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.

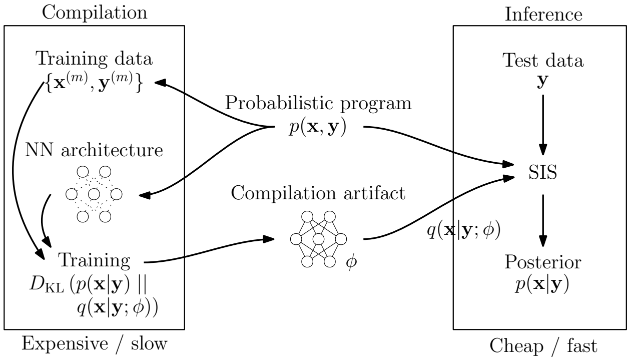

Compilation
Training data
$\{\mathbf{x}^{(m)}, \mathbf{y}^{(m)}\}$
NN architecture
Training
$D_{\mathrm{KL}}\left(p(\mathbf{x}|\mathbf{y}) \;||\; q(\mathbf{x}|\mathbf{y};\phi)\right)$
Expensive / slow
Probabilistic program
$p(\mathbf{x}, \mathbf{y})$
Compilation artifact
ϕ
$q(\mathbf{x}|\mathbf{y};\phi)$
Inference
Test data
$\mathbf{y}$
SIS
Posterior
$p(\mathbf{x}|\mathbf{y})$
Cheap / fast

Figure 1: Our approach to compiled inference. Given only a probabilistic program $p(\mathbf{x}, \mathbf{y})$, during *compilation* we automatically construct a neural network architecture comprising an LSTM core and various embedding and proposal layers specified by the probabilistic program and train this using an infinite stream of training data $\{\mathbf{x}^{(m)}, \mathbf{y}^{(m)}\}$ generated from the model. When this expensive compilation stage is complete, we are left with an artifact of weights ϕ and neural architecture specialized for the given probabilistic program. During *inference*, the probabilistic program and the compilation artifact is used in a sequential importance sampling procedure, where the artifact parameterizes the proposal distribution $q(\mathbf{x}|\mathbf{y};\phi)$.

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., 2008), 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 (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

Proceedings of the 20^{th} International Conference on Artificial Intelligence and Statistics (AISTATS) 2017, Fort Lauderdale, Florida, USA. JMLR: W&CP volume 54. Copyright 2017 by the author(s).

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 instead focuses on 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, 2014; Burda et al., 2016), 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 nonlinearly 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 inference compilation for sequential importance sampling, 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 (Gordon et al., 2014). 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 fixed. Hence we denote the observed values by $\mathbf{y} := (y_n)_{n=1}^N$ 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

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

where x_t, a_t, and i_t are respectively the sample value, address, and instance (call number) of the tth entry in a given trace, and T is a trace-dependent length. Instance values $i_t = \sum_{j=1}^t \mathbb{1}(a_t = a_j)$ 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 $\mathbf{x} := (x_t)_{t=1}^T$ holds the T sampled values from the `sample` statements.

The joint probability density of an execution trace is

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

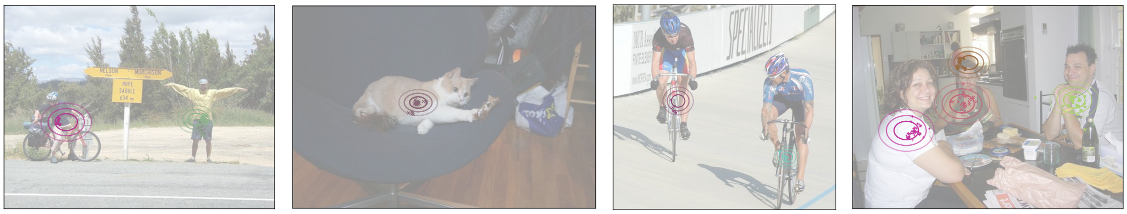


Figure 2: Results from counting and localizing objects detected in the PASCAL VOC 2007 dataset (Everingham et al., 2010). We use the corresponding categories of object detectors (i.e., person, cat, bicycle) from the MatConvNet (Vedaldi and Lenc, 2015) implementation of the Fast R-CNN (Girshick, 2015). The detector output is processed by using a high detection threshold and summarized by representing the bounding box detector output by a single central point. Inference using a single trained neural network was able to accurately identify both the number of detected objects and their locations for all categories. MAP results from 100 particles.

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 nth `observe` statement. $f_{a_t}(\cdot|x_{1:t-1})$ is called the prior conditional density given the sample values $x_{1:t-1}$ obtained before encountering the tth `sample` statement. $g_n(\cdot|x_{1:\tau(n)})$ is called the likelihood density given the sample values $x_{1:\tau(n)}$ obtained before encountering the nth `observe` statement, where τ 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 $p(\mathbf{x}|\mathbf{y})$ and its expected values $I_\zeta = \int \zeta(\mathbf{x}) p(\mathbf{x}|\mathbf{y})\,\mathrm{d}\mathbf{x}$ over chosen functions ζ.

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. This is related, but different to the approaches that adapt proposal distributions for the importance sampling family of algorithms (Gu et al., 2015; Cheng and Druzdzel, 2000).

2.2 Sequential Importance Sampling

Sequential importance sampling (SIS) (Arulampalam 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 $\{(w^k, \mathbf{x}^k)\}_{k=1}^K$ is used to approximate the posterior and the expectations of functions as

$$\hat{p}(\mathbf{x}|\mathbf{y}) = \sum_{k=1}^{K} w^k \delta(\mathbf{x}^k - \mathbf{x}) / \sum_{j=1}^{K} w^j \tag{3}$$

$$\hat{I}_\zeta = \sum_{k=1}^{K} w^k \zeta(\mathbf{x}^k) / \sum_{j=1}^{K} w^j, \tag{4}$$

where δ 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 $x_{1:T^k}^k$ 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_t^k is sampled from the proposal distribution $q_{a_t,i_t}(\cdot|x_{1:t-1}^k)$ given the proposal sample values until that point. We obtain K proposal execution traces $\mathbf{x}^k := x_{1:T^k}^k$ (possibly in parallel) to which we assign weights

$$w^k = \prod_{n=1}^{N} g_n(y_n|x_{1:\tau_k(n)}^k) \cdot \prod_{t=1}^{T^k} \frac{f_{a_t}(x_t^k|x_{1:t-1}^k)}{q_{a_t,i_t}(x_t^k|x_{1:t-1}^k)} \tag{5}$$

for $k = 1, \ldots, K$ with T^k denoting the length of the kth proposal execution trace.

3 APPROACH

We achieve inference compilation in universal probabilistic programming systems through proposal distribution adaptation, approximating $p(\mathbf{x}|\mathbf{y})$ in the framework of SIS. Assuming we have a set of adapted proposals $q_{a_t,i_t}(x_t|x_{1:t-1}, \mathbf{y})$ such that their joint $q(\mathbf{x}|\mathbf{y})$ is close to $p(\mathbf{x}|\mathbf{y})$, the resulting inference algorithm remains unchanged from the one described in Section 2.2, except the replacement of $q_{a_t,i_t}(x_t|x_{1:t-1})$ by $q_{a_t,i_t}(x_t|x_{1:t-1}, \mathbf{y})$.

Inference compilation amounts to minimizing a function, specifically the loss of a neural network architecture, which makes the proposal distributions good in the sense that we specify in Section 3.1. The process of generating training data for this neural network architecture from the generative model is described in Section 3.2. 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 3.3.

3.1 Objective Function

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

$$\begin{aligned}\mathcal{L}(\phi) &:= \mathbb{E}_{p(\mathbf{y})}\left[D_{\mathrm{KL}}\left(p(\mathbf{x}|\mathbf{y}) \;||\; q(\mathbf{x}|\mathbf{y};\phi)\right)\right] && (6)\\ &= \int_{\mathbf{y}} p(\mathbf{y}) \int_{\mathbf{x}} p(\mathbf{x}|\mathbf{y}) \log \frac{p(\mathbf{x}|\mathbf{y})}{q(\mathbf{x}|\mathbf{y};\phi)}\,\mathrm{d}\mathbf{x}\,\mathrm{d}\mathbf{y} \\ &= \mathbb{E}_{p(\mathbf{x},\mathbf{y})}\left[-\log q(\mathbf{x}|\mathbf{y};\phi)\right] + \text{const.} && (7)\end{aligned}$$

This objective function corresponds to the negative entropy criterion. Individual adapted proposals $q_{a_t,i_t}(x_t|\eta_t(x_{1:t-1},\mathbf{y},\phi)) =: q_{a_t,i_t}(x_t|x_{1:t-1},\mathbf{y})$ depend on η_t, the output of the neural network at time step t, parameterized by ϕ.

Considering the factorization

$$q(\mathbf{x}|\mathbf{y};\phi) = \prod_{t=1}^{T} q_{a_t,i_t}(x_t|\eta_t(x_{1:t-1},\mathbf{y},\phi)) \;, \qquad (8)$$

the neural network architecture must be able to map to a variable number of outputs, and incorporate sampled values in a sequential manner, concurrent with the running of the inference engine. We describe our neural network architecture in detail in Section 3.3.

3.2 Training Data

Since Eq. 7 is an expectation over the joint distribution, we can use the following noisy unbiased estimate of its gradient to minimize the objective:

$$\frac{\partial}{\partial\phi}\mathcal{L}(\phi) \approx \frac{1}{M}\sum_{m=1}^{M} \frac{\partial}{\partial\phi}\left(-\log q(\mathbf{x}^{(m)}|\mathbf{y}^{(m)};\phi)\right) \qquad (9)$$

$$(\mathbf{x}^{(m)},\mathbf{y}^{(m)}) \sim p(\mathbf{x},\mathbf{y}),\; m = 1,\ldots,M \;. \qquad (10)$$

Here, $(\mathbf{x}^{(m)},\mathbf{y}^{(m)})$ is the mth 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 `observe` statement in the original program by `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 (Cotter et al., 2011) sampled from the generative model $p(\mathbf{x},\mathbf{y})$:

$$\mathcal{D}_{\text{train}} = \left\{ \left(x_t^{(m)}, a_t^{(m)}, i_t^{(m)}\right)_{t=1}^{T^{(m)}}, \left(y_n^{(m)}\right)_{n=1}^{N} \right\}_{m=1}^{M} , \qquad (11)$$

where M is the minibatch size, and, for a given trace m, the sample values, addresses, and instances are respectively denoted $x_t^{(m)}$, $a_t^{(m)}$, and $i_t^{(m)}$, and the values sampled from the distributions in `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, specifically Adam (Kingma and Ba, 2015), for optimizing the neural network weights ϕ.

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 (Prechelt, 1998). We do sample a validation set that remains fixed during training to compute validation losses for tracking the progress of training in a less noisy way than that admitted by the training loss.

3.3 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 ϕ.

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, 2016). We use RNNs in this work owing to their ability to encode dependencies over time in the hidden state. In particular, we use the long short-term memory (LSTM) architecture which helps mitigate the vanishing and exploding gradient problems of RNNs (Hochreiter and Schmidhuber, 1997).

The overall architecture (Figure 3) is formed by combining the LSTM core with a domain-specific `observe` embedding layer f^{obs}, and several `sample` embedding layers $f^{\mathrm{smp}}_{a,i}$ and proposal layers $f^{\mathrm{prop}}_{a,i}$ that are distinct for each address–instance pair (a, i). As described in Section 3.2, 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 $f^{\mathrm{obs}}(\mathbf{y})$. 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 ρ_t of the LSTM is constructed as a concatenation of

1. the `observe` embedding $f^{\mathrm{obs}}(\mathbf{y})$,
2. the embedding of the previous `sample` $f^{\mathrm{smp}}_{a_{t-1},i_{t-1}}(x_{t-1})$, using zero for $t = 1$, and
3. the one-hot encodings of the current address a_t, instance i_t, and proposal type type(a_t) of the `sample` statement

for which the artifact will generate the parameter η_t of the proposal distribution $q_{a_t,i_t}(\cdot|\eta_t)$. The parameter η_t is obtained via the proposal layer $f^{\mathrm{prop}}_{a_t,i_t}(h_t)$, mapping the LSTM output h_t through the corresponding proposal layer. The LSTM network has the capacity to incorporate inputs in its hidden state. This allows the parametric proposal $q_{a_t,i_t}(x_t|\eta_t(x_{1:t-1}, \mathbf{y}, \phi))$ to take into account all previous samples and all observations.

During training (compilation), we supply the actual sample values $x^{(m)}_{t-1}$ to the embedding $f^{\mathrm{smp}}_{a_{t-1},i_{t-1}}$, and we are interested in the parameter η_t in order to calculate the per-sample gradient $\frac{\partial}{\partial\phi} - \log q_{a^{(m)}_t,i^{(m)}_t}(x^{(m)}_t|\eta_t(x_{1:t-1}, \mathbf{y}, \phi))$ to use in SGD.

During inference, the evaluation proceeds by requesting proposal parameters η_t from the artifact for specific address–instance pairs (a_t, i_t) as these are encountered. The value x_{t-1} is sampled from the proposal distribution in the previous time step.

The neural network artifact is implemented in Torch (Collobert et al., 2011), and it uses a ZeroMQ-based protocol for interfacing with the Anglican probabilistic programming system (Wood et al., 2014). 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. The source code for our framework and for reproducing the experiments in this paper can be found on our project page.[1]

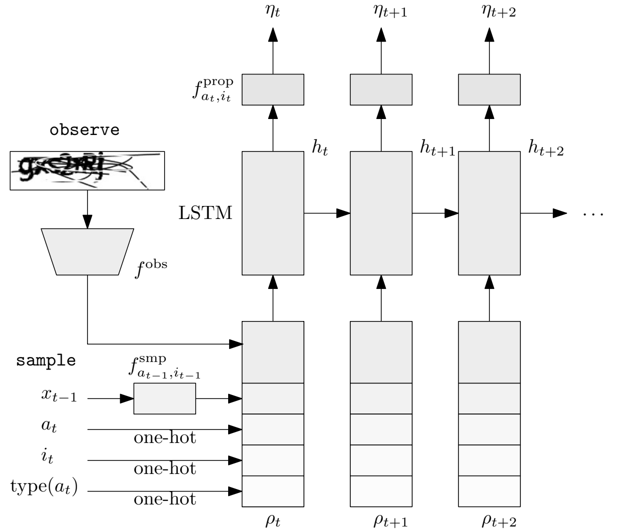


Figure 3: The neural network architecture. f^{obs}: `observe` embedding; $f^{\mathrm{smp}}_{a_{t-1},i_{t-1}}$: sample embeddings; x_{t-1}: previous `sample` value; a_t, i_t, type(a_t): one-hot encodings of current address, instance, proposal type; ρ_t: LSTM input; h_t: LSTM output; $f^{\mathrm{prop}}_{a_t,i_t}$: proposal layers; η_t: proposal parameters. Note that the LSTM core can possibly be a stack of multiple LSTMs.

4 EXPERIMENTS

We demonstrate our inference compilation framework on two examples. In our first example we demonstrate an open-universe mixture model. In our second, we demonstrate Captcha solving via probabilistic inference (Mansinghka et al., 2013).[2]

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 identify how many, where, and how big the clusters are, and optionally, which data points belong to each cluster.

We investigate inference compilation for a two-dimensional GMM in which the number of clusters is unknown. Inference arises from observing the val-

[1] `https://probprog.github.io/inference-compilation/`

[2] A video of inference on real test data for both examples is available at: `https://youtu.be/m-FYEXVyQjQ`

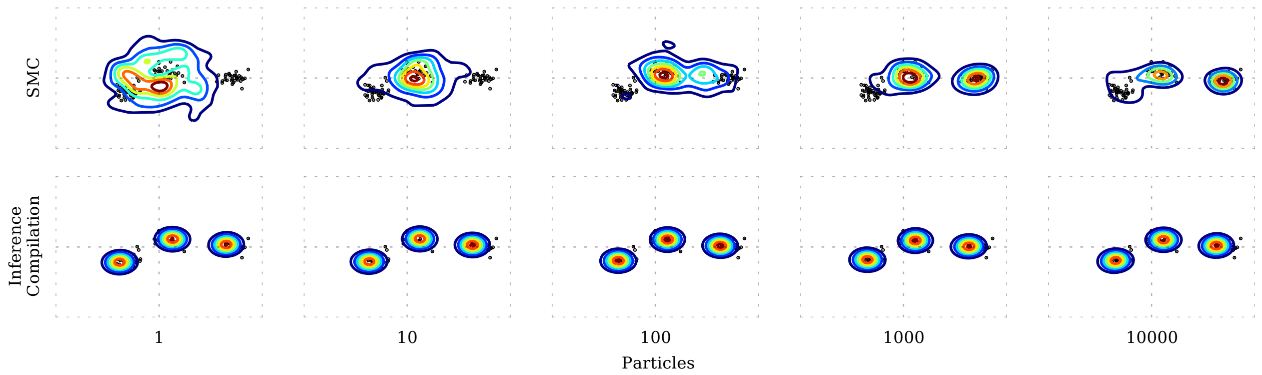


Figure 4: Typical inference results for an isotropic Gaussian mixture model with number of clusters fixed to $K = 3$. Shown in all panels: kernel density estimation of the distribution over maximum a posteriori values of the means $\{\max_{\mu_k} p(\mu_k|\mathbf{y})\}_{k=1}^3$ over 50 independent runs. This figure illustrates the uncertainty in the estimate of where cluster means are for each given number of particles, or equivalently, fixed amount of computation. The top row shows that, given more computation, inference, as expected, slowly becomes less noisy in expectation. In contrast, the bottom row shows that the proposal learned and used by inference compilation produces a low-noise, highly accurate estimate given even a very small amount of computation. Effectively, the encoder learns to simultaneously localize all of the clusters highly accurately.

ues of y_n (Figure 5, line 9) 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 normalized to lie within $[-1, 1]$ in both dimensions.

In order to make good proposals for such inference, the neural network 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, we select a convolutional neural network as the observation embedding, whose input is a two-dimensional histogram image of binned observed data $\mathbf{y}$.

In presenting observational data $\mathbf{y}$ 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 the Euclidian distance of their means 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.

```
 1: procedure GaussianMixture
 2:     $K \sim p(K)$                                   ▷ sample number of clusters
 3:     for $k = 1, \ldots, K$ do
 4:         $\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k \sim p(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$    ▷ sample cluster parameters
 5: Generate data:
 6:     $\boldsymbol{\pi} \leftarrow$uniform$(1, K)$
 7:     for $n = 1, \ldots, N$ do
 8:         $z_n \sim p(z_n|\boldsymbol{\pi})$                      ▷ sample class label
 9:         $y_n \sim p(y_n|z_n = k, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$       ▷ sample data
10:     return $y_n$
```

Figure 5: Pseudo algorithm for generating Gaussian mixtures of a variable number of clusters. At test time we `observe` data y_n and infer $K, \{\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\}_{k=1}^K$.

Figure 4, where we fix the number of clusters to 3, shows that we are able to learn a proposal that makes inference dramatically more efficient than sequential Monte Carlo (SMC) (Doucet and Johansen, 2009). Figure 2 shows one kind of application such an efficient inference engine can do: simultaneous object counting (Lempitsky and Zisserman, 2010) and localization for computer vision, where we achieve counting by setting the prior $p(K)$ over number of clusters to be a uniform distribution over $\{1, 2, \ldots, 5\}$.

4.2 Captcha Solving

We also demonstrate our inference compilation framework by writing generative probabilistic models for Captchas (von Ahn et al., 2003) and comparing our re-

```
procedure CAPTCHA
    $\nu \sim p(\nu)$                          ▷ sample number of letters
    $\kappa \sim p(\kappa)$                    ▷ sample kerning value
Generate letters:
    $\Lambda \leftarrow \{\}$
    for $i = 1, \ldots, \nu$ do
        $\lambda \sim p(\lambda)$              ▷ sample letter identity
        $\Lambda \leftarrow \text{append}(\Lambda, \lambda)$
Render:
    $\gamma \leftarrow \text{render}(\Lambda, \kappa)$
    $\pi \sim p(\pi)$                          ▷ sample noise parameters
    $\gamma \leftarrow \text{noise}(\gamma, \pi)$
    return $\gamma$
```

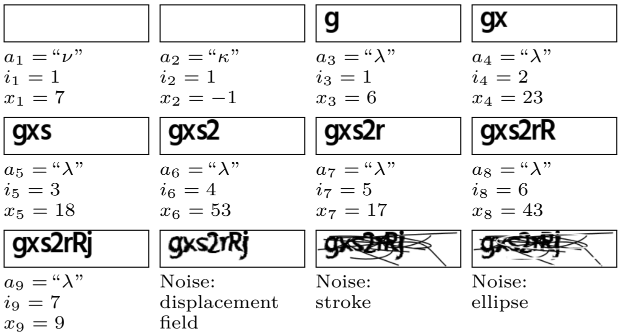
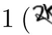
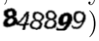
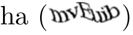
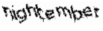

a_1 = "ν" $i_1 = 1$ $x_1 = 7$
g
a_2 = "κ" $i_2 = 1$ $x_2 = -1$
gx
a_3 = "λ" $i_3 = 1$ $x_3 = 6$
a_4 = "λ" $i_4 = 2$ $x_4 = 23$
gxs
gxs2
gxs2r
gxs2rR
a_5 = "λ" $i_5 = 3$ $x_5 = 18$
a_6 = "λ" $i_6 = 4$ $x_6 = 53$
a_7 = "λ" $i_7 = 5$ $x_7 = 17$
a_8 = "λ" $i_8 = 6$ $x_8 = 43$
gxs2rRj
a_9 = "λ" $i_9 = 7$ $x_9 = 9$
Noise: displacement field
Noise: stroke
Noise: ellipse

Figure 6: Pseudo algorithm and a sample trace of the Facebook Captcha generative process. Variations include sampling font styles, coordinates for letter placement, and language-model-like letter identity distributions $p(\lambda|\lambda_{1:t-1})$ (e.g., for meaningful Captchas). Noise parameters π may or may not be a part of inference. At test time we observe image γ and infer ν, Λ.

sults 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., 2014; 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 (), Baidu 2013 ( ), eBay (), Yahoo ( ), reCaptcha (), and Wikipedia (). 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 technique of Simard et al. (2003) proved instrumental in achieving our results. Note that the parameters of stochastic renderer noise are not inferred in the example of 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×Convolution]-MaxPooling-[3×Convolution]-MaxPooling-Convolution-MaxPooling-Linear-Linear, where convolutions are 3×3 with successively 64, 64, 64, 128, 128, 128 filters, max-pooling layers are 2×2 with step size 2, and the resulting embedding vector is of length 1024. All convolutions and linear layers are followed by ReLU activation. Depending on the particular style, each artifact has approximately 20M trainable parameters. Artifacts are trained end-to-end using Adam (Kingma and Ba, 2015) with initial learning rate $\alpha = 0.0001$, hyperparameters $\beta_1 = 0.9$, $\beta_2 = 0.999$, and minibatches of size 128.

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 reported with segment-and-classify approaches. We also compared our approach with the one by Mansinghka et al. (2013). Their method is slow since it must be run anew for each Captcha, taking in the order of minutes to solve one Captcha in our implementation of their method. The probabilistic program must also be written in a way amenable to Markov Chain Monte Carlo inference such as having auxiliary indicator random variables for rendering letters to overcome multimodality in the posterior.

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, and collected and labeled test sets of 500 images each.[3] 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

[3] Facebook Captchas are collected from a page for accessing groups. Wikipedia Captchas appear on the account creation page.

Table 1: Captcha recognition rates.

	Baidu 2011	Baidu 2013	eBay	Yahoo	reCaptcha	Wikipedia	Facebook
Our method	99.8%	99.9%	99.2%	98.4%	96.4%	93.6%	91.0%
Bursztein et al. (2014)	38.68%	55.22%	51.39%	5.33%	22.67%	28.29%	
Starostenko et al. (2015)				91.5%	54.6%		
Gao et al. (2014)	34%			55%	34%		
Gao et al. (2013)		51%		36%			
Goodfellow et al. (2014)					99.8%		
Stark et al. (2015)					90%		

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.[4]

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 (approx. 3 times) longer to train. This is because we are training $f^{\mathbf{obs}}$ end-to-end from scratch, and the former setup results in more frequent gradient updates for $f^{\mathbf{obs}}$ per training trace.[5]

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 inference compilation framework, we get the inference neural network architecture, training data, and labels for free. If you can create instances of a Captcha, you can break it.

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 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 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.

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, 2014; Burda et al., 2016)—is the possibility of model misspecification (Shalizi et al., 2009; Gelman and Shalizi, 2013). Section 3.2 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.

[4] Note that the synthetic/real boundary is not always clear: for instance, we assume that the Captcha results in Goodfellow et al. (2014) 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.

[5] Both Karpathy and Fei-Fei (2015) and Vinyals et al. (2015), who feed CNN output to an RNN only once, use pretrained embedding layers.

Acknowledgements

We would like to thank Hakan Bilen for his help with the MatConvNet setup and showing us how to use his Fast R-CNN implementation and Tom Rainforth for his helpful advice. Tuan Anh Le is supported by EPSRC DTA and Google (project code DF6700) studentships. Atılım Güneş Baydin and Frank Wood are supported under DARPA PPAML through the U.S. AFRL under Cooperative Agreement FA8750-14-2-0006, Sub Award number 61160290-111668.

References

M. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp. A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking. *IEEE Transactions on Signal Processing*, 50(2):174–188, 2002.

D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations (ICLR)*, 2014.

Y. Burda, R. Grosse, and R. Salakhutdinov. Importance weighted autoencoders. In *International Conference on Learning Representations (ICLR)*, 2016.

E. Bursztein, M. Martin, and J. Mitchell. Text-based CAPTCHA strengths and weaknesses. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 125–138. ACM, 2011.

E. Bursztein, J. Aigrain, A. Moscicki, and J. C. Mitchell. The end is nigh: generic solving of text-based CAPTCHAs. In *8th USENIX Workshop on Offensive Technologies (WOOT 14)*, 2014.

B. Carpenter, A. Gelman, M. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. A. Brubaker, J. Guo, P. Li, and A. Riddell. Stan: a probabilistic programming language. *Journal of Statistical Software*, 2015.

J. Cheng and M. J. Druzdzel. Ais-bn: An adaptive importance sampling algorithm for evidential reasoning in large bayesian networks. *Journal of Artificial Intelligence Research*, 2000.

R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A MATLAB-like environment for machine learning. In *BigLearn, NIPS Workshop*, EPFL-CONF-192376, 2011.

A. Cotter, O. Shamir, N. Srebro, and K. Sridharan. Better mini-batch algorithms via accelerated gradient methods. In *Advances in Neural Information Processing Systems*, pages 1647–1655, 2011.

J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. aurelio Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng. Large scale distributed deep networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1223–1231. Curran Associates, Inc., 2012.

A. Doucet and A. M. Johansen. A tutorial on particle filtering and smoothing: Fifteen years later. *Handbook of Nonlinear Filtering*, 12(656–704):3, 2009.

M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman. The Pascal visual object classes (VOC) challenge. *International Journal of Computer Vision*, 88(2):303–338, 2010.

H. Gao, W. Wang, J. Qi, X. Wang, X. Liu, and J. Yan. The robustness of hollow CAPTCHAs. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, pages 1075–1086. ACM, 2013.

H. Gao, W. Wang, Y. Fan, J. Qi, and X. Liu. The robustness of “connecting characters together” CAPTCHAs. *Journal of Information Science and Engineering*, 30(2):347–369, 2014.

A. Gelman and C. R. Shalizi. Philosophy and the practice of Bayesian statistics. *British Journal of Mathematical and Statistical Psychology*, 66(1):8–38, 2013.

S. J. Gershman and N. D. Goodman. Amortized inference in probabilistic reasoning. In *Proceedings of the 36th Annual Conference of the Cognitive Science Society*, 2014.

R. Girshick. Fast R-CNN. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1440–1448, 2015.

I. J. Goodfellow, Y. Bulatov, J. Ibarz, S. Arnoud, and V. Shet. Multi-digit number recognition from street view imagery using deep convolutional neural networks. In *International Conference on Learning Representations (ICLR)*, 2014.

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

A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani. Probabilistic programming. In *Future of Software Engineering, FOSE 2014*, pages 167–181. ACM, 2014.

S. Gu, Z. Ghahramani, and R. E. Turner. Neural adaptive sequential Monte Carlo. In *Advances in Neural Information Processing Systems*, pages 2611–2619, 2015.

G. E. Hinton, P. Dayan, B. J. Frey, and R. M. Neal. The wake-sleep algorithm for unsupervised neural networks. *Science*, 268(5214):1158–1161, 1995.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

M. Jaderberg, K. Simonyan, A. Vedaldi, and A. Zisserman. Synthetic data and artificial neural networks for natural scene text recognition. In *Workshop on Deep Learning, NIPS*, 2014.

M. J. Johnson, D. Duvenaud, A. B. Wiltschko, S. R. Datta, and R. P. Adams. Structured VAEs: Composing probabilistic graphical models and variational autoencoders. *arXiv preprint arXiv:1603.06277*, 2016.

A. Karpathy and L. Fei-Fei. Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3128–3137, 2015.

D. Kingma and J. Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015.

D. P. Kingma and M. Welling. Auto-encoding variational Bayes. In *International Conference on Learning Representations (ICLR)*, 2014.

T. D. Kulkarni, P. Kohli, J. B. Tenenbaum, and V. K. Mansinghka. Picture: a probabilistic programming language for scene perception. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

V. Lempitsky and A. Zisserman. Learning to count objects in images. In *Advances in Neural Information Processing Systems*, pages 1324–1332, 2010.

D. J. Lunn, A. Thomas, N. Best, and D. Spiegelhalter. WinBUGS–a Bayesian modelling framework: concepts, structure, and extensibility. *Statistics and Computing*, 10(4):325–337, 2000.

V. Mansinghka, T. D. Kulkarni, Y. N. Perov, and J. Tenenbaum. Approximate Bayesian image interpretation using generative probabilistic graphics programs. In *Advances in Neural Information Processing Systems*, pages 1520–1528, 2013.

V. Mansinghka, D. Selsam, and Y. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*, 2014.

T. Minka, J. Winn, J. Guiver, S. Webster, Y. Zaykov, B. Yangel, A. Spengler, and J. Bronskill. Infer.NET 2.6, 2014. Microsoft Research Cambridge. http://research.microsoft.com/infernet.

R. Nishihara, T. Minka, and D. Tarlow. Detecting parameter symmetries in probabilistic models. *arXiv preprint arXiv:1312.5386*, 2013.

B. Paige and F. Wood. Inference networks for sequential Monte Carlo in graphical models. In *Proceedings of the 33rd International Conference on Machine Learning*, volume 48 of *JMLR*, 2016.

B. Paige, F. Wood, A. Doucet, and Y. W. Teh. Asynchronous anytime sequential Monte Carlo. In *Advances in Neural Information Processing Systems*, pages 3410–3418, 2014.

L. Prechelt. Early stopping — but when? In *Neural Networks: Tricks of the Trade*, pages 55–69. Springer, 1998.

T. Rainforth, C. A. Naesseth, F. Lindsten, B. Paige, J.-W. van de Meent, A. Doucet, and F. Wood. Interacting particle Markov chain Monte Carlo. In *Proceedings of the 33rd International Conference on Machine Learning*, volume 48 of *JMLR: W&CP*, 2016.

S. Reed and N. de Freitas. Neural programmer-interpreters. In *International Conference on Learning Representations (ICLR)*, 2016.

D. Ritchie, A. Stuhlmüller, and N. D. Goodman. C3: Lightweight incrementalized MCMC for probabilistic programs using continuations and callsite caching. In *AISTATS 2016*, 2016a.

D. Ritchie, A. Thomas, P. Hanrahan, and N. Goodman. Neurally-guided procedural models: Amortized inference for procedural graphics programs using neural networks. In *Advances In Neural Information Processing Systems*, pages 622–630, 2016b.

C. R. Shalizi et al. Dynamics of Bayesian updating with dependent data and misspecified models. *Electronic Journal of Statistics*, 3:1039–1074, 2009.

P. Y. Simard, D. Steinkraus, and J. C. Platt. Best practices for convolutional neural networks applied to visual document analysis. In *Proceedings of the Seventh International Conference on Document Analysis and Recognition – Volume 2*, ICDAR '03, pages 958–962, Washington, DC, 2003. IEEE Computer Society.

F. Stark, C. Hazırbaş, R. Triebel, and D. Cremers. Captcha recognition with active deep learning. In *GCPR Workshop on New Challenges in Neural Computation*, Aachen, Germany, 2015.

O. Starostenko, C. Cruz-Perez, F. Uceda-Ponga, and V. Alarcon-Aquino. Breaking text-based CAPTCHAs with variable word and character orientation. *Pattern Recognition*, 48(4):1101–1112, 2015.

I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, pages 3104–3112, 2014.

T. Varga and H. Bunke. Generation of synthetic training data for an hmm-based handwriting recognition system. In *Seventh International Conference on Document Analysis and Recognition, 2003*, pages 618–622. IEEE, 2003.

A. Vedaldi and K. Lenc. Matconvnet – convolutional neural networks for MATLAB. In *Proceeding of the ACM International Conference on Multimedia*, 2015.

S. Venugopalan, H. Xu, J. Donahue, M. Rohrbach, R. Mooney, and K. Saenko. Translating videos to natural language using deep recurrent neural networks. *arXiv preprint arXiv:1412.4729*, 2014.

O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and tell: A neural image caption generator. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3156–3164, 2015.

L. von Ahn, M. Blum, N. J. Hopper, and J. Langford. CAPTCHA: Using hard AI problems for security. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 294–311. Springer, 2003.

D. Wingate, A. Stuhlmüller, and N. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics*, pages 770–778, 2011.

F. Wood, J. W. van de Meent, and V. Mansinghka. A new approach to probabilistic programming inference. In *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*, pages 1024–1032, 2014.

X. Zhang, Y. Fu, S. Jiang, L. Sigal, and G. Agam. Learning from synthetic data using a stacked multichannel autoencoder. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 461–464, Dec 2015. doi: 10.1109/ICMLA.2015.199.