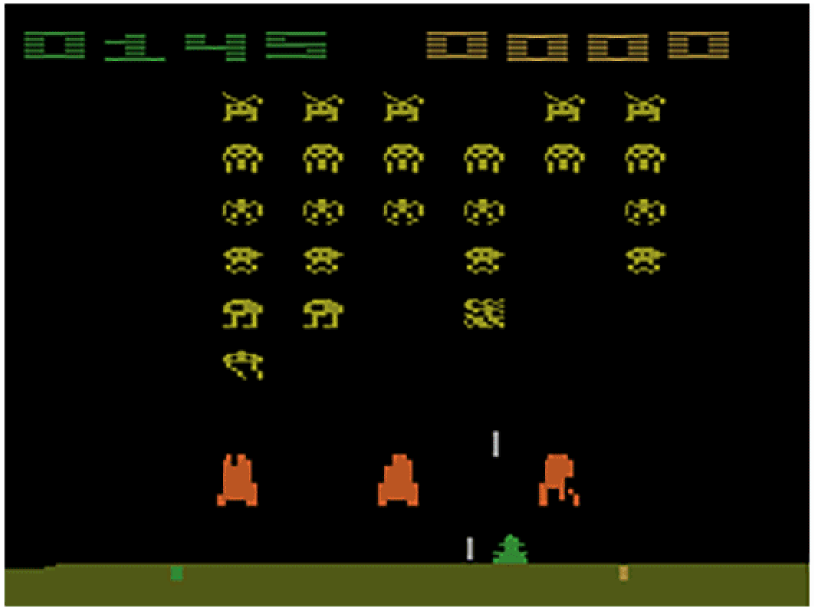
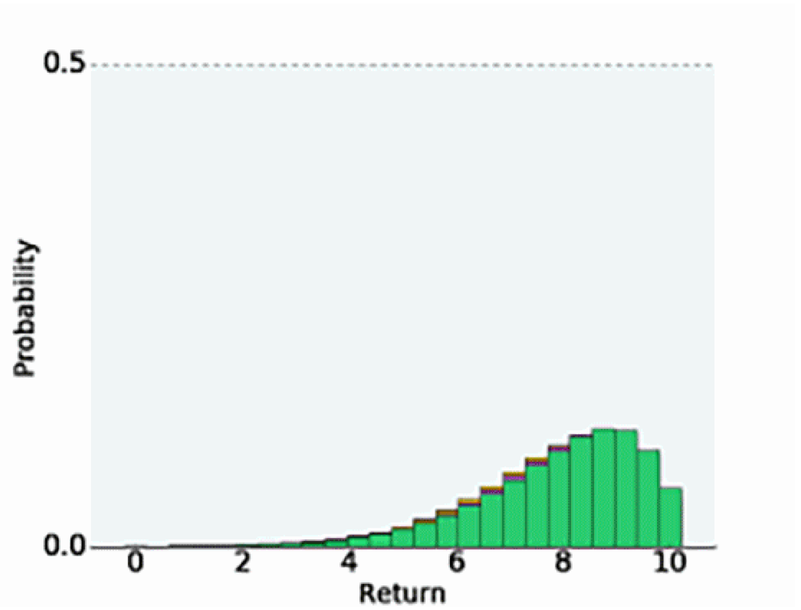
Probabilistic programming still matters

Rob Zinkov

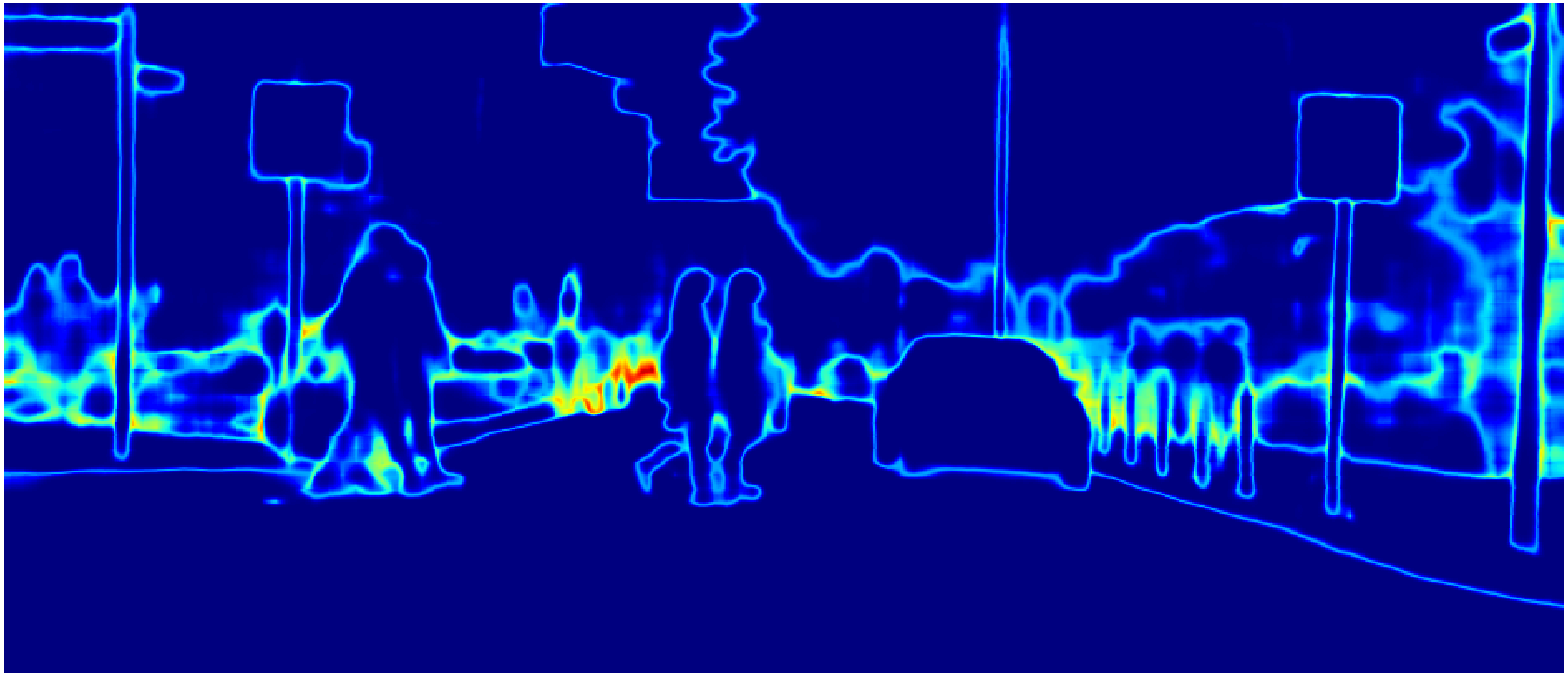
Why probabilistic modeling matters

Uncertainty in exploration

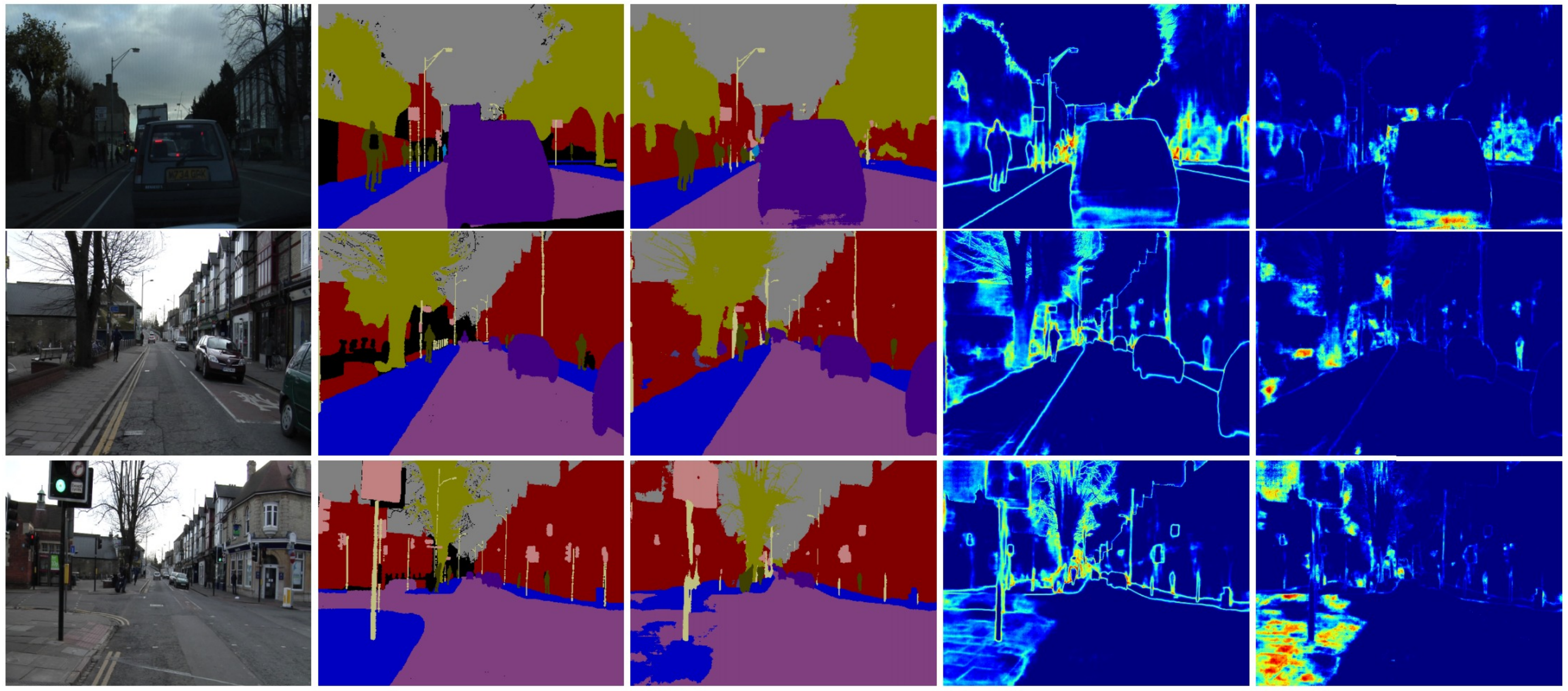




Modeling uncertainty

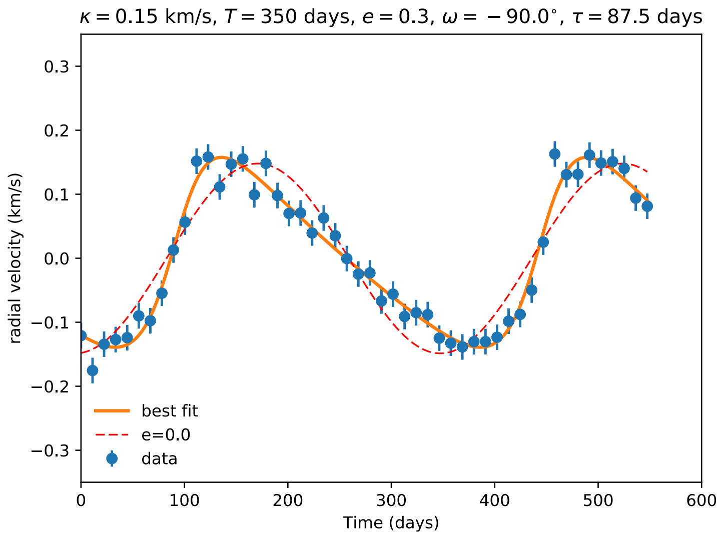


Modeling uncertainty



(a) Input Image (b) Ground Truth (c) Semantic Segmentation (d) Aleatoric Uncertainty (e) Epistemic Uncertainty

Exploring models



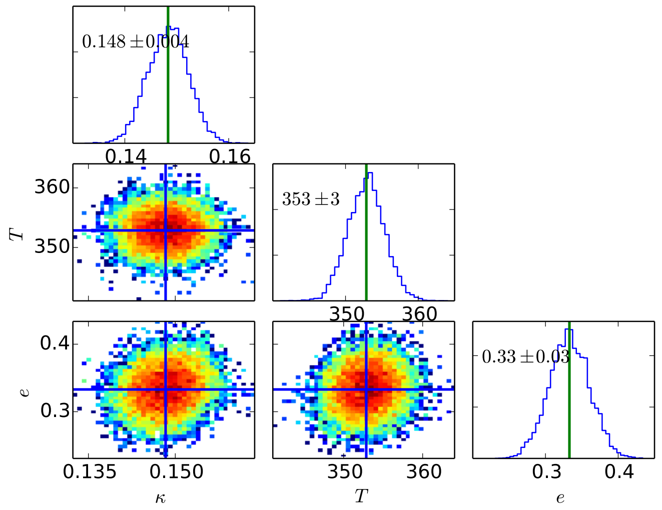
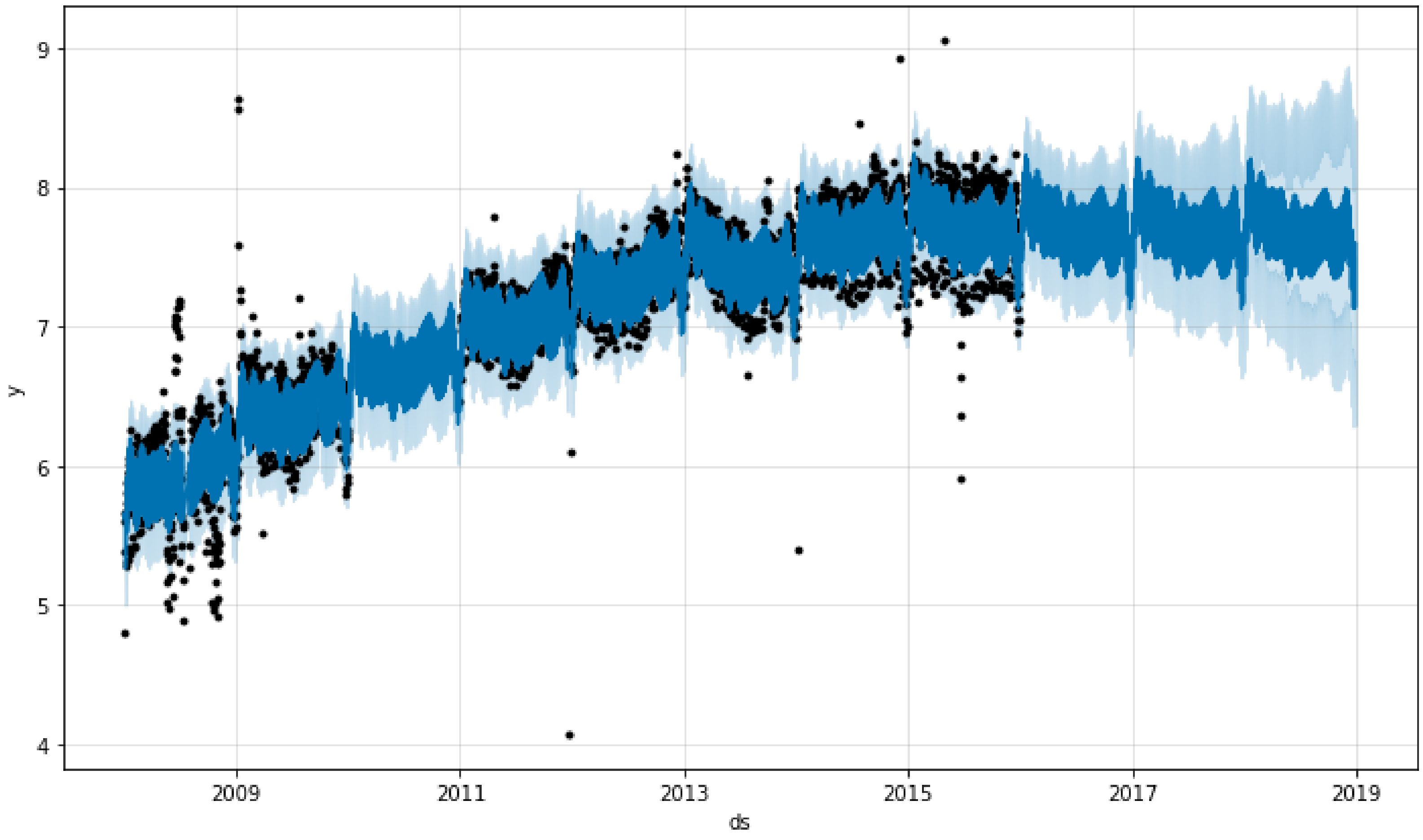


Figure 8

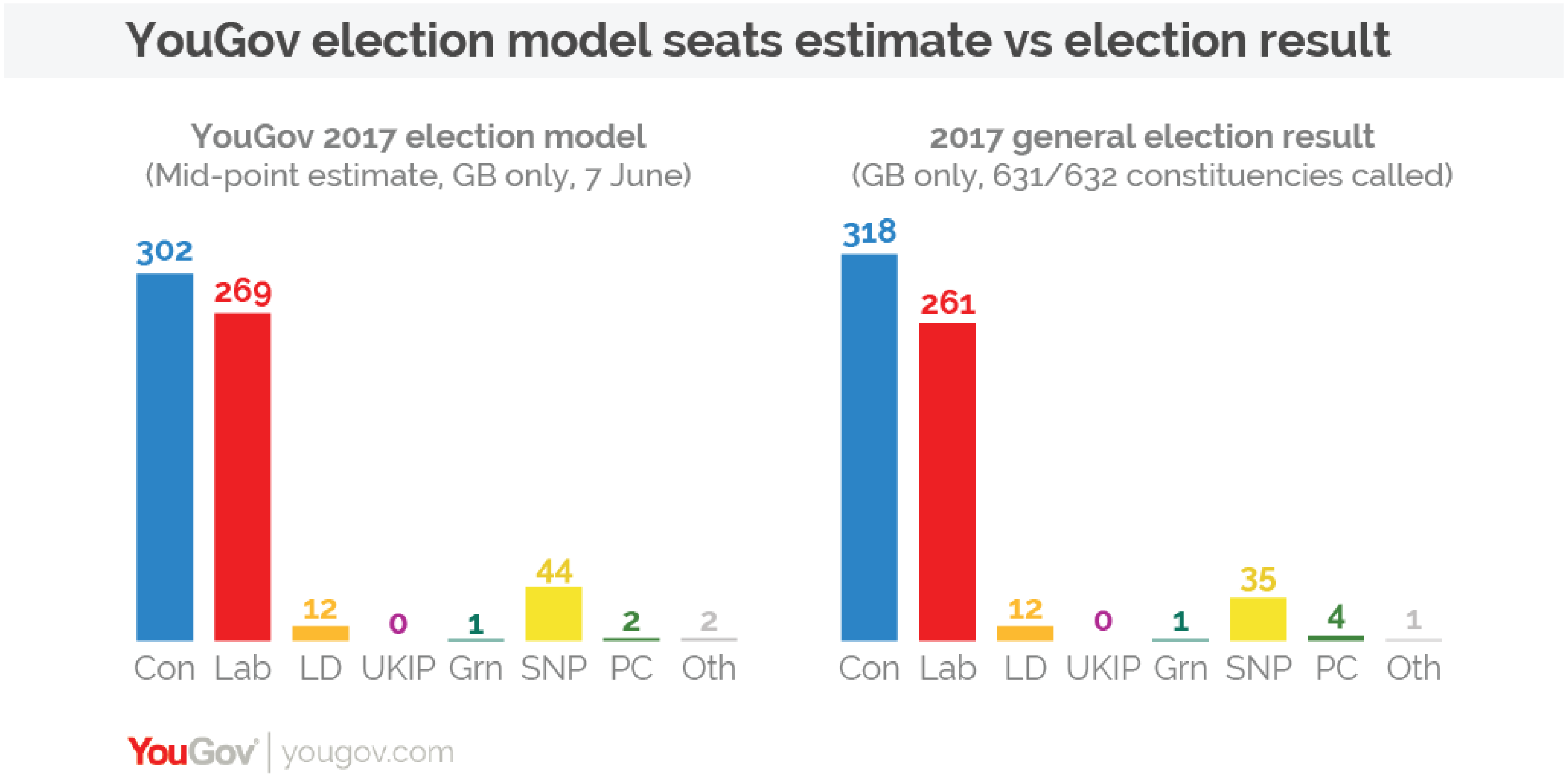
Left: Radial velocity as a function of time for a star in a binary system. The parameters of the binary system are listed on the top. The green line is the best fit solution obtained using an MCMC simulation. The red and the green curves are generated from Equation (84) and differ only in the eccentricity e. The plot shows that the shape of the radial velocity curve depends sensitively upon the eccentricity e of the orbit. Right: The posterior probability distribution of parameters obtained using the MCMC simulation.

Where and what for is Probabilistic programming used

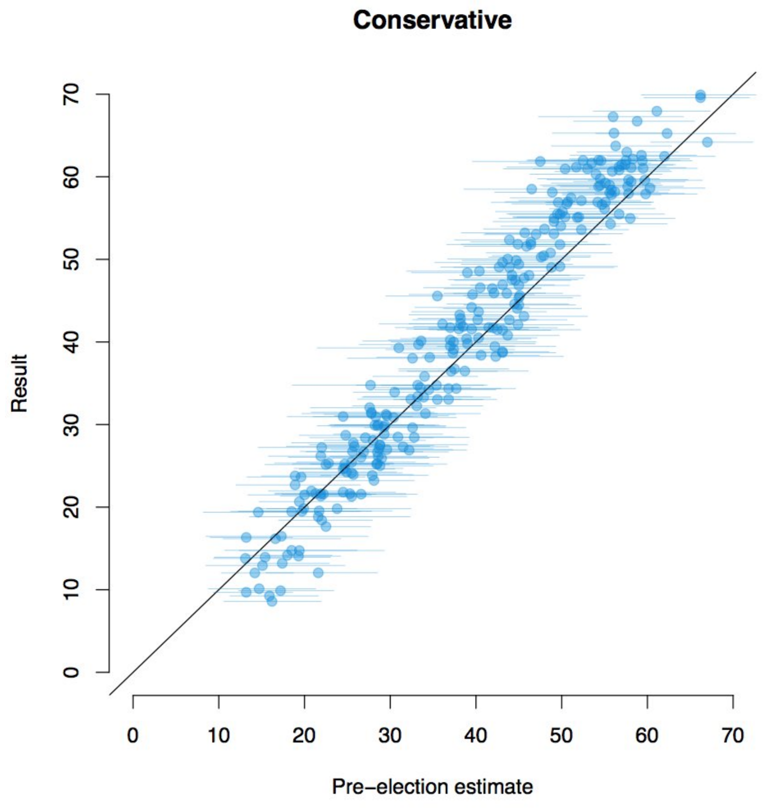
Forecasting at Facebook

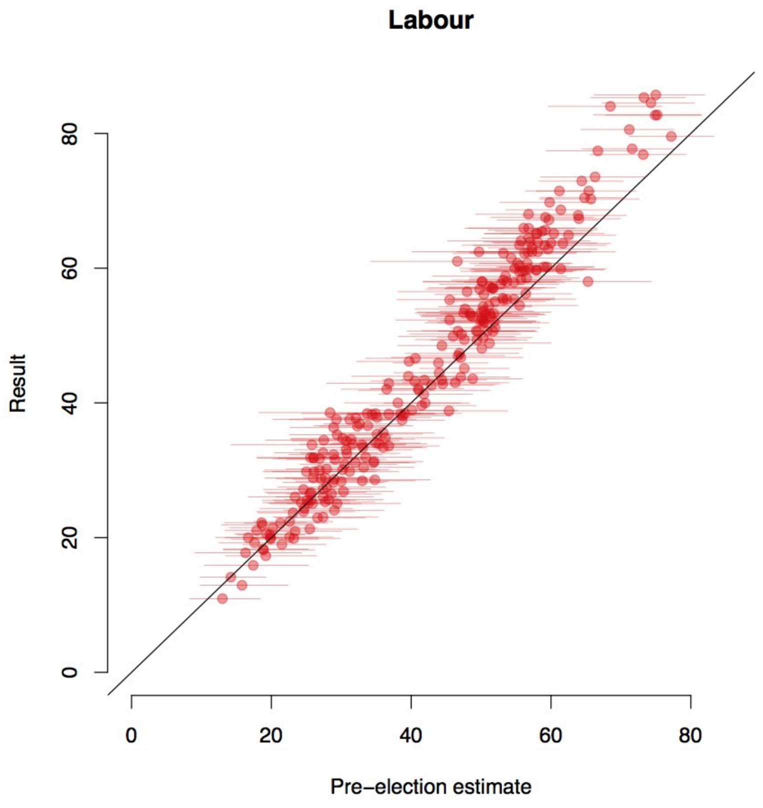


Predicting elections

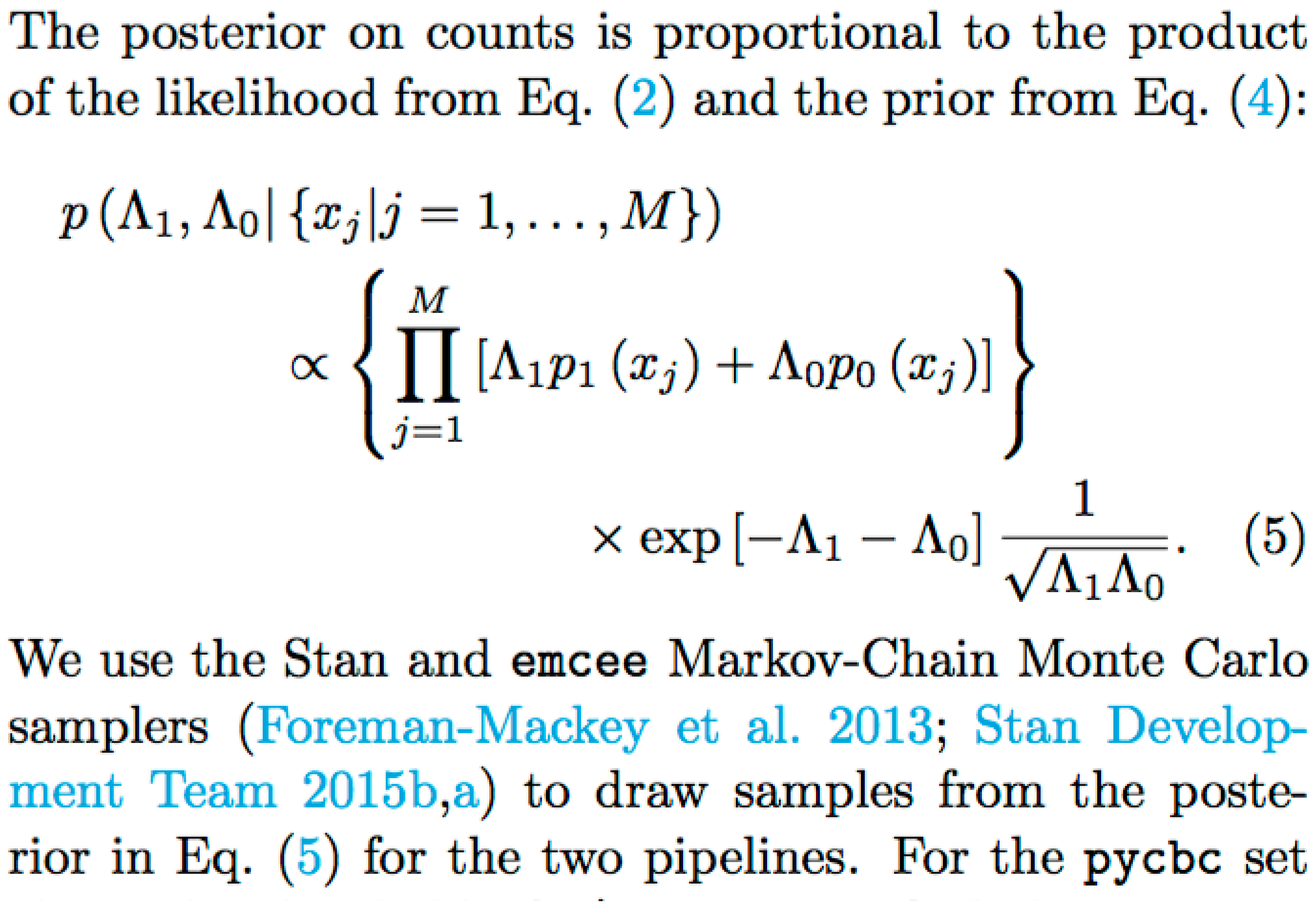


Predicting elections





Used in detecting graviational waves (LIGO)



The posterior on counts is proportional to the product of the likelihood from Eq. (2) and the prior from Eq. (4):

$$p\left(\Lambda_1, \Lambda_0 \middle| \left\{x_j | j = 1, \ldots, M\right\}\right) \propto \left\{\prod_{j=1}^{M} \left[\Lambda_1 p_1\left(x_j\right) + \Lambda_0 p_0\left(x_j\right)\right]\right\} \times \exp\left[-\Lambda_1 - \Lambda_0\right] \frac{1}{\sqrt{\Lambda_1 \Lambda_0}}. \quad (5)$$

We use the Stan and **emcee** Markov-Chain Monte Carlo samplers (Foreman-Mackey et al. 2013; Stan Development Team 2015b,a) to draw samples from the posterior in Eq. (5) for the two pipelines. For the **pycbc** set

PyMC3 testimonials

- "At Quantopian we use PyMC3 to track uncertainty in the performance of a trading algorithm." - Thomas Wiecki
- "We use PyMC3 to evaluate A/B test performance. Works great with very little code!" - Thomas Hunger, We Are Wizards
- "PyMC3 is used at VoiceBox Technologies to compare algorithm performances using Kruschke's BEST algorithm. More is in development."
- "Used in research code at Channel 4 for developing internal forecasting tools." - Peader Coyle
- "At Managed by Q, we use PyMC3 for all of our statistical modeling, including A/B test analysis, sales forecasting, and churn prediction." - Daniel Weitzenfeld
- "PyMC3 is my primary tool for statistical modeling at Salesforce. I use it to combine disparate sources of information and pretty much anywhere that quantifying uncertainty is important. For example, we build hierarchical models to evaluate varying effects in web experiments and then to build meta-analyses that quantify the expected returns of a subsequent experiment. We've also been experimenting with gaussian processes to model time series data for forecasting." - Eddie Landesberg. Manager, Data Scientist </small>

Definition of a probabilistic programming language

A programming language with

1. a mechanism to take draws from a probability distribution
2. a mechanism to condition on parts of program on data

Taxonomy of probabilistic programming languages

There are roughly two kinds of probabilistic programming languages.

1. Those where we statically know the number of latent variables
2. Those we have don't

Languages in category 2 are sometimes called universal probabilistic programming languages

Languages where we statically know the number of latent variables in the model

- Stan
- BUGS / JAGS
- PyMC3
- Infer.Net
- Hakaru
- Birch
- Nimble
- Augur
- Edward

Languages where we don't

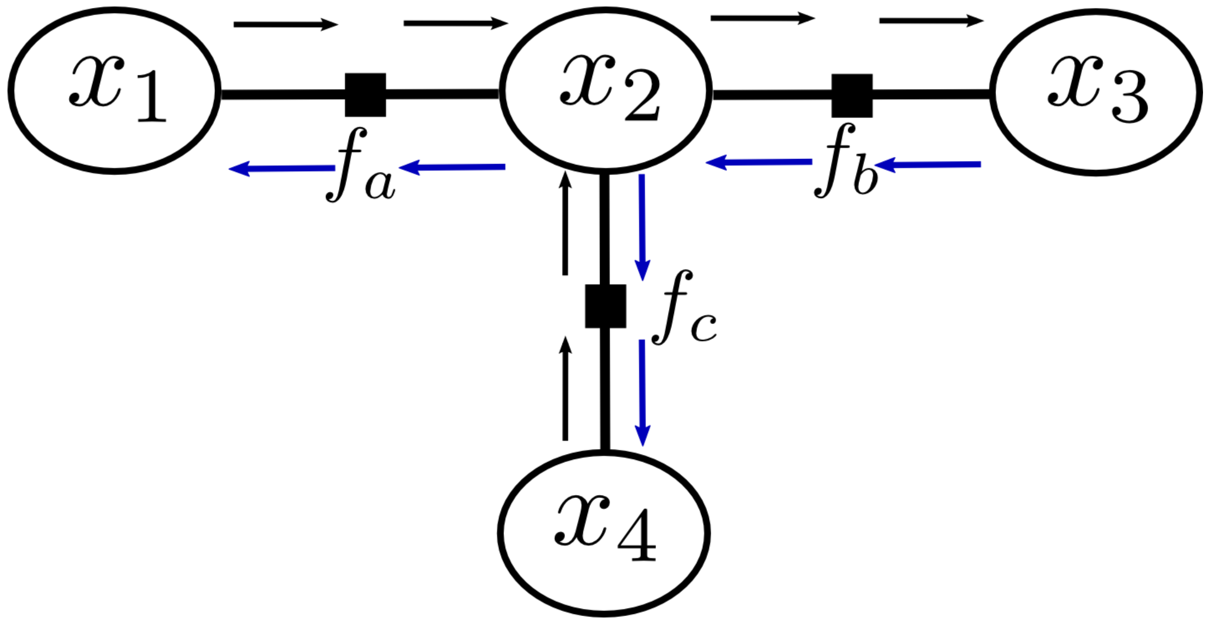
- Church / WebPPL
- Pyro
- MonadBayes
- Turing
- Anglican

Main algorithms in probabilistic programming systems

- Exact Inference
- Importance Sampling
- Trace Metropolis Hastings
- Sequential Monte Carlo
- Hamiltonian Monte Carlo
- Variational Inference

Exact inference

- Often intractable
- Still common in discrete domains
- Most effort goes into reusing computation



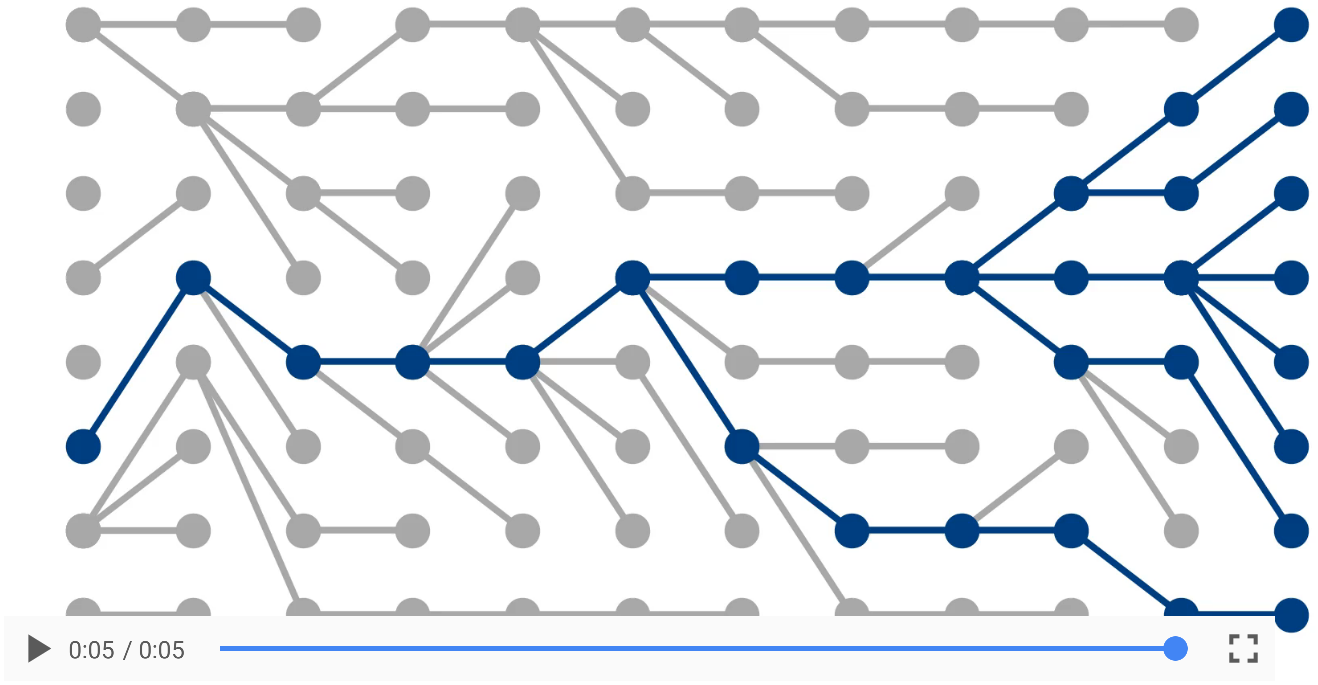
Importance Sampling

Main idea: generate samples from a program with the same return type. Then reweigh the samples based on the probability of the actual program generating these samples

Sequential Monte Carlo

Main idea: we maintain a population of samples as we run a program. Routinely, cull the population of low-probability samples and make extra copies of high-probability samples.

Sequential Monte Carlo



Trace Metropolis Hastings

Main idea: run the program to generate a trace. Modify the trace and rerun the program from the point of modification

Hamiltonian Monte Carlo and NUTS

Algorithm 1 Hamiltonian Monte Carlo

```
Given $\theta^0$, $\epsilon$, $L$, $\mathcal{L}, M$:
for $m = 1$ to $M$ do
    Sample $r^0 \sim \mathcal{N}(0, I)$.
    Set $\theta^m \leftarrow \theta^{m-1}, \tilde{\theta} \leftarrow \theta^{m-1}, \tilde{r} \leftarrow r^0$.
    for $i = 1$ to $L$ do
        Set $\tilde{\theta}, \tilde{r} \leftarrow$ Leapfrog($\tilde{\theta}, \tilde{r}, \epsilon$).
    end for
    With probability $\alpha = \min\left\{1, \frac{\exp\{\mathcal{L}(\tilde{\theta}) - \frac{1}{2}\tilde{r}\cdot\tilde{r}\}}{\exp\{\mathcal{L}(\theta^{m-1}) - \frac{1}{2}r^0\cdot r^0\}}\right\}$, set $\theta^m \leftarrow \tilde{\theta}$, $r^m \leftarrow -\tilde{r}$.
end for

function Leapfrog($\theta, r, \epsilon$)
Set $\tilde{r} \leftarrow r + (\epsilon/2)\nabla_\theta \mathcal{L}(\theta)$.
Set $\tilde{\theta} \leftarrow \theta + \epsilon \tilde{r}$.
Set $\tilde{r} \leftarrow \tilde{r} + (\epsilon/2)\nabla_\theta \mathcal{L}(\tilde{\theta})$.
return $\tilde{\theta}, \tilde{r}$.
```

where $\mathcal{L}$ is the logarithm of the joint density of the variables of interest θ (up to a normalizing constant) and $x \cdot y$ denotes the inner product of the vectors x and y. We can interpret this

Hamiltonian Monte Carlo and NUTS

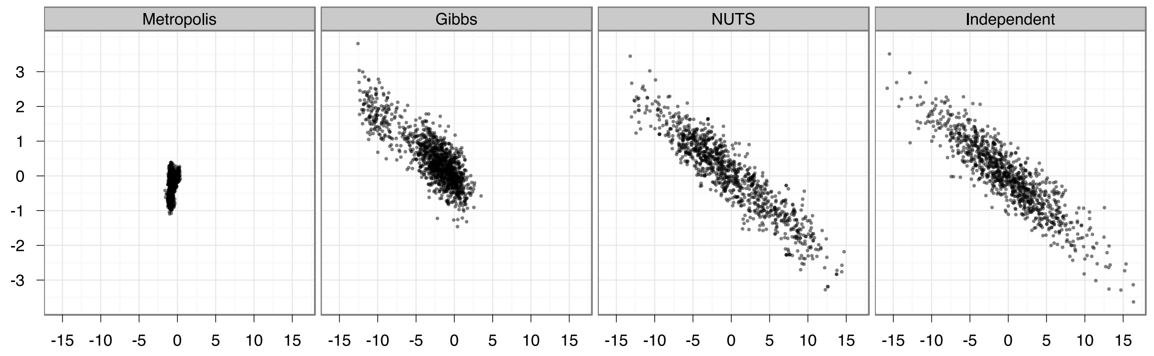


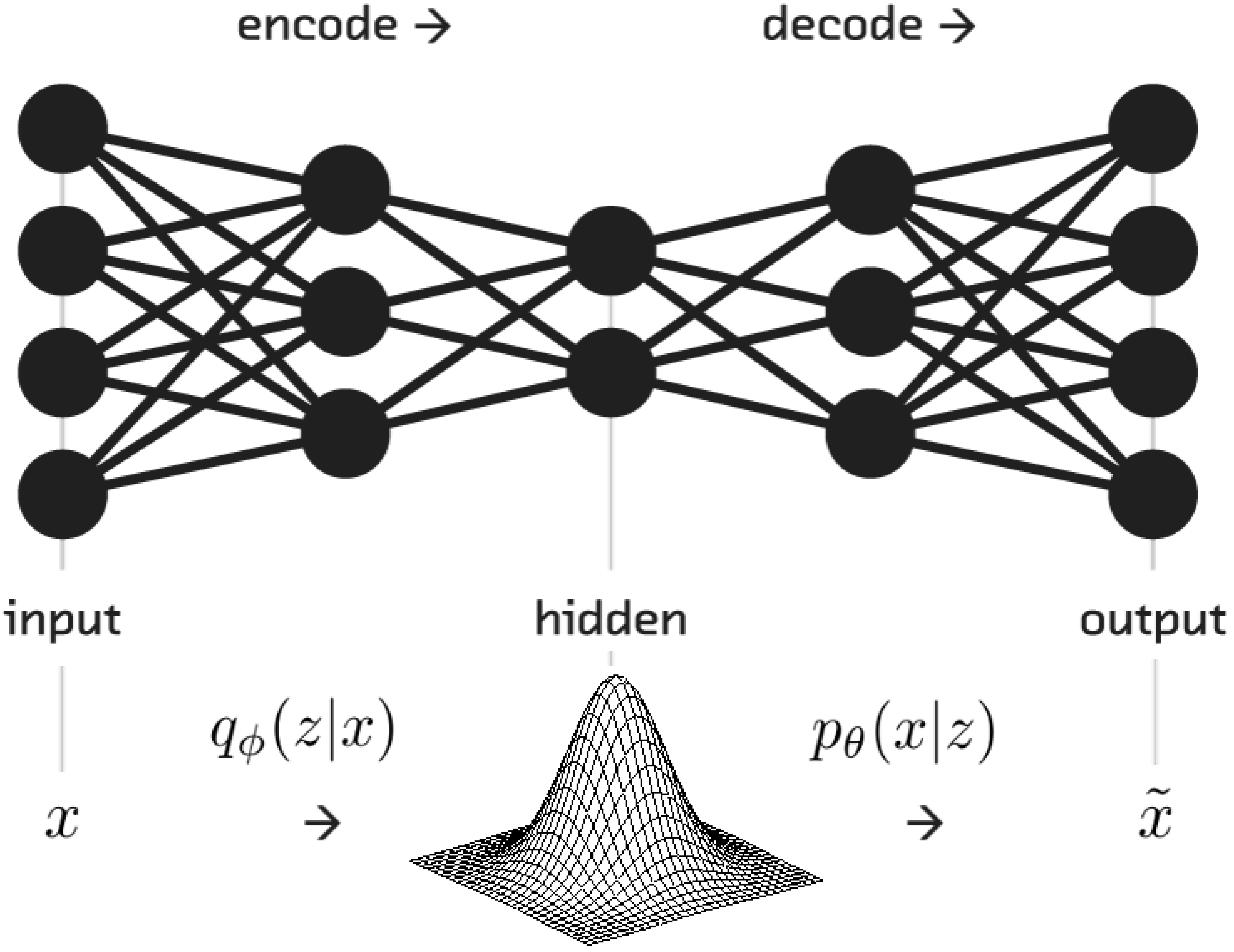
Figure 7: *Samples generated by random-walk Metropolis, Gibbs sampling, and NUTS. The plots compare 1,000 independent draws from a highly correlated 250-dimensional distribution (right) with 1,000,000 samples (thinned to 1,000 samples for display) generated by random-walk Metropolis (left), 1,000,000 samples (thinned to 1,000 samples for display) generated by Gibbs sampling (second from left), and 1,000 samples generated by NUTS (second from right). Only the first two dimensions are shown here.*

Variational inference

Approximate $p(z|x)$ with approximate distribution $q(z)$

This is done by minimizing the KL divergence

$$D_{\mathrm{KL}}(q(z) \| p(z|x)) = \int_{-\infty}^{\infty} q(z) \log \frac{q(z)}{p(z|x)} \, dz$$


encode →
decode →
input
hidden
output
$q_\phi(z|x)$
$p_\theta(x|z)$
x
→
→
$\tilde{x}$

Current challenges

- Hard to add inference algorithms to a system
- Posteriors are expensive to obtain for problems
- Lack of big applications
- Lack of tooling for diagnostics and debugging

Hard to add inference algorithms

- Most systems built to support a particular inference algorithm
- Hard to add new inference algorithms
- Systems end up over-specialized
- Newer systems suffer less from this problem (Hakaru, MonadBayes, Pyro, etc)

Lack of big applications

- Many of these languages are very expressive
- Few applications actually use this expressivity
- Larger probabilistic models don't often use PPLs

Debugging probabilistic programs

We need tools to help answer questions

- Are these samples from the true posterior?
- Is this a good model for my data?
- Is there a bug in my inference algorithm?
- Do I need to reparameterize my model?

Future directions

- Integration into deep learning frameworks
- Extending into Decision theory
- Causal Inference
- Extending into Bayesian Experimental Design

Integrating into deep learning frameworks

- Many inference algorithms can be posed as first-order optimization problems (require a gradient)
- Deep learning great for posing and computing gradients
- For example: Edward, Pyro, Probtorch

Integration into deep learning frameworks

Learning proposal distributions

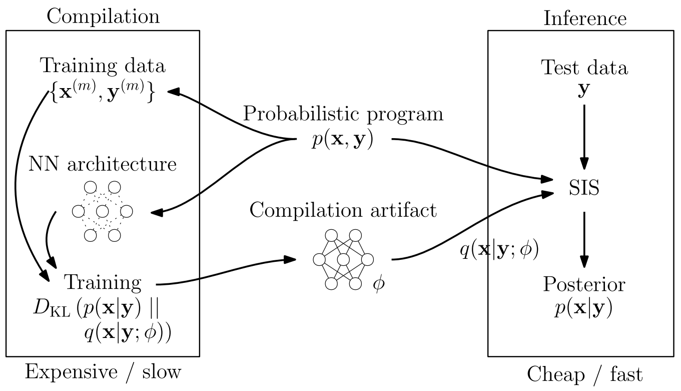

Compilation
Training data
$\{\mathbf{x}^{(m)}, \mathbf{y}^{(m)}\}$
NN architecture
Training
$D_{\mathrm{KL}}\left(p(\mathbf{x}|\mathbf{y}) \,||\, q(\mathbf{x}|\mathbf{y};\phi)\right)$
Expensive / slow
Probabilistic program
$p(\mathbf{x},\mathbf{y})$
Compilation artifact
ϕ
$q(\mathbf{x}|\mathbf{y};\phi)$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

Figure 7.1: An outline of an approach to inference compilation for amortized inference for probabilistic programs. Re-used with permission from [Le et al., 2017a]

Experimental Design

- We have some model of our data, we would like to collect data in a way to maximally improve our model

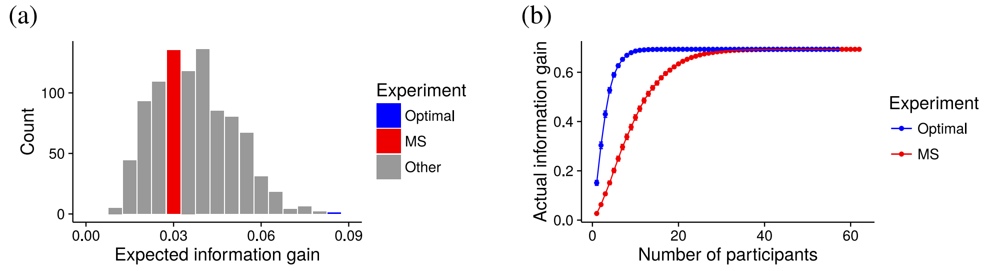


Figure 4: (a) Distribution of expected information gain for all possible category learning experiments for a single participant. MS has low expected information gain. (b) Actual information gain versus number of experimental participants included in analysis (error bars are 95% bootstrapped confidence intervals). MS requires three times as many participants to achieve maximum actual information gain.

Further references

- Stan reference (https://github.com/stan-dev/stan/releases/download/v2.17.0/stan-reference-2.17.0.pdf)
- Stan case studies (http://mc-stan.org/users/documentation/case-studies.html)
- Practical Probabilistic Programming (https://www.manning.com/books/practical-probabilistic-programming)

Main takeaways

- Probabilistic programming is already in use
- Inference algorithms are not blackbox yet
- Big applications still are needed
- Statistical correctness and diagnostics still developing