} \sum_{m=1}^M -\nabla_{\phi} \log q(\mathbf{x}^m | \mathbf{y}^m, \phi), \quad (3)$$

where  $(\mathbf{x}^m, \mathbf{y}^m) \sim p(\mathbf{x}, \mathbf{y})$  for  $m = \{1, \dots, M\}$ . Note that the loss used, and the estimates of the gradi-

ents, are identical to those in the sleep-phase of wake-sleep (Hinton et al., 1995).

### 2.2.2 Architecture

The architecture used in IC (Baydin et al., 2018; Le et al., 2017) consists of the black components shown in Figure 3b. Before performing inference, observations  $\mathbf{y}$  are embedded by a learned `observe` embedder,  $f^{obs}$ . At each `sample` statement encountered as the program is run, the LSTM is run for one time step. It receives an input consisting of the concatenation of the embedding of the observed values,  $f^{obs}(\mathbf{y})$ , an embedding of  $x_{t-1}$ , the value sampled at the previous `sample` statement, embeddings of the current address, instance and distribution-type, denoted  $a_t$ ,  $i_t$  and  $d_t$  respectively, embeddings of the previous address, instance and distribution type:  $a_{t-1}$ ,  $i_{t-1}$  and  $d_{t-1}$ .

The embedder used for  $x_{t-1}$  is specific to  $(a_{t-1}, i_{t-1})$ , the address and instance from which  $x_{t-1}$  was sampled. The output of the LSTM is fed into a proposal layer, which is specific to the address and instance ( $a_t$  and  $i_t$ ). The proposal layer outputs the parameters,  $\eta_t$ , of a proposal distribution for the variable at this `sample` statement.

### 2.2.3 Inference using Sequential Importance Sampling

In IC inference is performed by SIS (Arulampalam et al., 2002; Doucet and Johansen, 2009), which is compatible with latent variable inference in higher-order probabilistic programs (Wood et al., 2014). SIS produces a set of  $K$  weighted samples  $\{(\mathbf{x}_k, w_k)\}_{k=1}^K$ , such that the posterior and expectations of a function  $g(\mathbf{x})$  are approximated by

$$\begin{aligned} p(\mathbf{x}|\mathbf{y}) &\approx \hat{p}(\mathbf{x}|\mathbf{y}) = \frac{\sum_{k=1}^K w_k \delta(\mathbf{x}_k - \mathbf{x})}{\sum_{k=1}^K w_k} \\ \mathbb{E}[g(\mathbf{x})] &\approx \frac{\sum_{k=1}^K w_k g(\mathbf{x}_k)}{\sum_{k=1}^K w_k}, \end{aligned} \quad (4)$$

where  $\delta$  is the Dirac delta function. Each weight  $w_k$  is calculated for each trace  $\mathbf{x}_k$  according to

$$w_k = \prod_{n=1}^N g_n(y_n | x_{1:\tau_k(n)}^k) \prod_{t=1}^{T_k} \frac{f_{a_t}(x_t^k | x_{1:t-1}^k)}{q_{a_t, i_t}(x_t^k | x_{1:t-1}^k)},$$

where  $T_k$  denotes the  $k$ th trace length with  $k = \{1, \dots, K\}$ , and  $q_{a_t, i_t}$  is given by the inference network learned by minimizing the loss function (3).

## 2.3 Dot-product Attention

Attention has recently been shown to be useful in a number of tasks, including image captioning, machine

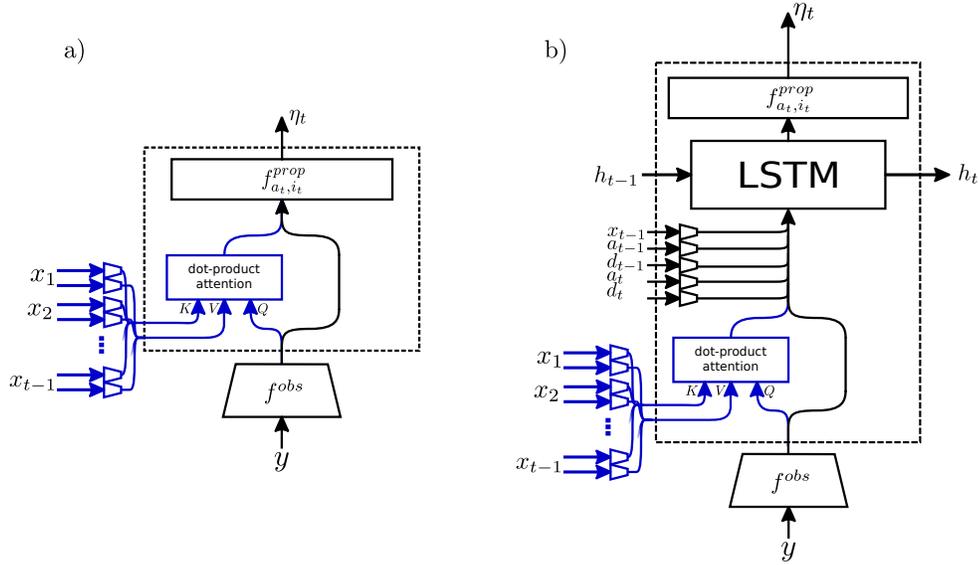


Figure 3: Feedforward and LSTM neural network architectures with attention mechanisms. The components inside the dashed line are run once at each `sample` statement in a program trace, while the parts outside this line are only run once per trace. The attention mechanism is denoted in blue.

translation, and image generation (Xu et al., 2015; Bahdanau et al., 2014; Gregor et al., 2015). The two broad types of attention are hard and soft attention. Hard attention (Ba et al., 2014; Xu et al., 2015) selects a single “location” to attend to, and thus requires only this location to be embedded. However, it is non-differentiable. In contrast, soft attention mechanisms (Vaswani et al., 2017; Xu et al., 2015) are fully differentiable. They typically require splitting the input into a finite number of locations. Each location is embedded separately, and a weighted average of these embeddings is returned as the output.

Our proposed architecture incorporates *dot-product* attention (Luong et al., 2015; Vaswani et al., 2017), a form of soft attention. This choice is justified by the embeddings of the value sampled at time step  $t$  being used at all later time steps  $t + 1, \dots, T$ . This means that the computational cost of calculating the embeddings scales linearly with the trace length. Since this is no worse than the rate that hard attention achieves, we select soft attention for its differentiability. The specific use of dot-product attention is due to the efficiency of the calculation of attention weights. Each trace requires the computation of  $\mathcal{O}(T^2)$  attention weights and some programs may contain thousands of `sample` statements (Baydin et al., 2018) so fast weight computation is paramount.

The dot-product attention module (Luong et al., 2015; Vaswani et al., 2017), shown in Figure 4, receives three inputs: one or more query vectors (which describe the desired properties of the locations to attend to), a key

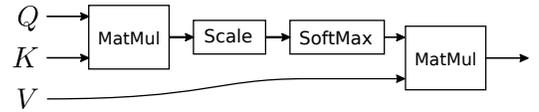


Figure 4: A scaled dot-product attention mechanism. Figure adapted from (Vaswani et al., 2017)

vector for each location, and a value vector for each location. In Figure 3 in the Appendix, these are represented as the matrices  $Q \in \mathbb{R}^{q \times k}$ ,  $K \in \mathbb{R}^{k \times l}$ , and  $V \in \mathbb{R}^{l \times v}$ , respectively. Here,  $l$  is the number of locations,  $k$  is the length of each query and key embedding,  $v$  is the length of each value embedding, and  $q$  is the number of queries. For each query, attention weights are computed for every location by taking the dot-product of the query vector and the relevant key. A SoftMax is then applied to ensure that the sum of the weights over every location is 1 (for each query). These weights are used to compute weighted averages of the values. The output of the attention mechanism is a concatenation of these. This procedure can be performed efficiently using matrix multiplications, and so calculating the attention weights is more computationally efficient than in other types of soft attention (which typically differ in how the weights are calculated) (Bahdanau et al., 2014). In scaled dot-product attention (Vaswani et al., 2017), the product  $QK$  is multiplied by a scalar before performing the softmax. This scalar is chosen to ensure that the output of the softmax is not saturated upon initialising the weights, so the gradients propagated through are large enough

for effective training.

### 3 METHOD

We compare against two “baseline architectures” in all experiments: a feedforward network learned for each address and instance pair  $(a, i)$ ; and the LSTM-based architecture described in section 2.2.2. These are shown as the black components of Figure 3a and 3b respectively. The feedforward inference network consists of a single proposal layer,  $f_{a_t, i_t}^{prop}$ . This embedding layer takes all observe embeddings as input and computes  $\eta_t$  as output. Since no part of its input is dependent on the previously sampled values, unlike the LSTM architectures, this architecture is completely unable to learn proposal distributions with dependencies between latent variables before the addition of an attention mechanism.

To attend to previously sampled variables we add dot-product attention, described in section 2.3, to both the LSTM and feedforward architectures, as shown in Figure 3. During training we build a data structure,  $d_{k,v,q}$ , with associative mappings linking address/instance pairs  $(a, i)$  to key, value and query embedders. The embedders in  $d_{k,v,q}$  are constructed dynamically for each new address and instance pair  $(a_t, i_t)$  encountered.

During inference, the queries, keys, and values input to the attention mechanism at each `sample` statement are calculated as follows: for the first `sample` statement, identified by  $(a_1, i_1)$ , no previously sampled variables exist and so the attention module outputs a vector of zeros. Using the associated key and value embedders in  $d_{k,v,q}$ , the variable sampled,  $x_1$ , is embedded to yield a key and a value,  $k_1$  and  $v_1$ .  $(k_1, v_1)$  are kept in memory throughout the trace, allowing fast access for subsequent `sample` statements. The second `sample` statement can attend to the first sampled variable via  $(k_1, v_1)$  using a query. The embedder used for finding the query takes as input the observe embedding,  $f^{obs}(\mathbf{y})$ , and is specific to the current address and instance  $(a_2, i_2)$ . As with the key/value embedders, the query embedder is found in  $d_{k,v,q}$ . The output of the attention module is then fed to the LSTM or proposal layer (see Figure 3). As for  $x_1$ ,  $x_2$  is sampled and embedded using the embedders stored in  $d_{k,v,q}$ , yielding the key, value pair  $(k_2, v_2)$ . This procedure is repeated until the end of the trace, as defined by the probabilistic program. In the context of higher-order programs, an address and instance pair may be encountered during inference that has not been seen during training. In this case the proposal layers are not trained, and so the standard IC approach is to use the prior as a proposal distribution. For the same reason, the key/-

value embedders do not exist and so no keys or values are created for this  $(a_t, i_t)$ . This prevents later `sample` statements from attending to the variable sampled at  $(a_t, i_t)$ .

### 4 EXPERIMENTS AND RESULTS

We implemented our attention mechanism in, and performed experiments with, *pyprob* (Le et al., 2017; Baydin and Le, 2018), a PPL designed for IC. For both experiments the inference networks were trained using Adam (Kingma and Ba, 2014) with hyperparameters  $\beta_1 = 0.9$  and  $\beta_2 = 0.999$  to optimize the loss given in Eq. (2). The attention modules we consider use  $q = 4$ ,  $k = 16$  and  $v = 8$ . We consider feedforward and LSTM architectures both with and without attention, denoted as  $M_{FF \text{ w/o ATT}}$ ,  $M_{FF \text{ w/ ATT}}$ ,  $M_{LSTM \text{ w/o ATT}}$  and  $M_{LSTM \text{ w/ ATT}}$ .

We test the architectures in two experiments: an illustrative example involving the estimation of a highly correlated posterior distribution over two latent variables, and an application of amortized inference to the problem of real-time circuit fault diagnosis.

We use the effective sample size (Kong, 1992) (ESS) of each estimated posterior as a quantitative metric to describe the quality of the proposal distribution, and resulting posterior estimate. The effective sample size is not a perfect metric for the quality of a posterior as, for example, it depends only on the importance weights of the posterior samples, and so can be high even if the samples lack diversity and/or miss modes. However, we use it alongside qualitative plots which show that the improvements in ESS come about through qualitatively better proposals and estimates of the posterior distributions.

#### 4.1 Magnitude of Random Vector

We first demonstrate the efficacy of our approach on a simple, pedagogical example. The task is to infer a distribution over two latent variables,  $x$  and  $y$ , conditioned on  $\hat{r}^2$ , a noisy observation of  $x^2 + y^2$ . The generative model first samples  $(x, y)$  coordinates from identical and independent Gaussian priors:  $x \sim \mathcal{N}(0, \sigma_p)$ ,  $y \sim \mathcal{N}(0, \sigma_p)$ . A noisy estimate of the radius,  $\hat{r}$ , is then observed with a likelihood given by  $p(\hat{r}^2 | x, y) = \mathcal{N}(\hat{r}^2 | x^2 + y^2, \sigma_l)$ . We use  $\sigma_p = 10$  and  $\sigma_l = 0.5$ , which leads to a tightly peaked posterior exhibiting circular symmetry, and so a strong dependence between  $x$  and  $y$ .

In the model as just described, where  $x$  and  $y$  are sampled consecutively, it is trivial for either the LSTM or the attentive architecture to learn this relationship. However, we are interested in testing the learning of

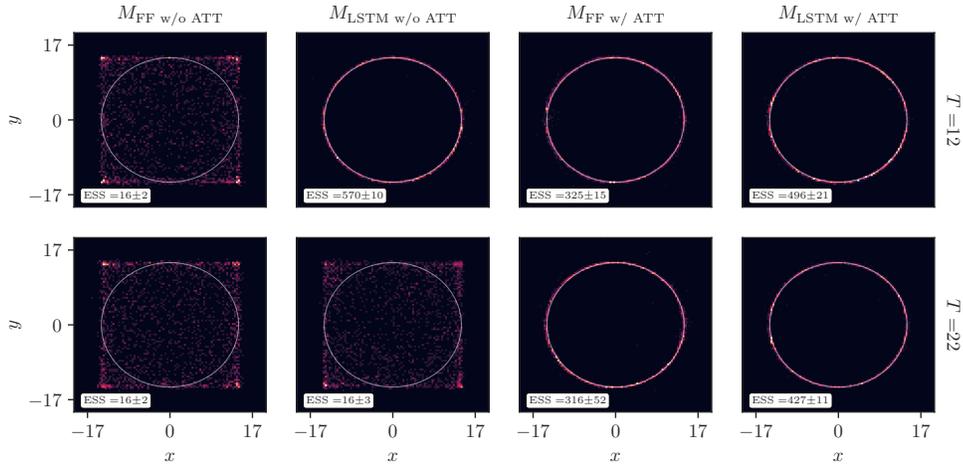


Figure 5: Different IC neural network architectures, with and without attention, showing improved IC performance for those with attention on an inference task involving highly correlated latent random variables. Each image shows 2000 posterior samples of  $x$  and  $y$  from the model described in section 4 conditioned on  $\hat{r}^2 = 200$ . The top row contains samples from program with 10 nuisance variables, i.e. trace length  $T = 12$ . The bottom row contains samples from the model with 20 nuisance variables, i.e. trace length  $T = 22$ . The white circles indicate the mode of the true posterior.  $M_{FF\ w/o\ ATT}$  is unable to learn any dependency between  $x$  and  $y$ . With 10 nuisance variables, we see that attention provides no advantage over the standard LSTM IC architecture. However, as the number of nuisance variables increases from 10 to 20, attention becomes beneficial, maintaining a high effective sample size (ESS) while the performance of  $M_{LSTM\ w/o\ ATT}$  reduces to that of  $M_{FF\ w/o\ ATT}$ . The results were similar when different radii were observed. The ESS in the figure is calculated by averaging 10 estimates.

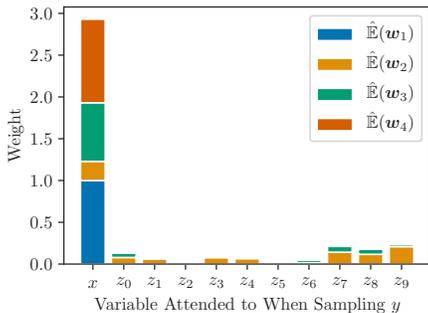


Figure 6: Attention weights used on each previously sampled variable.

long-range dependencies. We therefore consider variations of this model where, after sampling  $x$ , and before sampling  $y$ , some number of “nuisance” random variables are sampled. These nuisance random variables are not used elsewhere in the program, and so serve only to increase the “distance” between  $x$  and  $y$ . In particular, we consider two programs: one containing 10 “nuisance” variables, and one containing 20. Program 1 provides pseudocode for these programs. All inference networks are trained using  $6 \times 10^5$  traces with minibatches of size 128. The learning rate

is decreased every  $2 \times 10^5$  traces, iterating through  $\{10^{-3}, 10^{-4}, 10^{-5}\}$ .

Figure 5 shows 2000 samples from the proposal distributions for each program,  $q(x, y | \hat{r}^2) = \int_{z_{1:i}} q(x, y, z_{1:i} | \hat{r}^2) dz_{1:i}$  for  $i = \{10, 20\}$ , produced by each architecture. We see that  $M_{FF\ w/o\ ATT}$  is unable to make proposals capturing the dependency between  $x$  and  $y$ . This illustrates that allowing the neural network to “remember” (via attention or the LSTM core) the sampled value of  $x$  is necessary to accurately approximate the posterior. We observe that with only 10 nuisance random variables there is no advantage in using the attention mechanism compared to an LSTM core. However, when the number of nuisance random variables increases to 20 the LSTM core is no longer able to capture the dependency between  $x$  and  $y$ . In contrast, the architectures with attention mechanisms are unaffected.

Program 1: Generative model for the magnitude of a random vector with  $M$  nuisance random variables.

```
def magnitude(obs, M):
    x = sample(Normal(0, 10))
    for _ in range(M):
        # nuisance variables to extend trace
        _ = sample(Normal(0, 10))
    y = sample(Normal(0, 10))
```

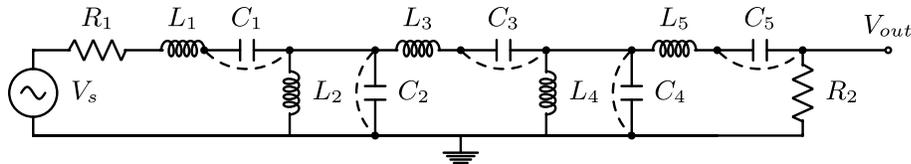


Figure 7: Fifth-order band-pass Butterworth filter with resistors, capacitors and inductors denoted by  $R$ ,  $C$ , and  $L$  respectively. The dashed lines represent possible short circuits. The existence of these short circuits and whether or not each component is faulty (represented by a noisy component value) or disconnected is sampled according to the generative model. Given observations of  $V_{out}$  for various input frequencies, the task is to infer a distribution over possible faults such as short circuits and poorly connected or incorrectly valued components.

```

observe(obs2,
        Likelihood=Normal(x2 + y2, 0.1))
return x, y

```

Figure 6 shows the average attention weights given to each previously sampled variable (by each of the four queries) when creating a proposal distribution for  $y$ . It can be seen that queries 1 and 4 attend solely to  $x$ , explaining how the attention mechanism enables the inference network to capture the long-term dependency, and ignore the nuisance variables.

## 4.2 Electronic Circuit Fault Diagnosis

For our second experiment, we consider performing inference in a probabilistic program that imports and uses a pre-existing electronic circuit simulator (Venturini et al., 2017). Specifically, we will consider a Butterworth filter as shown in Figure 7. This is operated with an input voltage composed of a 5V DC signal and a 1V AC signal. The observable output voltages at different AC frequencies are shown in Figure 4 in the Appendix. The task is to infer whether or not each component in the Butterworth filter is faulty given the observed complex-valued output voltage  $V_{out}$  (i.e. voltage magnitude and phase) at 40 different frequencies. To perform inference we write a probabilistic program that iterates through each component of the circuit and samples in the following order: first, whether or not it is correctly connected to the rest of the circuit. Second, the component value is sampled from a mixture of a broad uniform distribution and a tightly peaked Gaussian, both centered on the nominal value. The value is sampled from the tightly peaked Gaussian with 98% probability and from the uniform distribution with 2% probability. Conceptually, one can interpret the tightly peaked Gaussian as the distribution given that the component has been correctly made. The broad uniform distribution represents the distribution for components that are faulty.

To test each inference network, we generate 100 different observations by running the probabilistic program, and attempt to infer the posterior using each different network architecture. For each inference network

architecture, we estimate the posterior distribution 5 times using importance sampling with 20 traces each time. Across the 5 estimates, we compute the average ESS, and average these over all 100 observations. The averaged results were 1.40 for  $M_{FF \text{ w/o ATT}}$ , 7.26 for  $M_{LSTM \text{ w/o ATT}}$ , 8.46 for  $M_{FF \text{ w/ ATT}}$  and 8.35 for  $M_{LSTM \text{ w/ ATT}}$ .

The attention-based architecture has an 16.5% higher average ESS than the LSTM core, showing that the use of attention leads to quantitatively better proposal distributions. We further find that whenever the observed signal appears to originate from a correctly working Butterworth filter, all architectures seem to produce reasonable predictive posterior distributions - i.e. the distribution of the voltage signal generated by the sampled latent variables. However, the attention-based architectures yield a higher average ESS with only a few exceptions. When the observed signal clearly originates from an erroneous filter,  $M_{FF \text{ w/o ATT}}$  produces predictive posterior distributions which poorly fit the observed data. The LSTM-based architecture produces better predictive posterior distributions but these are still significantly worse than the distributions produced by the attention-based architecture in almost all cases where the filter is broken. Figure 8 shows inference performance for one such observation originating from a filter in which the component is faulty. We plot voltages generated according to the sampled latent variables from the predictive proposal distributions using each architecture. The proposals from  $M_{FF \text{ w/ ATT}}$  and  $M_{LSTM \text{ w/ ATT}}$  are clustered near to the observations, whereas the proposals from  $M_{FF \text{ w/o ATT}}$  and  $M_{LSTM \text{ w/o ATT}}$  produce many proposals that do not fit the observations.

We suspect that these outliers occur due to the inability of  $M_{FF \text{ w/o ATT}}$  and  $M_{LSTM \text{ w/o ATT}}$  to learn long-range dependencies. For example, an output voltage of zero could be explained by a number of different faults (e.g. a short-circuit across  $C_2$  or across  $C_4$ ). If the resulting dependency between these can be learned, the proposals could consistently predict that only one is broken (predicting more would be unlikely due to the strong prior on parts working). However, if the

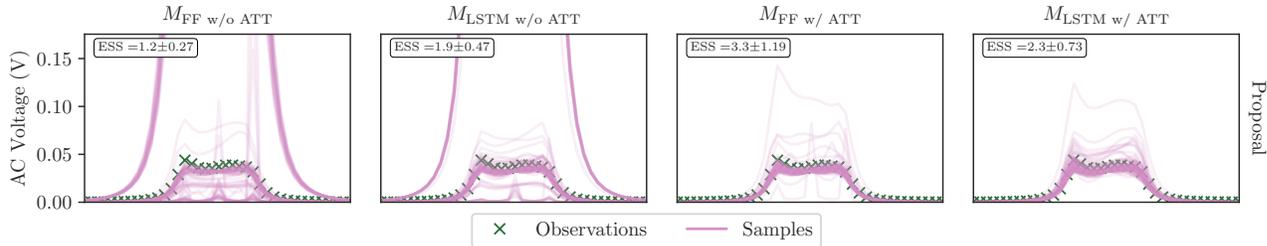


Figure 8: Reconstruction of the output voltage using samples from each proposal distribution. In the architectures with attention, the sampled voltages are almost all close to the observations (green ‘x’s) whereas, without attention, the proposals place high probability in regions which do not fit the observations. These better proposal distributions lead the higher effective sample sizes shown in each figure (mean and standard deviation, calculated with 5 estimates). The proposal distribution is shown using 100 samples form each architecture.

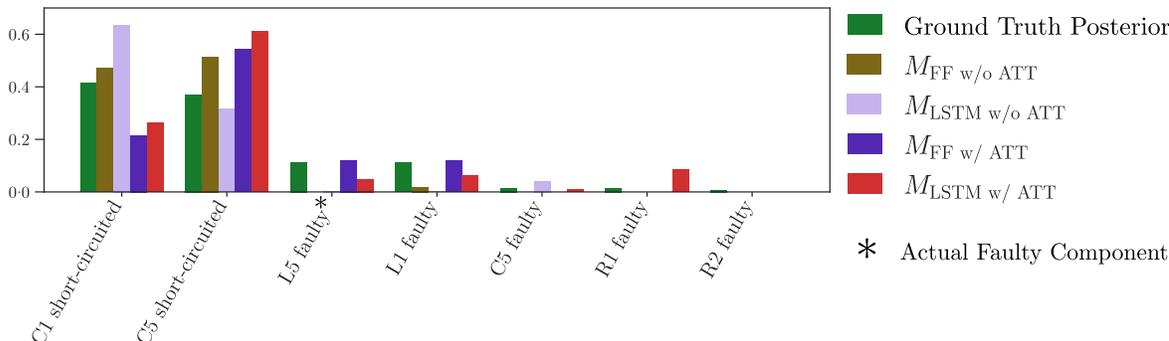


Figure 9: Posterior probability assigned to possible causes of failure. The observations used are the same as those in Figure 8. In the simulation which generated these observations,  $L_1$  was the component at fault. However, the observations do not contain enough information to infer this exactly, and so being close to the ground truth posterior (as shown in green) is the best the inference networks can do. The ground truth posterior was inferred using importance sampling with no inference network and 17 million samples. The inferred posteriors shown were each estimated using 1000 samples. It can be seen that the architectures without attention place very little probability on  $L_1$  or  $L_5$  being broken, despite these having a combined probability of about 20% according to the true posterior. In particular, they place almost no probability on  $L_5$  being faulty, even though it was the faulty component in the simulation used to create the observed voltages.

dependency is not captured, the proposals would be prone to predicting that zero or multiple components are broken. This interpretation is supported by Figure 8, where both architectures without attention are seen to sometimes propose an output voltage corresponding closely to a working circuit.

Figure 9 shows an example of the posterior distributions inferred over possible faults by each architecture. For this purpose, a component is considered faulty when its value is outside of a 0.3% tolerance of its nominal value.  $L_5$  was at fault when the observations used were generated, but the observations do not allow this to be inferred with confidence as other failures could have produced the same observations. This is reflected in the ground truth posterior placing little probability on  $L_5$  being faulty. The architectures with attention manage to most closely fit the ground truth posterior. In particular, they assign probabil-

ity to  $L_5$  being faulty whereas the other architectures assign very little.

## 5 DISCUSSION AND CONCLUSION

We have demonstrated that the standard LSTM core used in IC can fail to capture long-range dependencies between latent variables. To address this, we have proposed an attention mechanism which enables the inference network to attend to the most salient previously sampled variables in an execution trace. We further demonstrate that attention improves inference in a practical application of IC, yielding better proposal distributions and thus posteriors with an effective sample size 16.5% higher than when using the LSTM core.

Our work leads to several avenues of future research: first, while we demonstrate the efficacy of an architecture using only an attention mechanism relative to

an architecture using only an LSTM, further investigation is needed to determine whether there could be additional benefits from using both in conjunction. Additionally, this work raises the possibility of extending the usage of such attention mechanisms to attend to the observations in a generic way. This could enable the inference network to filter out noise in the observations and reduce computation time. Furthermore, the inference compilation framework currently requires a fixed number of observations. An attention mechanism over the observations may allow this requirement to be relaxed.

### Acknowledgements

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), the Canada CIFAR AI Chairs Program, Compute Canada, Intel, and DARPA under its D3M and LWLL programs.

### References

- Arulampalam, M., Maskell, S., Gordon, N., and Clapp, T. (2002). A tutorial on particle filters for on-line nonlinear/non-gaussian bayesian tracking. *Ieee Transactions on Signal Processing*, 50(2):174–188.
- Ba, J., Mnih, V., and Kavukcuoglu, K. (2014). Multiple object recognition with visual attention. *arXiv preprint arXiv:1412.7755*.
- Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.
- Baydin, A. G., Heinrich, L., Bhimji, W., Gram-Hansen, B., Louppe, G., Shao, L., Cranmer, K., Wood, F., et al. (2018). Efficient probabilistic inference in the quest for physics beyond the standard model. *arXiv preprint arXiv:1807.07706*.
- Baydin, A. G. and Le, T. A. (2018). *pyprob*.
- Del Moral, P., Doucet, A., and Jasra, A. (2006). Sequential monte carlo samplers. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(3):411–436.
- Doucet, A. and Johansen, A. M. (2009). A tutorial on particle filtering and smoothing: Fifteen years later. *Handbook of nonlinear filtering*, 12(656-704):3.
- Gershman, S. and Goodman, N. (2014). Amortized inference in probabilistic reasoning. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, volume 36.
- Goodman, N. D., Mansinghka, V. K., Roy, D., Bonawitz, K., and Tenenbaum, J. B. (2008). Church: A language for generative models. *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence, Uai 2008*, pages 220–229.
- Gordon, A. D., Henzinger, T. A., Nori, A. V., and Rajamani, S. K. (2014). Probabilistic programming. *Future of Software Engineering, Fose 2014 - Proceedings*, pages 167–181.
- Gregor, K., Danihelka, I., Graves, A., Rezende, D. J., and Wierstra, D. (2015). Draw: A recurrent neural network for image generation. *arXiv preprint arXiv:1502.04623*.
- Hinton, G. E., Dayan, P., Frey, B. J., and Neal, R. M. (1995). The "wake-sleep" algorithm for unsupervised neural networks. *Science*, 268(5214):1158–1161.
- Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8):1735–1780.
- Jaderberg, M., Simonyan, K., Zisserman, A., et al. (2015). Spatial transformer networks. In *Advances in neural information processing systems*, pages 2017–2025.
- Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.
- Kong, A. (1992). A note on importance sampling using standardized weights. *University of Chicago, Dept. of Statistics, Tech. Rep.*, 348.
- Kulkarni, T. D., Kohli, P., Tenenbaum, J. B., and Mansinghka, V. (2015). Picture: A probabilistic programming language for scene perception. In *Proceedings of the ieee conference on computer vision and pattern recognition*, pages 4390–4399.
- Le, T. A., Baydin, A. G., and Wood, F. (2017). Inference compilation and universal probabilistic programming. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, volume 54 of *Proceedings of Machine Learning Research*, pages 1338–1348, Fort Lauderdale, FL, USA. PMLR.
- Luong, M.-T., Pham, H., and Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*.
- Mansinghka, V., Selsam, D., and Perov, Y. (2014). Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*.
- Milch, B., Marthi, B., Russell, S., Sontag, D., Ong, D. L., and Kolobov, A. (2005). Blog: Probabilistic models with unknown objects. *Ijcai International Joint Conference on Artificial Intelligence*, pages 1352–1359.

- Minka, T., Winn, J., Guiver, J., Zaykov, Y., Fabian, D., and Bronskill, J. (2018). /Infer.NET 0.3. Microsoft Research Cambridge. <http://dotnet.github.io/infer>.
- Pfeffer, A. (2009). Figaro: An object-oriented probabilistic programming language. *Charles River Analytics Technical Report*, 137:96.
- Seo, M., Kembhavi, A., Farhadi, A., and Hajishirzi, H. (2016). Bidirectional attention flow for machine comprehension. *arXiv preprint arXiv:1611.01603*.
- van de Meent, J.-W., Paige, B., Yang, H., and Wood, F. (2018). An introduction to probabilistic programming. *arXiv preprint arXiv:1809.10756*.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008.
- Venturini, G., Daniher, I., Crowther, R., and KOLANICH (2017). *ahkab*.
- Wingate, D., Andreas Stuhlmüller, A., and Goodman, N. D. (2011). Lightweight implementations of probabilistic programming languages via transformational compilation. *Journal of Machine Learning Research*, 15:770–778.
- Wood, F., Meent, J. W., and Mansinghka, V. (2014). A new approach to probabilistic programming inference. In *Artificial Intelligence and Statistics*, pages 1024–1032.
- Xu, K., Ba, J., Kiros, R., Cho, K., Courville, A., Salakhudinov, R., Zemel, R., and Bengio, Y. (2015). Show, attend and tell: Neural image caption generation with visual attention. In *International Conference on Machine Learning*, pages 2048–2057.