Symbolic Differentiation for Rapid Model Prototyping in Machine Learning and Data Analysis — a Hands-on Tutorial

Yarin Gal

yg279@cam.ac.uk

November 13th, 2014

A TALK IN TWO ACTS, based on the online tutorial
deeplearning.net/software/theano/tutorial
The Theory

Theano in practice

Two Example Models: Logistic Regression and a Deep Net

Rapid Prototyping of Probabilistic Models with SVI (time permitting)
Some Theory
Symbolic differentiation is *not* automatic differentiation, nor numerical differentiation [source: Wikipedia].

Symbolic *computation* is a scientific area that refers to the study and development of algorithms and software for manipulating mathematical expressions and other mathematical objects.
What’s Theano?

- Theano was the priestess of Athena in Troy [source: Wikipedia].
- It is also a **Python package for symbolic differentiation**.
- Open source project primarily developed at the University of Montreal.
- Symbolic equations compiled to run efficiently on CPU and GPU.
- Computations are expressed using a NumPy-like syntax:
  - `numpy.exp()` – `theano.tensor.exp()`
  - `numpy.sum()` – `theano.tensor.sum()`
How does Theano work?

Internally, Theano builds a graph structure composed of:

- interconnected variable nodes (red),
- operator (op) nodes (green),
- and “apply” nodes (blue, representing the application of an op to some variables)

```python
1 import theano.tensor as T
2 x = T.dmatrix('x')
3 y = T.dmatrix('y')
4 z = x + y
```
Theano basics – differentiation

Computing automatic differentiation is simple with the graph structure.

- The only thing tensor.grad() has to do is to traverse the graph from the outputs back towards the inputs.
- Gradients are composed using the chain rule.

**Code for derivatives of $x^2$:**

```python
1 x = T.scalar('x')
2 f = x**2
3 df_dx = T.grad(f, [x])  # results in 2x
```
When compiling a Theano graph, graph optimisation...

- Improves the way the computation is carried out,
- Replaces certain patterns in the graph with faster or more stable patterns that produce the same results,
- And detects identical sub-graphs and ensures that the same values are not computed twice (*mostly*).

For example, one optimisation is to replace the pattern $\frac{xy}{y}$ by $x$. 
The Practice
>>> `import` `theano.tensor` `as` `T`
>>> `from` `theano` `import` `function`
>>> `x` = `T.dscalar`(`'x'`)  
>>> `y` = `T.dscalar`(`'y'`)  
>>> `z` = `x + y`  # `same graph as before`

>>> `f` = `function`([`x`, `y`], `z`)  # compiling the graph  
# the function inputs are `x` and `y`, its output is `z`
>>> `f`(2, 3)  # evaluating the function on integers
`array`(5.0)
>>> `f`(16.3, 12.1)  # ...and on floats
`array`(28.4)

>>> `z.eval`({`x` : 16.3, `y` : 12.1})
`array`(28.4)  # a quick way to debug the graph

>>> `from` `theano` `import` `pp`
>>> `print` `pp`(`z`)  # print the graph
`x + y`
If you don’t have Theano installed, you can SSH into one of the following computers and use the Python console:

- riemann
- dirichlet
- bernoulli
- grothendieck
- robbins
- explorer

Syntax (from an external network):

1. ssh [user name]@gate.eng.cam.ac.uk
2. ssh [computer name]
3. python
4. >>> import theano
5. >>> import theano.tensor as T

Exercise files are on http://goo.gl/r5uwGI
1. Type and run the following code:

```python
import theano
import theano.tensor as T
a = T.vector()  # declare variable
out = a + a**10  # build symbolic expression
f = theano.function([a], out)  # compile function
print f([0, 1, 2])  # prints ‘array([0, 2, 1026])’
```

2. Modify the code to compute $a^2 + 2ab + b^2$ element-wise.
import theano
import theano.tensor as T
a = T.vector()  # declare variable
b = T.vector()  # declare variable
out = a**2 + 2*a*b + b**2  # build symbolic expression
f = theano.function([a, b], out)  # compile function
print f([1, 2], [4, 5])  # prints [ 25. 49.]
Implement the *Logistic Function*:

\[ s(x) = \frac{1}{1 + e^{-x}} \]

(adapt your NumPy implementation, you will need to replace “np” with “T”; this will be used later in Logistic regression)
```python
>>> x = T.dmatrix('x')
>>> s = 1 / (1 + T.exp(-x))
>>> logistic = theano.function([x], s)
>>> logistic([[0, 1], [-1, -2]])
array([[ 0.5 , 0.73105858],
       [ 0.26894142, 0.11920292]])
```

Note that the operations are performed element-wise.
We can compute the elementwise *difference*, *absolute difference*, and *squared difference* between two matrices $a$ and $b$ at the same time.

```python
>>> a, b = T.dmatrices('a', 'b')
>>> diff = a - b
>>> abs_diff = abs(diff)
>>> diff_squared = diff**2
>>> f = function([a, b], [diff, abs_diff, diff_squared])
```
Theano basics – shared variables

Shared variables allow for functions with internal states.

- hybrid symbolic and non-symbolic variables,
- value may be shared between multiple functions,
- used in symbolic expressions but also have an internal value.

The value can be accessed and modified by the .get_value() and .set_value() methods.

Accumulator

The state is initialized to zero. Then, on each function call, the state is incremented by the function’s argument.

```python
>>> state = theano.shared(0)
>>> inc = T.iscalar('inc')
>>> accumulator = theano.function([inc], state,
                                  updates=[(state, state+inc)])
```
Updates can be supplied with a list of pairs of the form (shared-variable, new expression),

Whenever function runs, it replaces the value of each shared variable with the corresponding expression’s result at the end.

In the example above, the accumulator replaces state’s value with the sum of state and the increment amount.

```
>>> state.get_value()
array(0)

>>> accumulator(1)
array(0)

>>> state.get_value()
array(1)

>>> accumulator(300)
array(1)

>>> state.get_value()
array(301)
```
Two Example Models: Logistic Regression and a Deep Net
Logistic regression is a probabilistic linear classifier.

It is parametrised by a weight matrix $W$ and a bias vector $b$.

The probability that an input vector $x$ is classified as 1 can be written as:

$$P(Y = 1|x, W, b) = \frac{1}{1 + e^{-(Wx + b)}} = s(Wx + b)$$

The model’s prediction $y_{pred}$ is the class whose probability is maximal, specifically for every $x$:

$$y_{pred} = 1(P(Y = 1|x, W, b) > 0.5)$$

And the optimisation objective (negative log-likelihood) is

$$-y \log(s(Wx + b)) - (1 - y) \log(1 - s(Wx + b))$$

(you can put a Gaussian prior over $W$ if you so desire.)

Using the Logistic Function, implement Logistic Regression.
...  
2 x = T.matrix("x")  
3 y = T.vector("y")  
4 w = theano.shared(np.random.randn(784), name="w")  
5 b = theano.shared(0., name="b")  

# Construct Theano expression graph  
prediction, obj, gw, gb # Implement me!  

# Compile  
train = theano.function(inputs=[x,y],  
outputs=[prediction, obj],  
updates=((w, w - 0.1 * gw), (b, b - 0.1 * gb)))  
predict = theano.function(inputs=[x], outputs=prediction)  

# Train  
for i in range(training_steps):  
    pred, err = train(D[0], D[1])  
...
... 

# Construct Theano expression graph

# Probability that target = 1
p_1 = 1 / (1 + T.exp(-T.dot(x, w) - b))

# The prediction thresholded
prediction = p_1 > 0.5

# Cross-entropy loss function
obj = -y * T.log(p_1) - (1-y) * T.log(1-p_1)

# The cost to minimize
cost = obj.mean() + 0.01 * (w ** 2).sum()

# Compute the gradient of the cost

...
Implement an MLP, following section *Example: MLP* in
http://nbviewer.ipython.org/github/craffel/theano-tutorial/blob/master/Theano%20Tutorial.ipynb#example-mlp
```python
class Layer(object):
    def __init__(self, W_init, b_init, activation):
        n_output, n_input = W_init.shape
        self.W = theano.shared(value=W_init.astype(theano.config.floatX),
                                name='W',
                                borrow=True)
        self.b = theano.shared(value=b_init.reshape(-1, 1).astype(theano.config.floatX),
                                name='b',
                                borrow=True,
                                broadcastable=(False, True))
        self.activation = activation
        self.params = [self.W, self.b]

    def output(self, x):
        lin_output = T.dot(self.W, x) + self.b
        return (lin_output if self.activation is None else self.activation(lin_output))
```
```python
class MLP(object):
    def __init__(self, W_init, b_init, activations):
        self.layers = []
        for W, b, activation in zip(W_init, b_init, activations):
            self.layers.append(Layer(W, b, activation))

        self.params = []
        for layer in self.layers:
            self.params += layer.params

    def output(self, x):
        for layer in self.layers:
            x = layer.output(x)
        return x

    def squared_error(self, x, y):
        return T.sum((self.output(x) - y)**2)
```
```python
def gradient_updates_momentum(cost, params, learning_rate, momentum):
    updates = []
    for param in params:
        param_update = theano.shared(param.get_value() * 0.,
                                       broadcastable=param.broadcastable)
        updates.append(((param, 
                         param - learning_rate * param_update))
                        updates.append(((param_update, momentum * param_update
                                         + (1. - momentum) * T.grad(cost, param)))
    return updates
```
Rapid Prototyping of Probabilistic Models with Stochastic Variational Inference
In **data analysis** we often have to develop new models

- This can be a lengthy process
  - We need to derive appropriate inference
  - Often cumbersome implementation which changes regularly

**Rapid prototyping** is used to answer similar problems in manufacturing

- “Quick fabrication of scale models of a physical part”
- Probabilistic programming can be used for rapid prototyping in machine learning
In **data analysis** we often have to develop new models

This can be a lengthy process
- We need to derive appropriate inference
- Often cumbersome implementation which changes regularly

**Rapid prototyping** is used to answer similar problems in manufacturing
- “Quick fabrication of scale models of a physical part”
- Probabilistic programming can be used for rapid prototyping in machine learning
Rapid Prototyping

- In **data analysis** we often have to develop new models
- This can be a lengthy process
  - We need to derive appropriate inference
  - Often cumbersome implementation which changes regularly
- **Rapid prototyping** is used to answer similar problems in manufacturing
  - “Quick fabrication of scale models of a physical part”
  - Probabilistic programming can be used for rapid prototyping in machine learning
In *data analysis* we often have to develop new models. This can be a lengthy process. We need to derive appropriate inference and often cumbersome implementation which changes regularly.

**Rapid prototyping** is used to answer similar problems in manufacturing. “Quick fabrication of scale models of a physical part”

Probabilistic programming can be used for rapid prototyping in machine learning. Stochastic Variational Inference (SVI) can be used for rapid prototyping as well, with several advantages over probabilistic programming.
SVI is not usually considered as means of speeding-up development.

But this new inference technique allows us to simplify the derivations for a large class of models. With this we can take advantage of effective symbolic differentiation. Models are often mathematically too cumbersome otherwise.

Similar principles have been used for rapid model prototyping in deep learning for NLP for quite some time [Socher, Ng, and Manning 2010, 2011, 2012].
SVI is simply **variational inference** used with **noisy gradients** – we thus replace the optimisation with stochastic optimisation.

**Variational inference**
- We approximate the posterior of the latent variables with distributions from a tractable family (\(q(X)\) for example)

---

**Example model: \(X \rightarrow Y\)**

\[
\log P(Y) \geq \int q(X) \log \frac{P(Y|X)P(X)}{q(X)} = E_q[\log P(Y|X)] - KL(q||P)
\]
What is SVI?

- Stochastic variational inference
  - Often used to speed-up inference using **mini-batches**
    \[
    \log P(Y) \geq \frac{N}{|S|} \sum_{i \in S} E_q[\log P(Y_i|X_i)] - KL(q||P)
    \]
    summing over random subsets of the data points
  - But can also be used to **approximate integrals** through Monte Carlo integration [Kingma and Welling 2014, Rezende et al. 2014, Titsias and Lazaro-Gredilla 2014]
    \[
    E_q[\log P(Y|X)] \approx \frac{1}{K} \sum_{i=1}^{K} \log P(Y|X_i), \ X_i \sim q(X)
    \]
    summing over samples from the approximating distribution
  - Optimising these objectives relies on non-deterministic gradients
What is SVI?

- Stochastic variational inference
  - Often used to speed-up inference using **mini-batches**

\[
\log P(Y) \geq \frac{N}{|S|} \sum_{i \in S} E_q[\log P(Y_i|X_i)] - KL(q||P)
\]

summing over random subsets of the data points

- But can also be used to **approximate integrals** through Monte Carlo integration [Kingma and Welling 2014, Rezende et al. 2014, Titsias and Lazaro-Gredilla 2014]

\[
E_q[\log P(Y|X)] \approx \frac{1}{K} \sum_{i=1}^{K} \log P(Y|X_i), \ X_i \sim q(X)
\]

summing over samples from the approximating distribution

- Optimising these objectives relies on non-deterministic gradients
What is SVI?

- Stochastic variational inference
  - Often used to speed-up inference using **mini-batches**
    \[
    \log P(Y) \geq \frac{N}{|S|} \sum_{i \in S} E_q[\log P(Y_i|X_i)] - KL(q||P)
    \]
    summing over random subsets of the data points
  - But can also be used to **approximate integrals** through Monte Carlo integration [Kingma and Welling 2014, Rezende et al. 2014, Titsias and Lazaro-Gredilla 2014]
    \[
    E_q[\log P(Y|X)] \approx \frac{1}{K} \sum_{i=1}^{K} \log P(Y|X_i), \ X_i \sim q(X)
    \]
    summing over samples from the approximating distribution
  - Optimising these objectives relies on non-deterministic gradients
Stochastic optimisation

- Using gradient descent with noisy gradients and decreasing learning-rates, we are guaranteed to converge to an optimum

\[ \theta_{t+1} = \theta_t + \alpha f'(\theta_t) \]

- Learning-rates (\(\alpha\)) are hard to tune...
  - Use learning-rate free optimisation (again, from deep learning)

  - AdaGrad [Duchi et. al 2011], AdaDelta [Zeiler 2012]

  - RMSPROP [Tieleman and Hinton 2012, Lecture 6.5, COURSERA: Neural Networks for Machine Learning]

\[ \theta_{t+1} = \theta_t + \frac{\alpha}{\sqrt{r_t}} f'(\theta_t); \quad r_t = (1 - \gamma) f'(\theta)^2 + \gamma r_{t-1} \]

  and increase \(\alpha\) times \(1 + \epsilon\) if the last two grads' directions agree

- These have been compared to each other and others \textit{empirically} in a variety of settings in [Schaul 2014]
Stochastic optimisation

- Using gradient descent with noisy gradients and decreasing learning-rates, we are guaranteed to converge to an optimum

\[ \theta_{t+1} = \theta_t + \alpha f'(\theta_t) \]

- Learning-rates (\(\alpha\)) are hard to tune...
  - Use learning-rate free optimisation (again, from deep learning)
    - AdaGrad [Duchi et. al 2011], AdaDelta [Zeiler 2012]
    - RMSPROP [Tieleman and Hinton 2012, Lecture 6.5, COURSERA: Neural Networks for Machine Learning]

\[ \theta_{t+1} = \theta_t + \frac{\alpha}{\sqrt{r_t}} f'(\theta_t); \quad r_t = (1 - \gamma) f'(\theta)^2 + \gamma r_{t-1} \]

and increase \(\alpha\) times \(1 + \epsilon\) if the last two grads’ directions agree

- These have been compared to each other and others empirically in a variety of settings in [Schaul 2014]
Using gradient descent with noisy gradients and decreasing learning-rates, we are guaranteed to converge to an optimum

\[ \theta_{t+1} = \theta_t + \alpha f'(\theta_t) \]

Learning-rates (\( \alpha \)) are hard to tune...

- Use learning-rate free optimisation (again, from deep learning)

- AdaGrad [Duchi et. al 2011], AdaDelta [Zeiler 2012]

- RMSPROP [Tieleman and Hinton 2012, Lecture 6.5, COURSERA: Neural Networks for Machine Learning]

\[ \theta_{t+1} = \theta_t + \frac{\alpha}{\sqrt{r_t}} f'(\theta_t); \ r_t = (1 - \gamma) f'(\theta)^2 + \gamma r_{t-1} \]

and increase \( \alpha \) times \( 1 + \epsilon \) if the last twograds’ directions agree

These have been compared to each other and others empirically in a variety of settings in [Schaul 2014]
Using gradient descent with noisy gradients and decreasing learning-rates, we are guaranteed to converge to an optimum

\[ \theta_{t+1} = \theta_t + \alpha f'(\theta_t) \]

Learning-rates (\(\alpha\)) are hard to tune...
- Use learning-rate free optimisation (again, from deep learning)
- AdaGrad [Duchi et. al 2011], AdaDelta [Zeiler 2012]
- RMSPROP [Tieleman and Hinton 2012, Lecture 6.5, COURSERA: Neural Networks for Machine Learning]

\[ \theta_{t+1} = \theta_t + \frac{\alpha}{\sqrt{r_t}} f'(\theta_t); \quad r_t = (1 - \gamma) f'(\theta)^2 + \gamma r_{t-1} \]

and increase \(\alpha\) times \(1 + \epsilon\) if the last two grads’ directions agree

These have been compared to each other and others empirically in a variety of settings in [Schaul 2014]
Stochastic optimisation

- Using gradient descent with noisy gradients and decreasing learning-rates, we are guaranteed to converge to an optimum

\[ \theta_{t+1} = \theta_t + \alpha f'(\theta_t) \]

- Learning-rates (\(\alpha\)) are hard to tune...
  - Use learning-rate free optimisation (again, from deep learning)
    - AdaGrad [Duchi et. al 2011], AdaDelta [Zeiler 2012]
    - RMSPROP [Tieleman and Hinton 2012, Lecture 6.5, COURSERA: Neural Networks for Machine Learning]

\[ \theta_{t+1} = \theta_t + \frac{\alpha}{\sqrt{r_t}} f'(\theta_t); \quad r_t = (1 - \gamma) f'(\theta)^2 + \gamma r_{t-1} \]

and increase \(\alpha\) times \(1 + \epsilon\) if the last two grads’ directions agree

- These have been compared to each other and others empirically in a variety of settings in [Schaul 2014]
With **Monte Carlo integration** we can greatly simplify model and inference description.

**Example model: X → Y**

**Lower bound:**

1. Simulate $X_i \sim q(X)$ for $i \leq K$
2. Evaluate $P(Y|X_i)$
3. Return $\frac{1}{K} \sum_{i=1}^{K} \log P(Y|X_i) - KL(q||P)$

**Objective:**

$$q_{opt} = \arg \max_{q(X)} \frac{1}{K} \sum_{i=1}^{K} \log P(Y|X_i) - KL(q||P)$$
Example model: $X \rightarrow Y$

Objective:

$$q_{opt} = \arg\max_{q(X)} \frac{1}{K} \sum_{i=1}^{K} \log P(Y|X_i) - KL(q||P)$$

**Symbolic differentiation** is straight-forward in this representation:

$$\frac{\partial}{\partial \theta} \log P(Y|X), \quad \frac{\partial}{\partial \theta} KL$$

are easy to compute for a large class of models [Titsias and Lazaro-Gredilla 2014]
**Examples:** Bayesian logistic regression, variable selection, Gaussian process (GP) hyper-parameter estimation, and more [Titsias and Lazaro-Gredilla 2014]

**Example: Bayesian logistic regression**

Given dataset with $x_i \in \mathbb{R}^d$ and $y_i \in \{0, 1\}$ for $n \leq N$, we define

$$P(Y|X, \eta) = \prod_{i=1}^{N} \sigma(y_i x_i^T \eta)$$

for some vector of weights $\eta$ with prior $P(\eta) = \mathcal{N}(0, I_d)$. Define

$$q(\eta|\theta = \{\mu, C\}) = \mathcal{N}(\eta; \mu, CC^T)$$

Symbolically differentiate and optimise wrt $\frac{\partial}{\partial \theta} \log \left( \prod_{i=1}^{N} \sigma(y_i x_i^T \eta) \right)$, $\frac{\partial}{\partial \theta} KL$
Concrete example

Non-linear density estimation of categorical data (work in progress with Yutian Chen)

Model (using sparse GP with $M$ inducing inputs / outputs $Z$ and $U$):

$$X \sim \mathcal{N}(0, I)$$

$$(F_K, U_K) \sim \text{GP}(X, Z)$$

$$Y \sim \text{Softmax}(F_1, ..., F_K)$$

Approximating distributions: $q(X, F, U) = q(X)q(U)p(F|X, U)$, defining $q(x_n) = \mathcal{N}(m_n, s_n^2)$ and $q(u_k) = \mathcal{N}(\mu_k, C C^T)$

We have (with $\epsilon \sim \mathcal{N}(0, I)$):

$$x_n = m_n + s_n\epsilon_n$$

$$u_k = \mu_k + C\epsilon_k$$

$$f_{nk} = K_{nM}K_{MM}^{-1}u_k + \sqrt{K_{nn} - K_{nM}K_{MM}^{-1}K_{Mn}\epsilon_{nk}}$$

$$y_n = \text{Softmax}(f_{n1}, ..., f_{nK})$$
Concrete example

- Original approach took half a year to develop –
  - Deriving variational inference
  - Researching appropriate bound in the statistics literature
  - Derivations for the model

\[
\mathcal{L} = -\sum_{n=1}^{N} \text{KL}(q(x_n) || p(x_n)) \\
+ \sum_{d=1}^{D} \left\{ -\frac{1}{2} \text{Tr}\left[ (I_K \otimes K_{d,M,M}^{-1})(\Sigma_d + \hat{\mu}_d \hat{\mu}_d^T) \right] + \frac{MK}{2} + \frac{1}{2} \log |\Sigma_d| - \frac{1}{2} \log |K_{d,M,M}| \right\} \\
+ \sum_{d=1}^{D} \sum_{n=1}^{N} \left\{ -\frac{1}{2} \text{Tr}\left[ (A_d \otimes K_{d,M,M}^{-1} K_{d,M,n} K_{d,n,M}^{-1} K_{d,M,M}^{-1}) x_n K_{d,M,M}^{-1} \right] (\Sigma_d + \hat{\mu}_d \hat{\mu}_d^T) \right\} \\
+ \sum_{n=1}^{N} [y_{nd} + b_{nd}]^T \otimes ((K_{d,n,M} x_n K_{d,M,M}^{-1})) \hat{\mu}_d - c_{nd} - \frac{1}{2} \text{Tr}[A_d (K_{d,n,n} x_n)] + \frac{1}{2} \text{Tr}(A_d) \text{Tr}(K) \\
= -\sum_{n=1}^{N} \text{KL}(q(x_n) || p(x_n)) \\
+ \sum_{d=1}^{D} \left\{ -\frac{1}{2} \text{Tr}\left[ (I_K \otimes K_{M,M}^{-1})(\Sigma + \hat{\mu} \hat{\mu}^T) \right] + \frac{MK}{2} + \frac{1}{2} \log |\Sigma| - \frac{1}{2} \log |K_{M,M}| \right\}
\]

- Implementation (hundreds of lines of python code)

- New approach –
  - Derivations took a day
  - Programming took a day (15 lines of Python)
Concrete example

- Original approach took half a year to develop –
  - Deriving variational inference
  - Researching appropriate bound in the statistics literature
  - Derivations for the model
  - Implementation (hundreds of lines of python code)

- New approach –
  - Derivations took a day
  - Programming took a day (15 lines of Python)

```python
print 'Building model...'
X = m + T.exp(s) * eps_NQ
U = mu + T.tril(L).dot(eps_MK)

dist_ZZ = T.sum(((T.reshape(Z, (M, 1, Q))) - Z) / ard)**2, 2)
dist_ZX = T.sum(((T.reshape(Z, (M, 1, Q))) - X) / ard)**2, 2)
Kmm = sf2 * T.exp(-dist_ZZ / 2.0)
Kmn = sf2 * T.exp(-dist_ZX / 2.0)
Knn = sf2

KmmInv = sT.matrix_inverse(Kmm)
A = KmmInv.dot(Kmn)
B = Knn - T.sum(Kmn * KmmInv.dot(Kmn), 0)
F = A.T.dot(U) + B[:, None] * eps_NK**0.5 * eps_NK
S = T.nnet.softmax(F)
```
Disadvantages of this approach

- Studying how symbolic differentiation works is important though –
  - Careless implementation can take long to run
  - But careful implementation (together with mini batches) can actually scale well!

- Only suitable when variational inference is; As usual in variational inference depends on the family of approximating distributions

- We can have large variance in the approximate integration
  - Either use more samples (slower to run),
  - Or use variance reduction techniques [Wang, Chen, Smola, and Xing 2013]
Disadvantages of this approach

- Studying how symbolic differentiation works is important though –
  - Careless implementation can take long to run
  - But careful implementation (together with mini batches) can actually scale well!

- Only suitable when variational inference is; As usual in variational inference depends on the family of approximating distributions
  - We can have large variance in the approximate integration
    - Either use more samples (slower to run),
    - Or use variance reduction techniques [Wang, Chen, Smola, and Xing 2013]
Disadvantages of this approach

- Studying how symbolic differentiation works is important though –
  - Careless implementation can take long to run
  - But careful implementation (together with mini batches) can actually scale well!

- Only suitable when variational inference is; As usual in variational inference depends on the family of approximating distributions

- We can have large variance in the approximate integration
  - Either use more samples (slower to run),
  - Or use variance reduction techniques [Wang, Chen, Smola, and Xing 2013]
Thank you