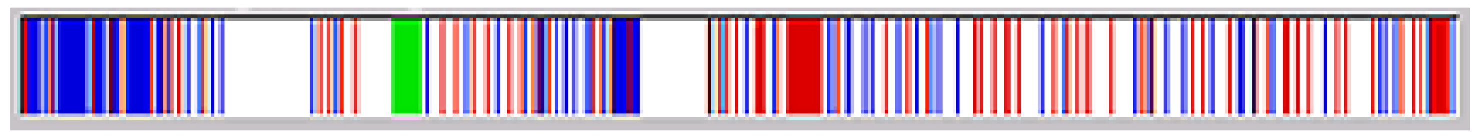
A Variational Inference Toolbox

Is it time for a community-driven VI toolbox?

Yarin Gal (**yg279@cam.ac.uk**), University of Cambridge

UNIVERSITY OF CAMBRIDGE

Variational inference is Fragmented



- Most advances in Deep Learning from the last few years are due to central code repositories exploiting model compositionality.
  - Vast number of published papers can be built from simpler building blocks, becoming themselves higher level building blocks,
  - Example: Simple deep networks → Recurrent Neural Networks → Neural Turing Machine → The Neural Queue.
- In contrast, VI has no central repository, or even an agreed-upon framework.
- Instead we often re-implement existing work in VI, wasting weeks at a time.
- Is it time for a community driven VI toolbox?

The time is Right

- Relying on recent advances in stochastic inference and sampling based variational inference (replacing integration with stochastic optimisation),
- Taking advantage of frameworks developed within the deep learning community: Theano, Torch, TensorFlow, etc.
- Allows us to design simple VI building blocks to compose together.
- Allows us to combine deep learning and VI seamlessly.

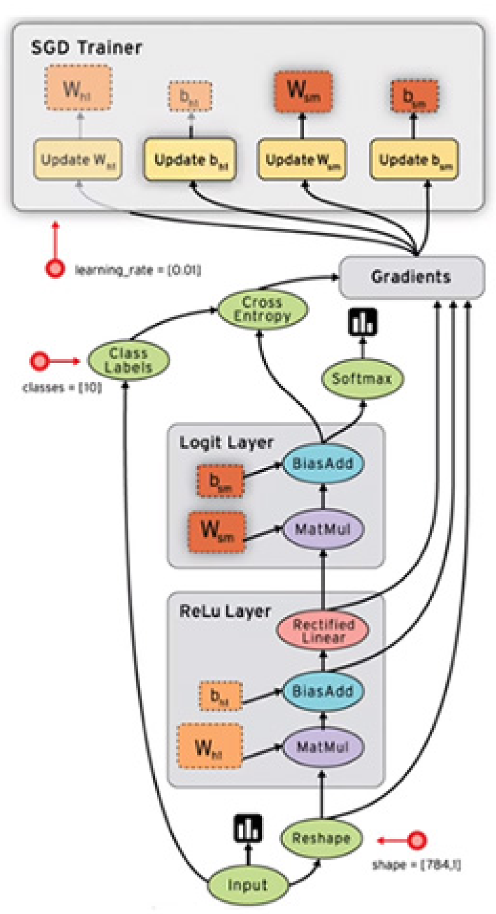


Image sources: Wikimedia, `tensorflow.org`, `deeplearning.net`

Example: symbolic differentiation (Theano).

- Builds a graph of symbolic variables and operations on these,
- automatically optimises structure to make computations efficient,
- propagates chain rule throughout the graph.

Vanilla variational inference

- Given data $\mathbf{X}$ design initial probabilistic model,

$$p(x^*|\mathbf{X}) = \int p(x^*|\omega)p(\omega|\mathbf{X})\mathrm{d}\omega$$

with some latent random variable ω. The posterior $p(\omega|\mathbf{X})$ is intractable.

- Choose an approximating variational distribution $q_\theta(\omega)$ matching posterior properties.
- Evaluate divergence between approximating posterior and true posterior obtaining a lower bound,

$$\mathcal{L}(\theta) := \int q_\theta(\omega)\log p(\mathbf{X}|\omega)\mathrm{d}\omega - \mathrm{KL}(q_\theta(\omega)||p(\omega)).$$

And then...

- Spend weeks calculating and implementing derivatives, testing with finite differences, and optimising computations for performance and numerical stability.

We could do better.

If we had modular VI Building Blocks...

- Replace the last two steps in vanilla VI.
- Collect common VI building blocks into a central repository.
- Write down generative model in a symbolic language with existing VI blocks (creating new ones as necessary),

$$\begin{aligned}&\texttt{var}\ \omega;\\ &f(\omega) = \text{Block}_1(\text{Block}_2(\text{Block}_3(\omega)))\\ &\mathbf{X} = f(\omega);\end{aligned}$$

- Simulate T samples from the approximate posterior and propagate them down the generative model (forward pass),

$$\begin{aligned}\omega_t &\sim q_\theta(\omega);\\ \mathbf{X}_t &= f(\omega_t);\end{aligned}$$

- Evaluate the objective with the output of the generative model,

$$\mathcal{L}(\theta) \approx \frac{1}{T}\sum_{t=1}^{T}\log p(\mathbf{X}_t) - \mathrm{KL}(q_\theta(\omega)||p(\omega)).$$

- Symbolically differentiate the objective:
  - evaluate derivatives with the same samples
  - obtaining a noisy but unbiased gradient estimate
  - this is a backward pass.
- Optimise with a stochastic optimiser.

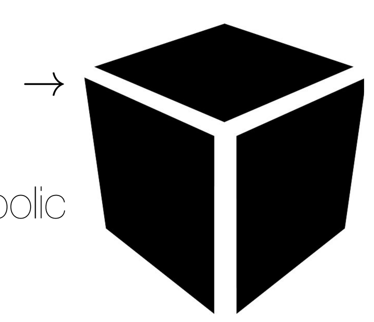
Example

```python
import theano.tensor as T
m = T.dmatrix('m') # ... and other variational parameters
X = m + s * randn(N, Q) # these are the generative model's variables
U = mu + L.dot(randn(M, K))
Kmm = RBF(sf2, l, Z)
Kmn = RBF(sf2, l, Z, X)
Knn = RBFnn(sf2, l, X)
KmmInv = T.matrix_inverse(Kmm)
A = KmmInv.dot(Kmn)
B = Knn - T.sum(Kmn * KmmInv.dot(Kmn), 0)
F = A.T.dot(U)+B[:,None]**0.5 * randn(N,K)
S = T.nnet.softmax(F) # model's output - Softmax probabilities
KL_U, KL_X = get_KL_U(), get_KL_X() # these are the KL terms
LS = T.sum(T.log(T.sum(Y * S, 1)))
            - KL_U - KL_X # and this is the lower bound we optimise
LS_func = theano.function(['''inputs'''], # compile the model
            LS)
dLS_dm = theano.function(['''inputs'''], # and the derivatives
            T.grad(LS, m))
# ... and optimise LS with RMS-PROP
```

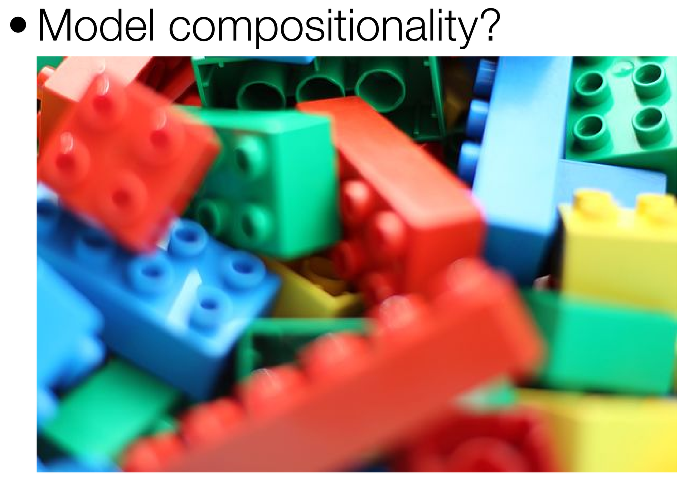
Example Python code using the new pipeline. Here, `m`, `s`, `mu`, and `L` are the variational parameters, and the generative model `S` (the probabilities of the discrete variables) is a function of latents `X`, `U`, and `F`. Our objective is `LS`.

Emerging Challenges

- Existing tools lack...
  - good support for many operations used in VI (matrix inverses, matrix determinants, etc.).
  - "tricks-of-the-trade" used in VI to avoid problems of numerical instability and large matrix multiplications.
  - Would these lead to more efficient models, smaller, readable, and extendible code-bases?
- Black-box variance reduction
  - Variance reduction forces model re-parametrisation → complicated inference and code.
  - Apply variance reduction automatically to the symbolic graph?



- Model compositionality?



  - Speed-up the innovation cycle allowing fast-evolving model complexity,
  - What are the basic VI building blocks?
  - Recent work casting deep learning tools as VI in Bayesian neural networks (see other poster) – already have many building blocks to start with!

A unified framework will make VI accessible to larger audiences.

Full paper: "Rapid Prototyping of Probabilistic Models: Emerging Challenges in Variational Inference". Photos taken from Wikimedia unless specified otherwise.