Tutorial on Differentiable Probabilistic Programming for Agent-Based Models

ICAIF 2024

nicholas.bishop@cs.ox.ac.uk

joel.dyer@cs.ox.ac.uk

Large Agent Collider

"Effect a step change in the ability to deploy valid and robust large-scale agent-based models"







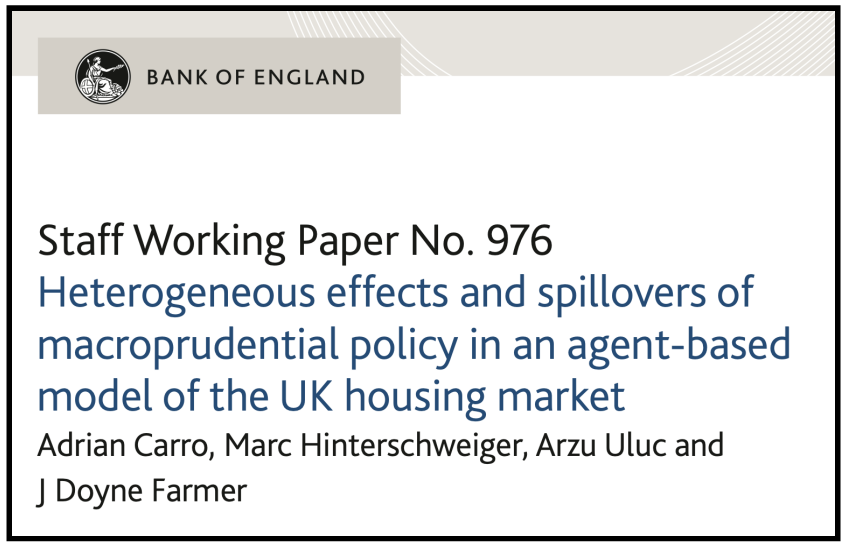
ABMs

A population of **agents**

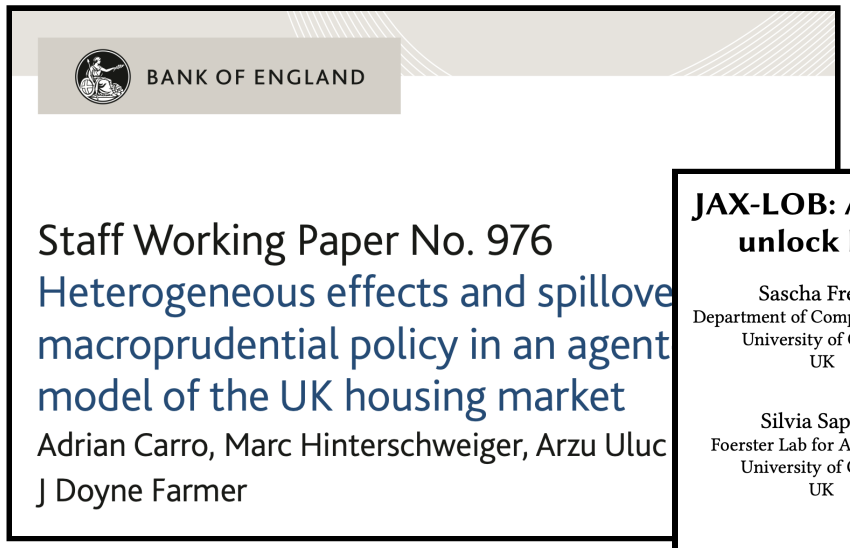
Interacting with each other and the **environment**

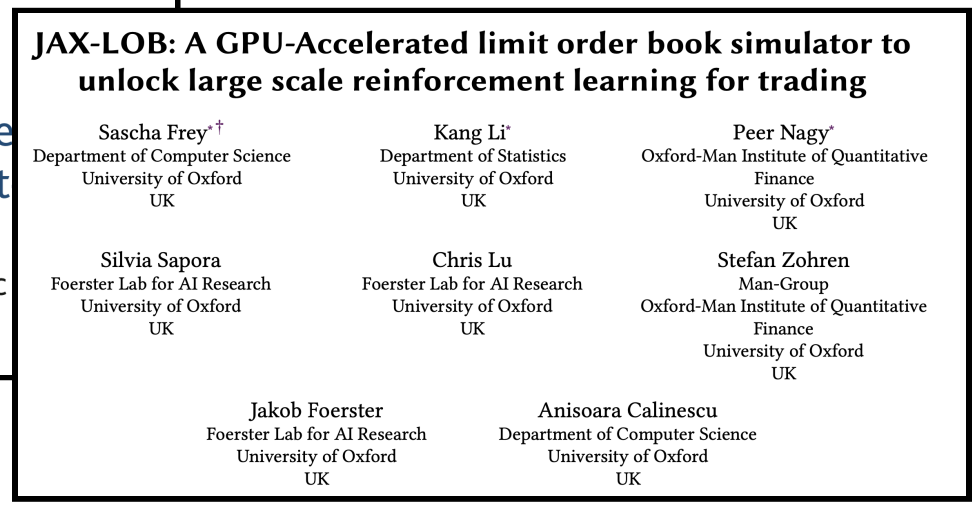
Microscopic Behaviour → **Macroscopic Behaviour**

ABMs in Finance

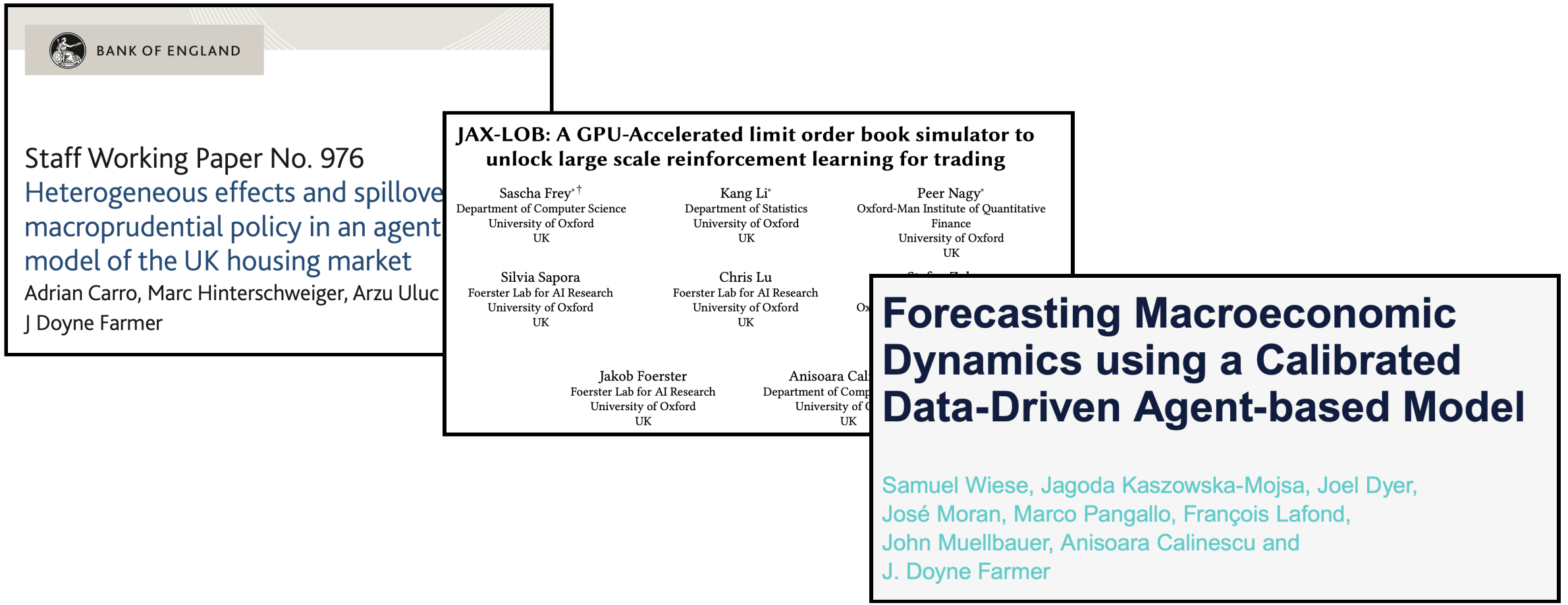


ABMs in Finance





ABMs in Finance



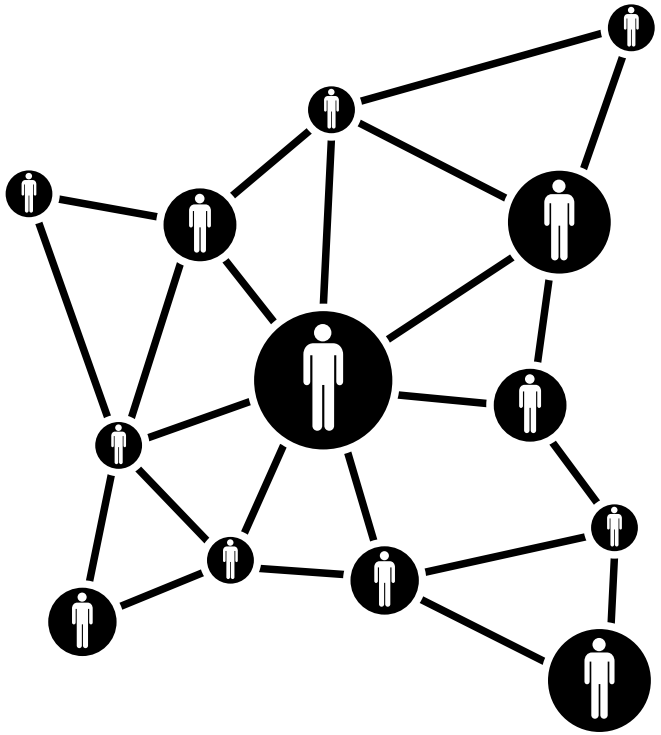
Calibration

Calibration refers to the procedure of **tuning an ABM's parameters** so that its behaviour matches that of the real world

This is in contrast to **validation**, which is the process of determining whether the underlying **assumptions and rule-sets used by an ABM are sound**

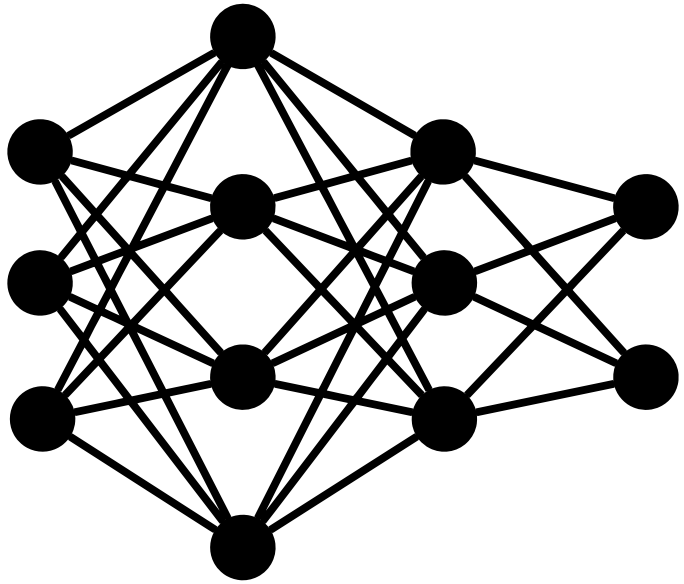
ABM	ML
Calibration	Training
Validation	Model Selection

Agent-based models (ABMs) can be comprised of **millions of agents** interacting over long time horizons



This makes ABMs very difficult to **calibrate**

Neural networks can be comprised of **billions of neurons** interacting over long sequence horizons



Can we **calibrate ABMs** in the same way we **train neural networks**?

Gradient descent and **automatic differentiation (AD)** are the computational workhorses behind modern deep learning

Can we **apply the same tools to ABM** calibration?

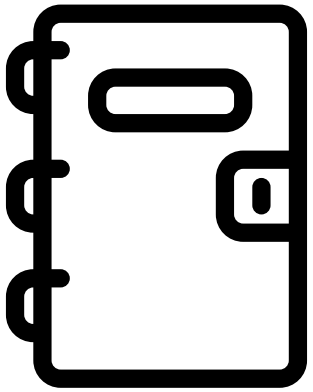
Agenda

A Brief Overview of Automatic Differentiation

Differentiating Through Discrete and Stochastic Programs

Building Differentiable Programs and ABMs

Generalised Variational Inference with Differentiable ABMs



Follow along using the Jupyter notebooks
available at

`https://largeagentcollider.github.io/icaif_tutorial/`

Agenda

A Brief Overview of Automatic Differentiation

Differentiating Through Discrete and Stochastic Programs

Building Differentiable Programs and ABMs

Generalised Variational Inference with Differentiable ABMs

Finite Differences

$$\frac{df}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Finite Differences

$$\frac{df}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

$$\frac{df}{dx} \approx \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

$$f : \mathbb{R}^2 \to \mathbb{R}$$

$$f : \mathbb{R}^2 \to \mathbb{R}$$

$$\frac{\partial f}{\partial x_1} \approx \frac{f(x_1 + \epsilon, x_2) - f(x_1, x_2)}{\epsilon}$$

$$f : \mathbb{R}^2 \to \mathbb{R}$$

$$\frac{\partial f}{\partial x_1} \approx \frac{f(x_1 + \epsilon, x_2) - f(x_1, x_2)}{\epsilon}$$

$$\frac{\partial f}{\partial x_2} \approx \frac{f(x_1, x_2 + \epsilon) - f(x_1, x_2)}{\epsilon}$$

$$f : \mathbb{R}^2 \to \mathbb{R}$$

$$\frac{\partial f}{\partial x_1} \approx \frac{f(x_1 + \epsilon, x_2) - f(x_1, x_2)}{\epsilon}$$

$$\frac{\partial f}{\partial x_2} \approx \frac{f(x_1, x_2 + \epsilon) - f(x_1, x_2)}{\epsilon}$$

⛔ Three function evaluations required!

$$f : \mathbb{R}^n \to \mathbb{R}$$

$$\frac{\partial f}{\partial x_n} \approx \frac{f(x_1 + \epsilon, \ldots, x_n) - f(x_1, \ldots, x_n)}{\epsilon}$$

$$\vdots$$

$$\frac{\partial f}{\partial x_n} \approx \frac{f(x_1, \ldots, x_n + \epsilon) - f(x_1, \ldots, x_n)}{\epsilon}$$

Number of function evaluations scales with input dimension!

Symbolic Diff

Hard-code the derivatives for simple expressions $\cos$ $\exp$ x^n x^{-1}

Symbolic Diff

Hard-code the derivatives for simple expressions $\cos \quad \exp \quad x^n \quad x^{-1}$

Write functions as **symbolic expressions**

Symbolic Diff

Hard-code the derivatives for simple expressions $\cos \quad \exp \quad x^n \quad x^{-1}$

Write functions as **symbolic expressions**

Recursively apply the rules of calculus until the expression reduces to simple expressions

Symbolic Diff

Hard-code the derivatives for simple expressions $\cos \quad \exp \quad x^n \quad x^{-1}$

Write functions as **symbolic expressions**

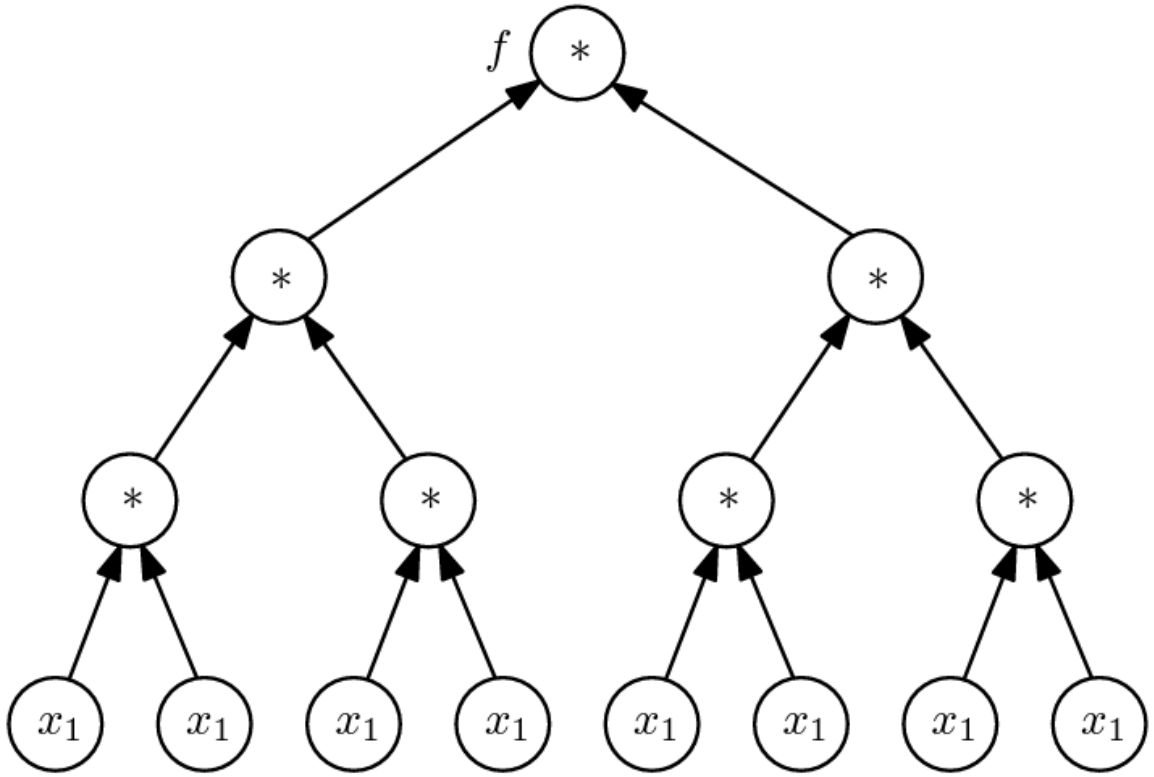
Recursively apply the rules of calculus until the expression reduces to simple expressions

Compact Representations

```
1 def f(x):
2     x = x*x
3     x = x*x
4     x = x*x
5     return x
```

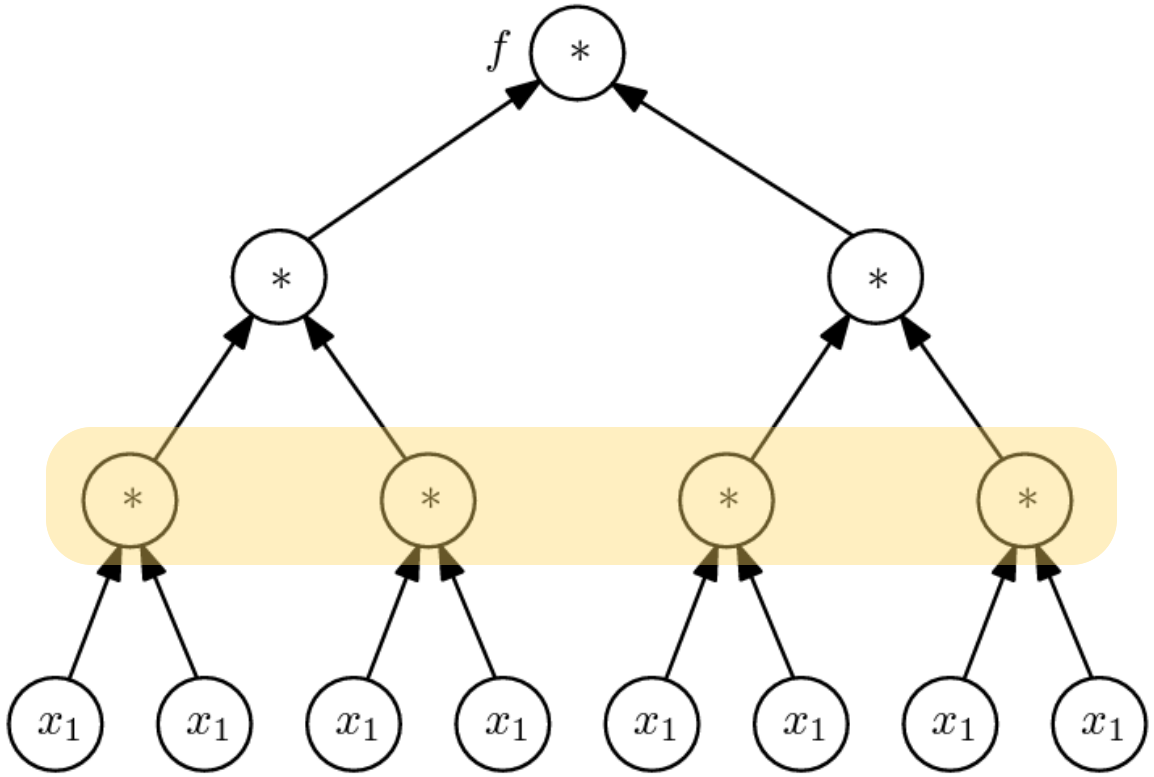
Compact Representations

```
1 def f(x):
2     x = x*x
3     x = x*x
4     x = x*x
5     return x
```



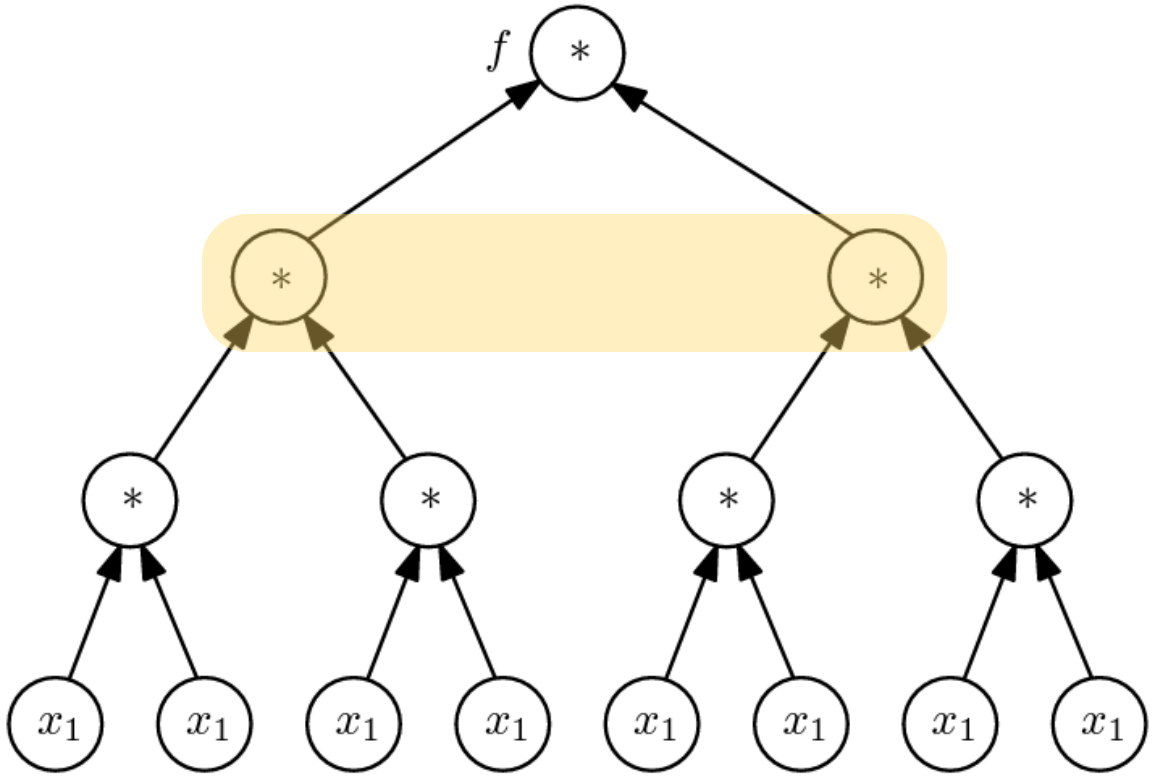
Compact Representations

```
1  def f(x):
2      x = x*x
3      x = x*x
4      x = x*x
5      return x
```



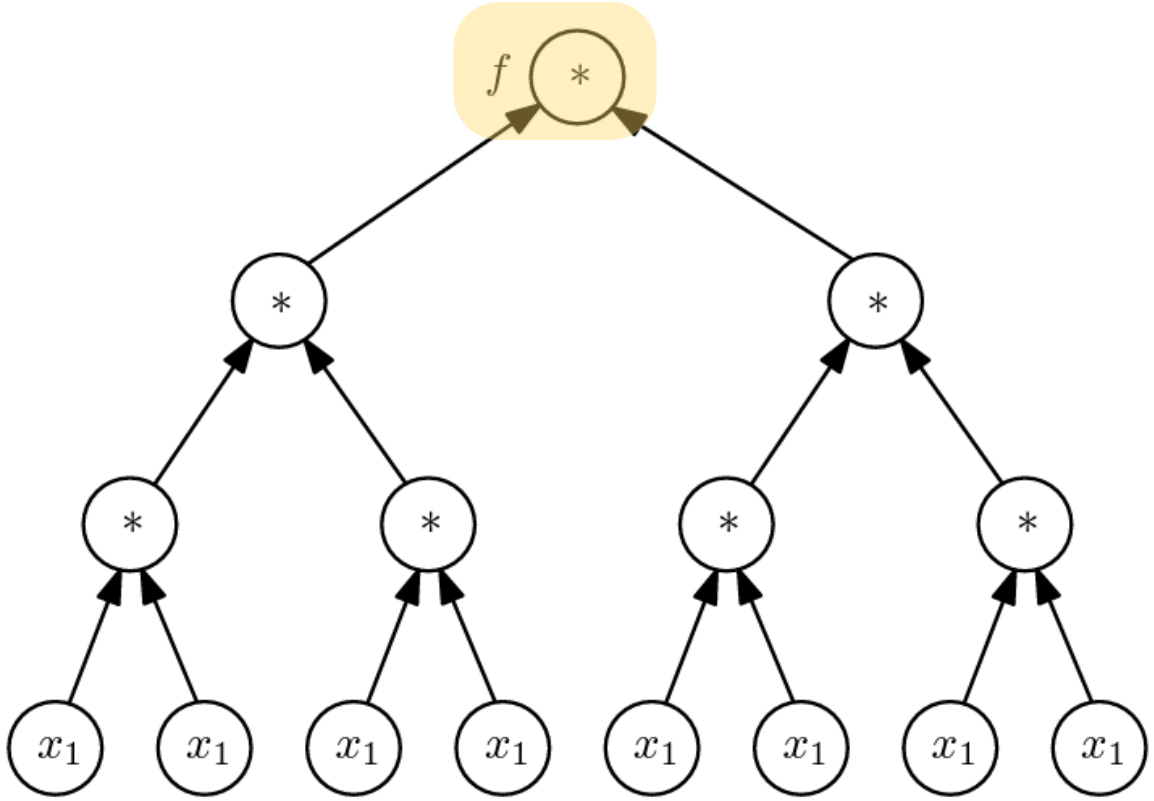
Compact Representations

```
1  def f(x):
2      x = x*x
3      x = x*x
4      x = x*x
5      return x
```



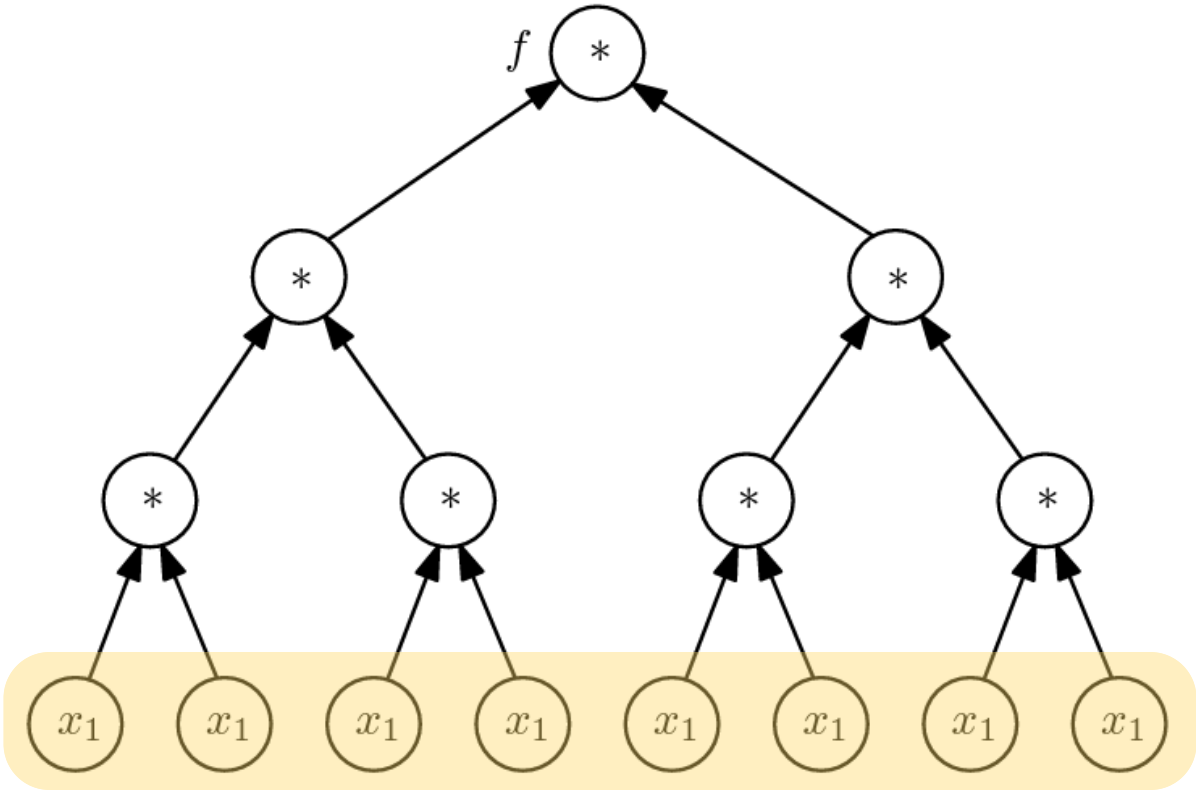
Compact Representations

```
1  def f(x):
2      x = x*x
3      x = x*x
4      x = x*x
5      return x
```



Compact Representations

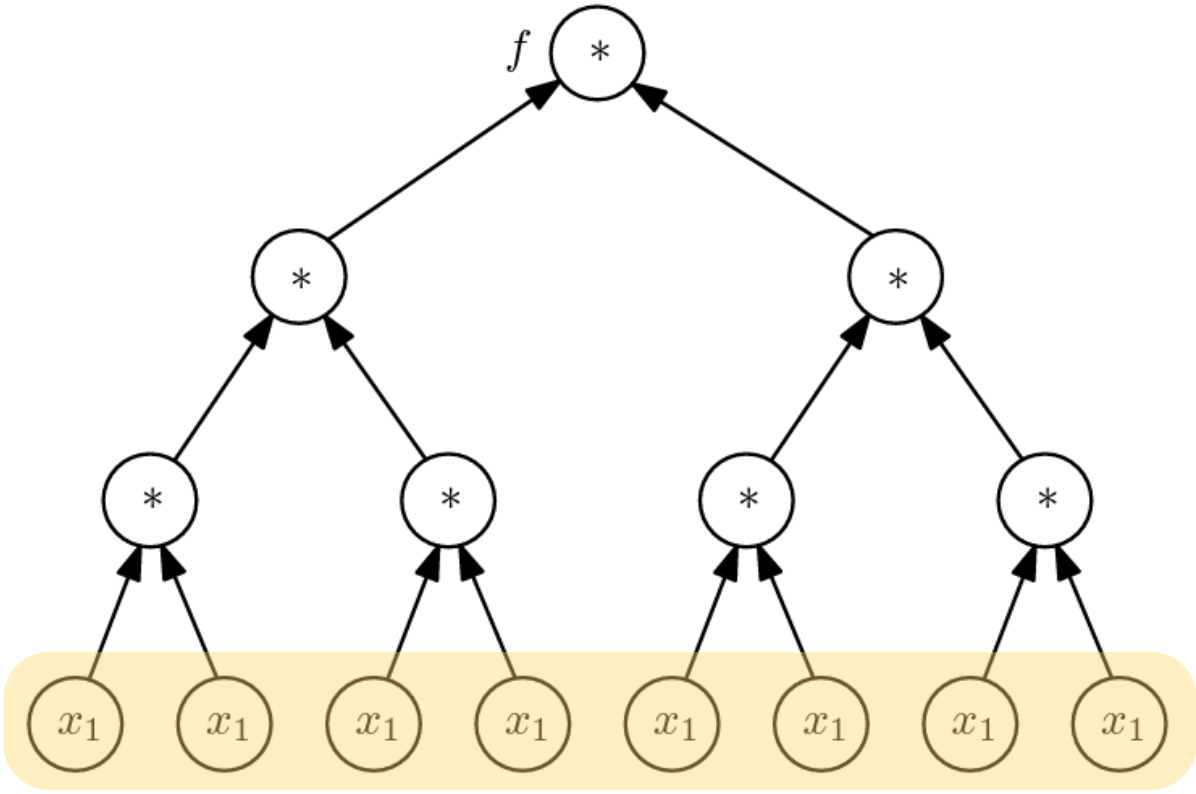
```
1 def f(x):
2       x = x*x
3       x = x*x
4       x = x*x
5       return x
```



Compact Representations

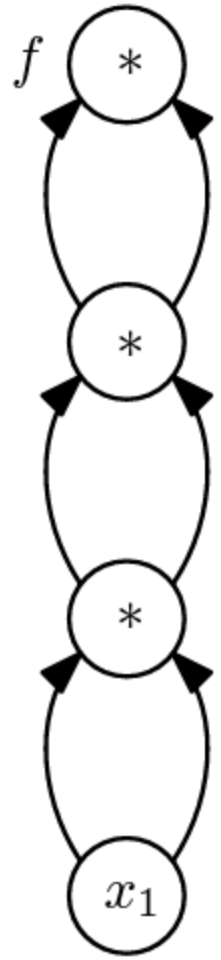
```
1 def f(x):
2       x = x*x
3       x = x*x
4       x = x*x
5       return x
```

Inefficient Representation



Compact Representations

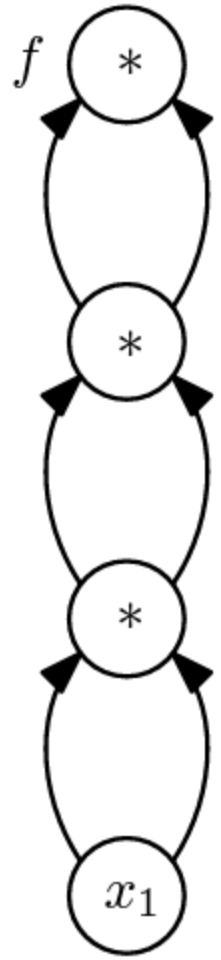
```
1 def f(x):
2     x = x*x
3     x = x*x
4     x = x*x
5     return x
```



Compact Representations

```
1 def f(x):
2     x = x*x
3     x = x*x
4     x = x*x
5     return x
```

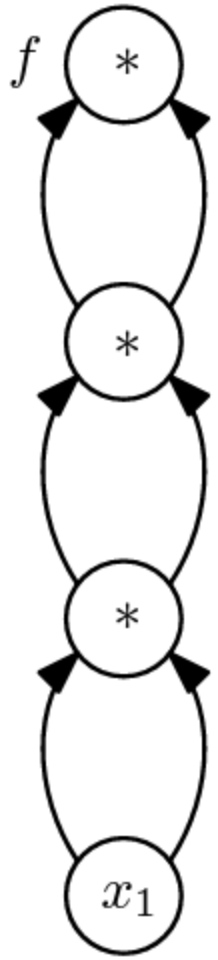
Each operation and variable is represented only once



Compact Representations

```
1 def f(x):
2       x = x*x
3       x = x*x
4       x = x*x
5       return x
```

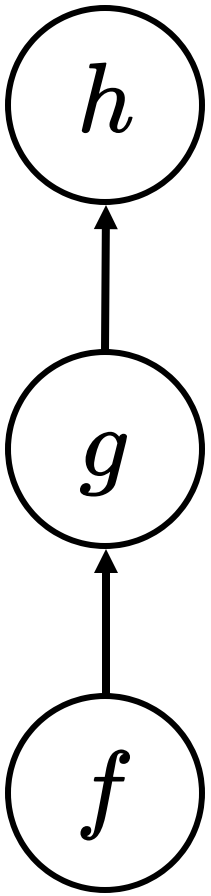
Each operation and variable is represented only once



PyTorch

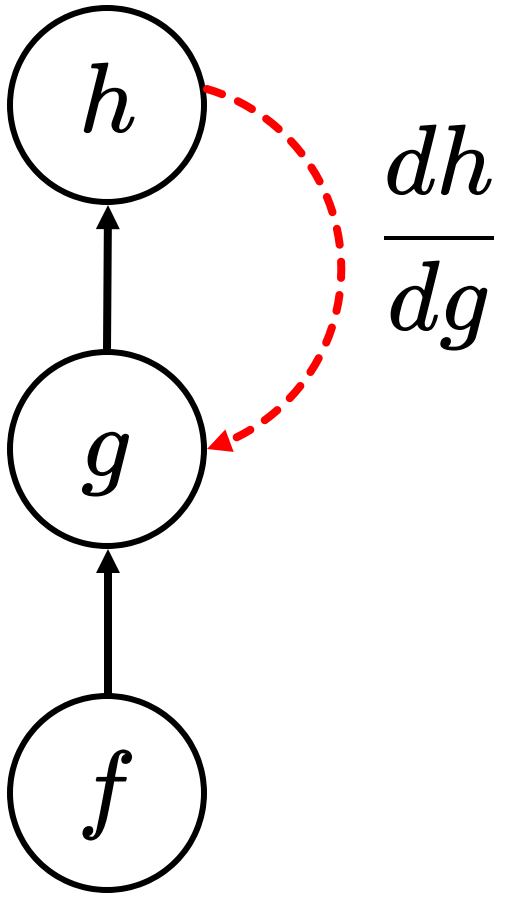
TensorFlow

We can apply the chain rule in **two directions**



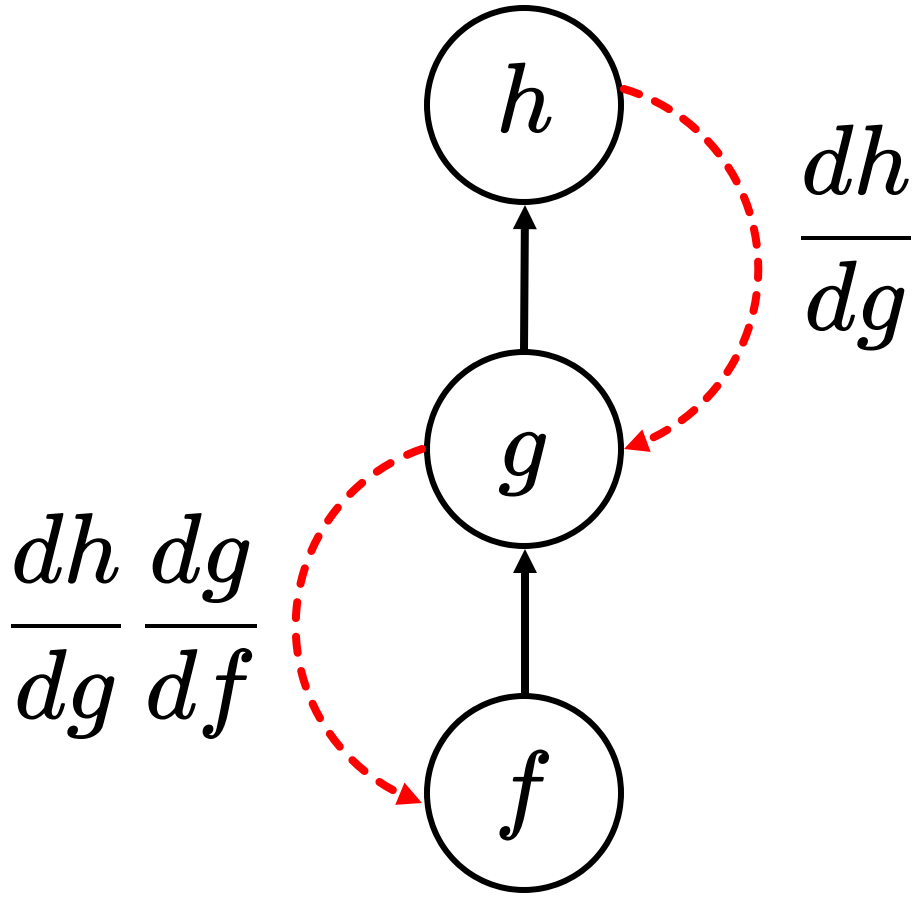
We can apply the chain rule in **two directions**

If we apply the chain rule backward we obtain **reverse-mode AD**



We can apply the chain rule in **two directions**

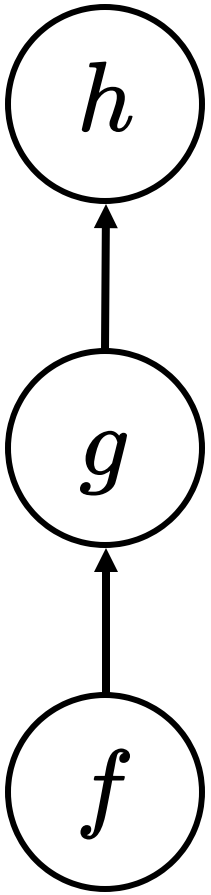
If we apply the chain rule backward we obtain **reverse-mode AD**



We can apply the chain rule in **two directions**

If we apply the chain rule backward we obtain **reverse-mode AD**

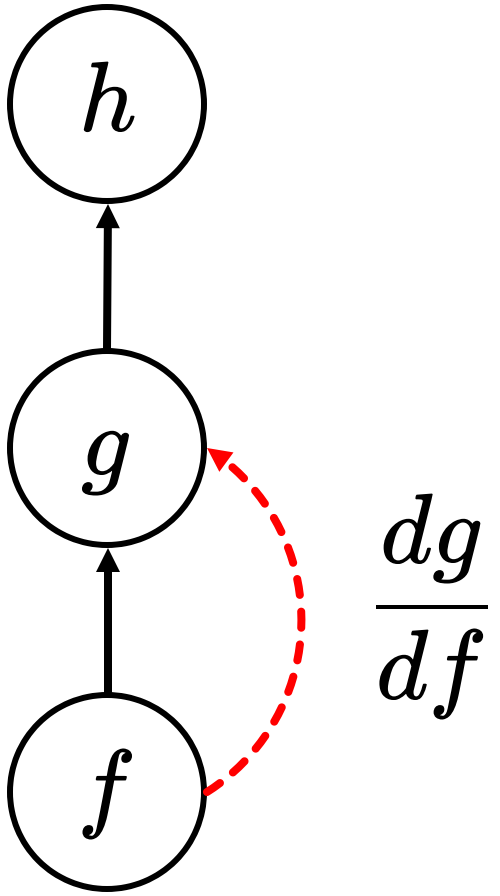
If we apply the chain rule forward we obtain **forward-mode AD**



We can apply the chain rule in **two directions**

If we apply the chain rule backward we obtain **reverse-mode AD**

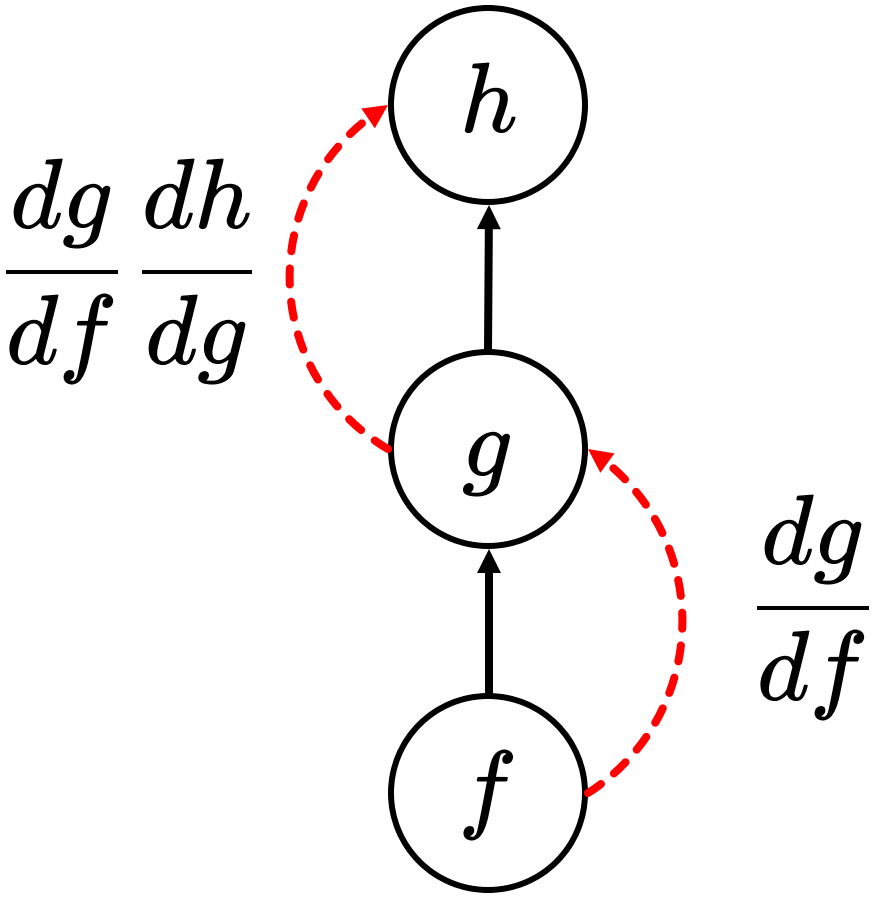
If we apply the chain rule forward we obtain **forward-mode AD**



We can apply the chain rule in **two directions**

If we apply the chain rule backward we obtain **reverse-mode AD**

If we apply the chain rule forward we obtain **forward-mode AD**

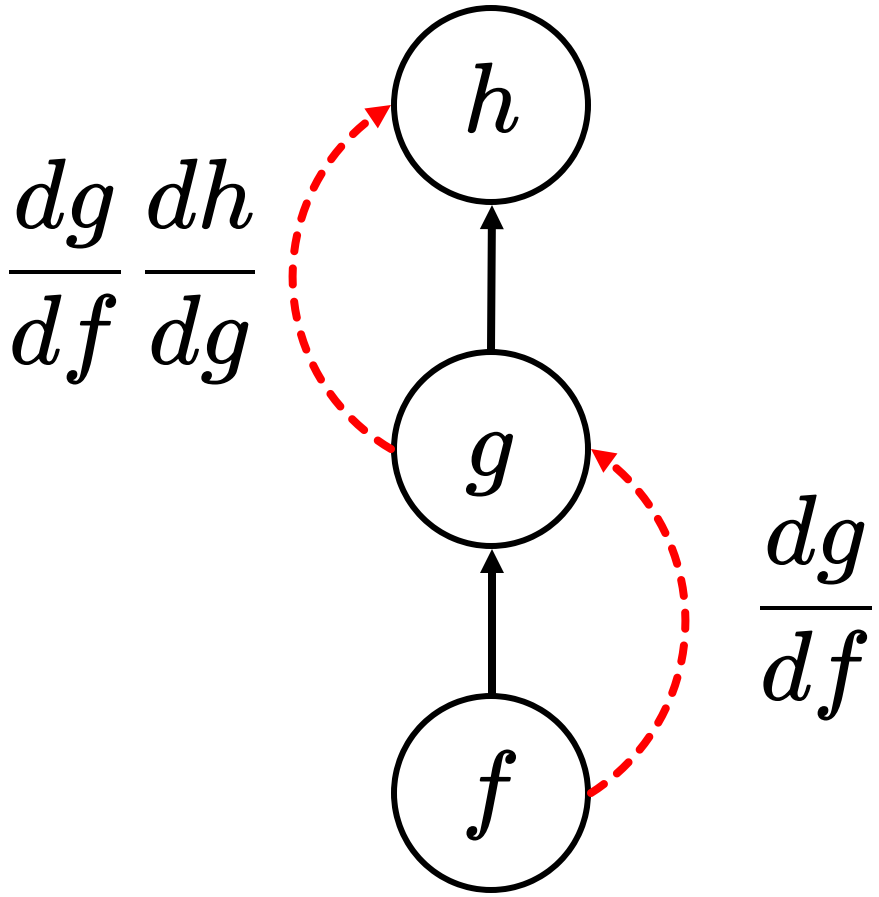


We can apply the chain rule in **two directions**

If we apply the chain rule backward we obtain **reverse-mode AD**

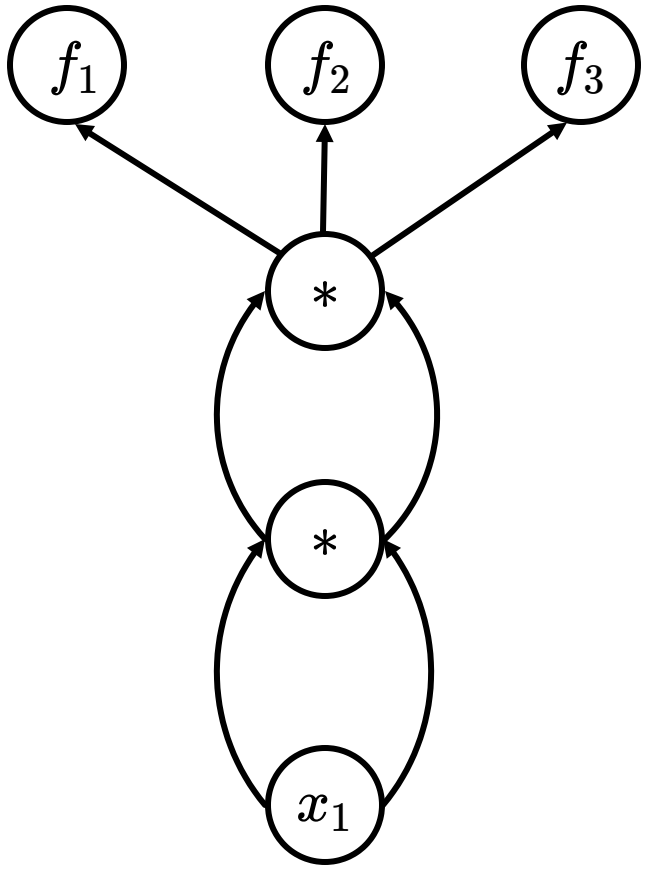
If we apply the chain rule forward we obtain **forward-mode AD**

Which should we **use**?

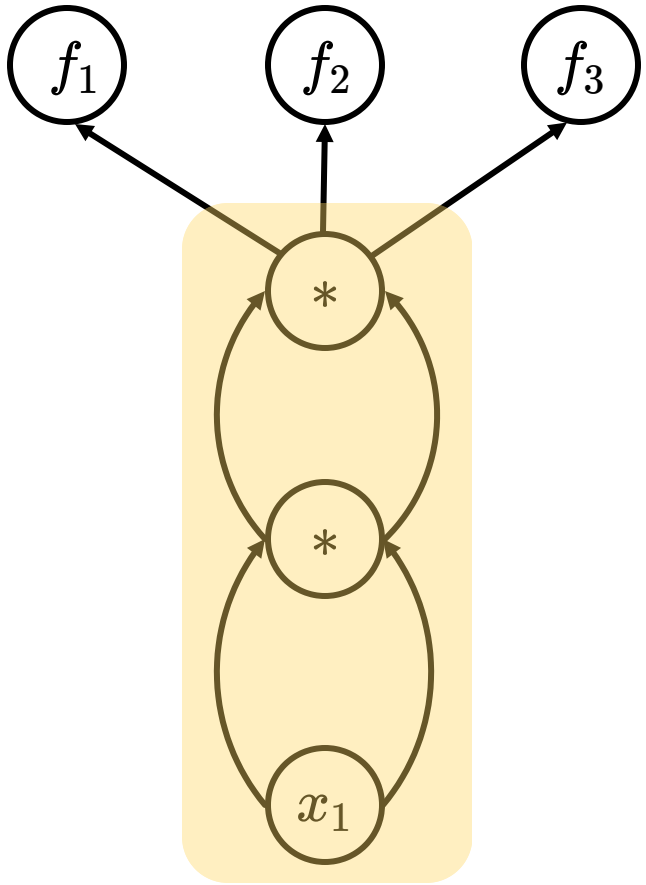


```
1 def f1(x1):
2     x1 = x1 * x1
3     x1 = x1 * x1
4     return [x1, x1, x1]
```

```python
def f1(x1):
    x1 = x1 * x1
    x1 = x1 * x1
    return [x1, x1, x1]
```

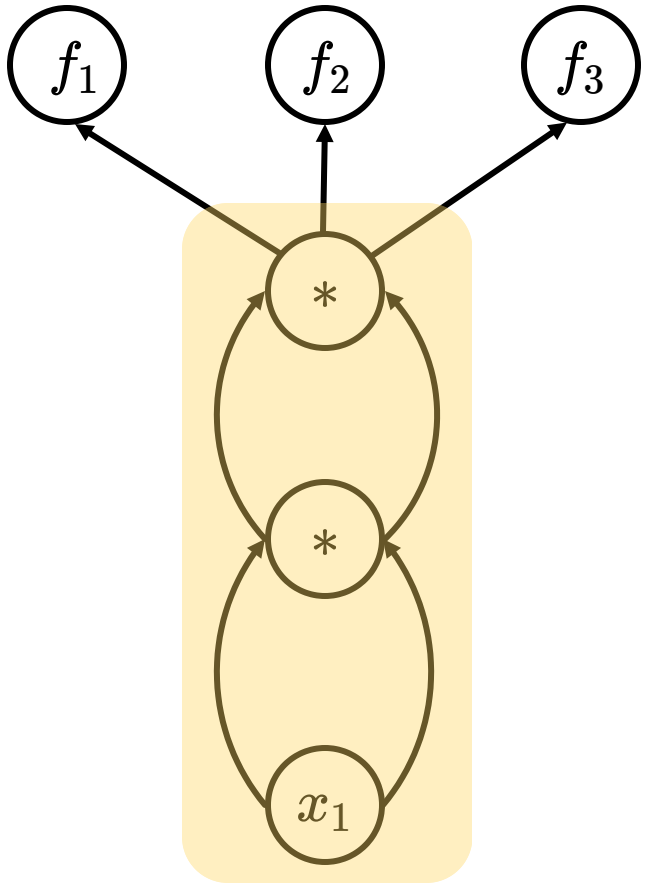


```python
def f1(x1):
    x1 = x1 * x1
    x1 = x1 * x1
    return [x1, x1, x1]
```



```
1 def f1(x1):
2     x1 = x1 * x1
3     x1 = x1 * x1
4     return [x1, x1, x1]
```

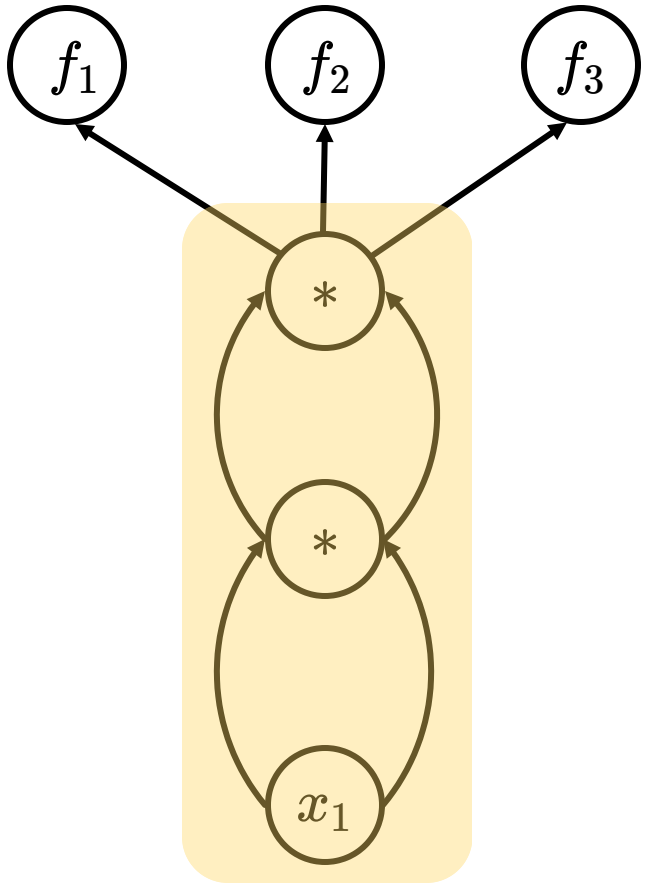
In **forward-mode**, we only need to propagate through the circled nodes **once**!



```
1 def f1(x1):
2     x1 = x1 * x1
3     x1 = x1 * x1
4     return [x1, x1, x1]
```

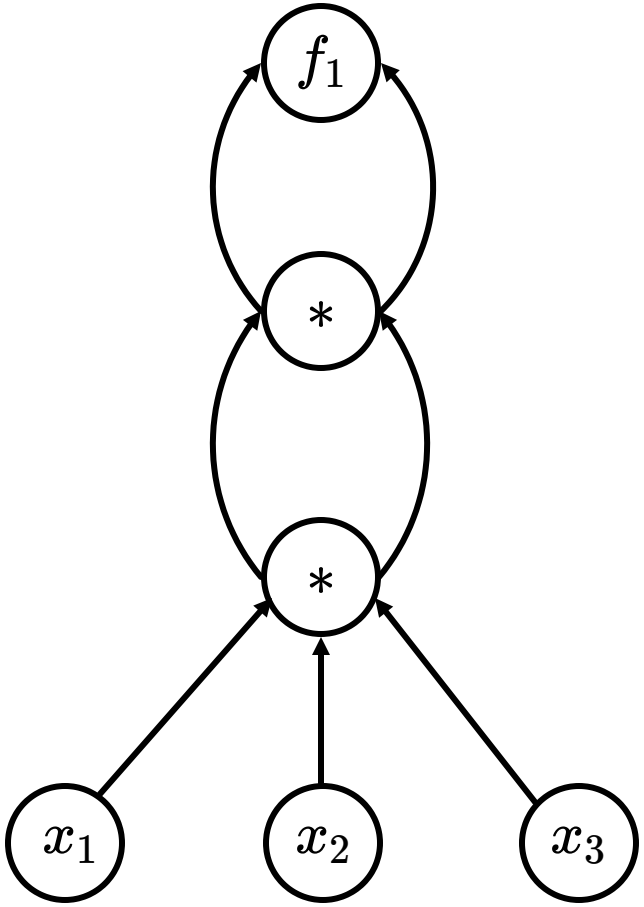
In **forward-mode**, we only need to propagate through the circled nodes **once**!

In **reverse-mode**, we only need to propagate through the circled nodes **three times**!

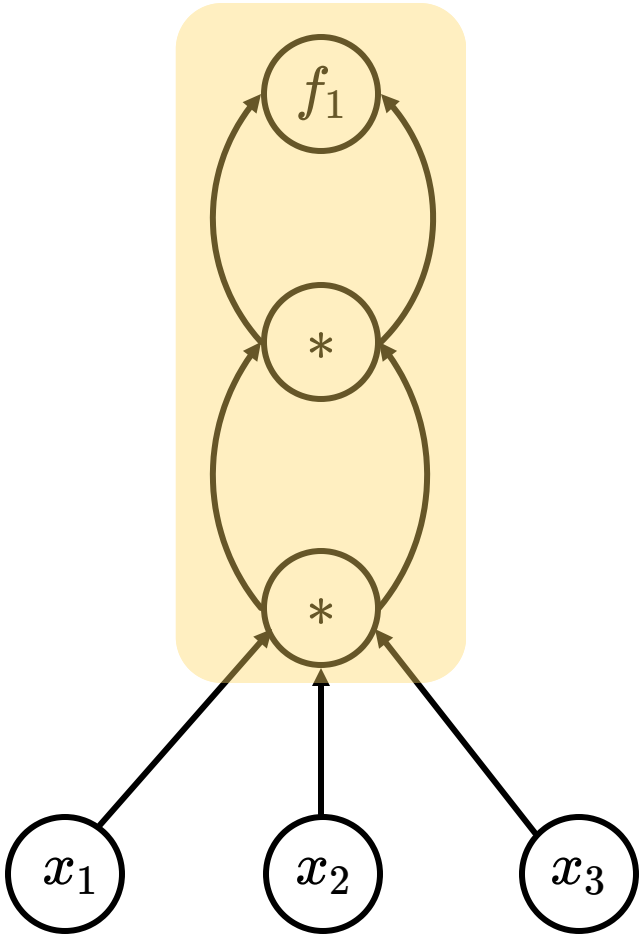


```
1 def f2(x1, x2, x3):
2     x1 = x1 * x2 * x3
3     x1 = x1 * x1
4     return x1
```

```python
def f2(x1, x2, x3):
    x1 = x1 * x2 * x3
    x1 = x1 * x1
    return x1
```

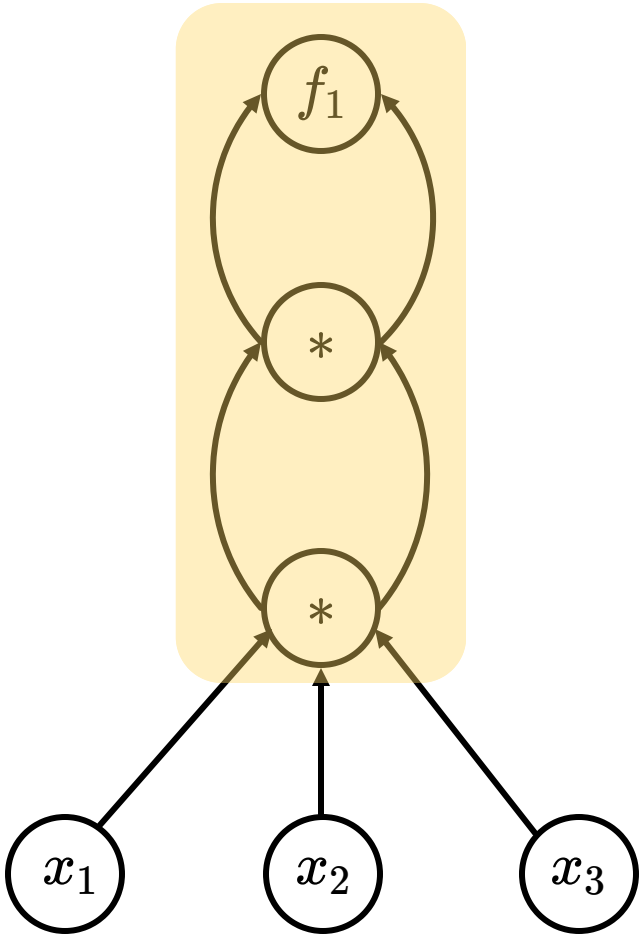


```
1 def f2(x1, x2, x3):
2     x1 = x1 * x2 * x3
3     x1 = x1 * x1
4     return x1
```



```
1 def f2(x1, x2, x3):
2     x1 = x1 * x2 * x3
3     x1 = x1 * x1
4     return x1
```

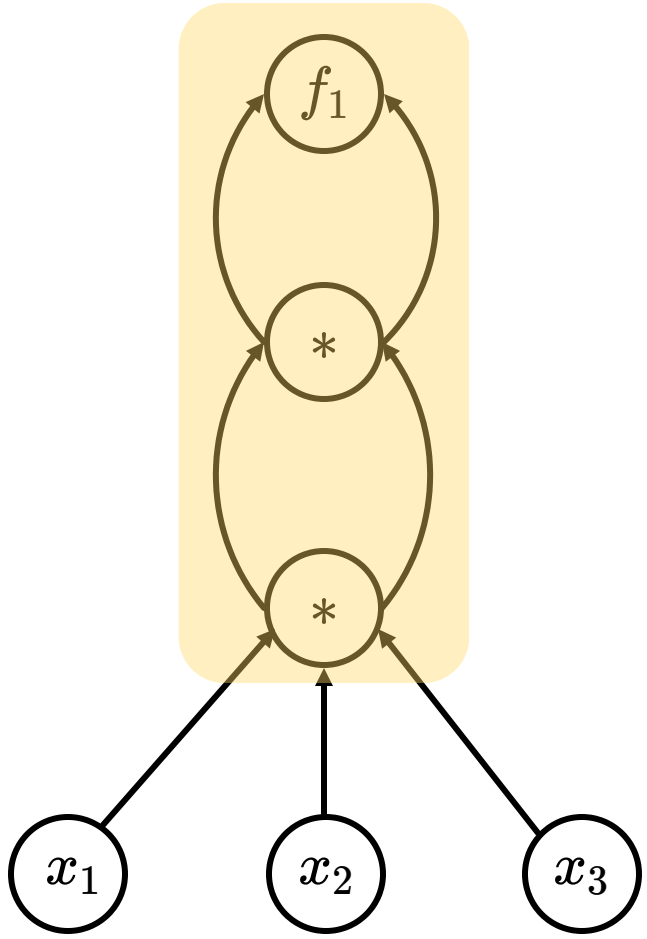
In **forward-mode**, we only need to propagate through the circled nodes **three times**!



```
1 def f2(x1, x2, x3):
2     x1 = x1 * x2 * x3
3     x1 = x1 * x1
4     return x1
```

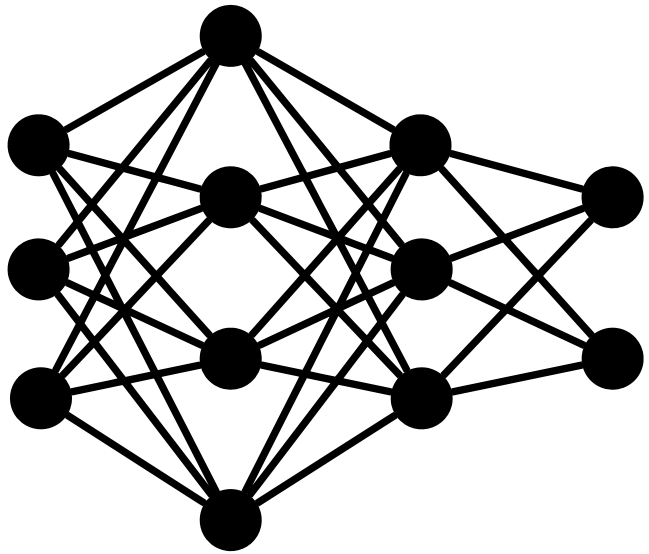
In **forward-mode**, we only need to propagate through the circled nodes **three times**!

In **reverse-mode**, we only need to propagate through the circled nodes **once**!



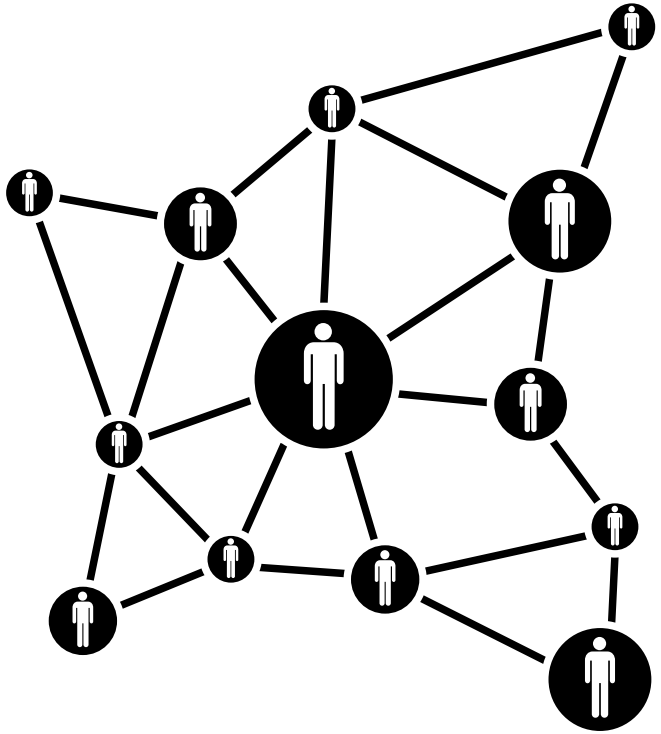
Training **neural networks** involves a very **large number of inputs** (network parameters) and only **one output** (the loss).

Machine learning packages like PyTorch and Tensorflow **focus on reverse-mode AD** as a result.

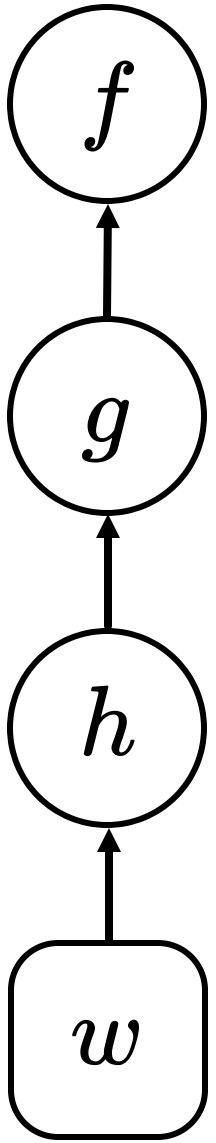


In contrast, **ABMs often have far fewer parameters** making **forward-mode AD viable**.

Moreover, **forward-mode AD is more memory efficient**, which is especially relevant for ABMs!

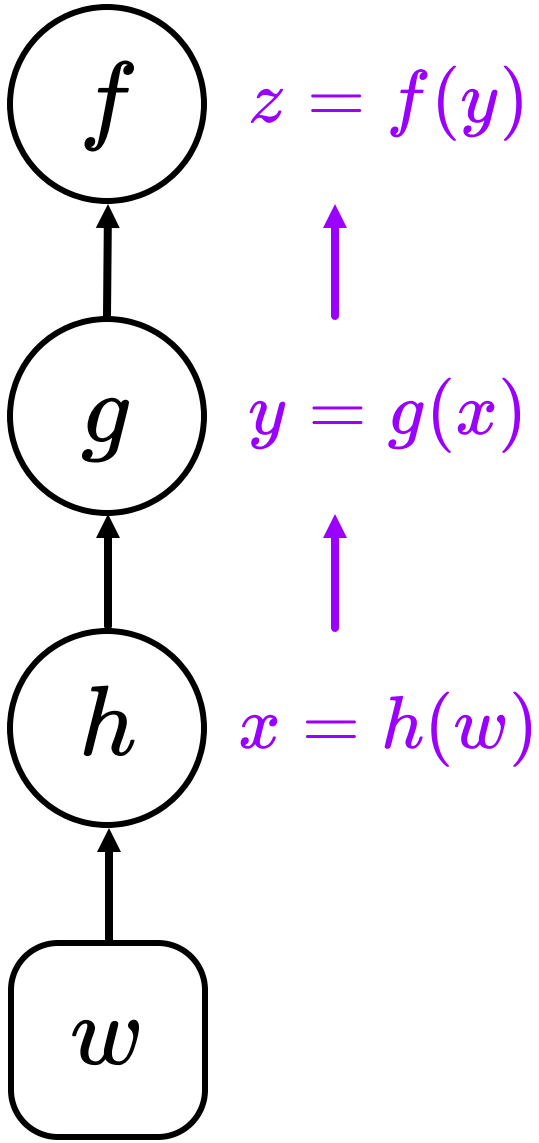


Let's compute gradients with
reverse-mode AD



Let's compute gradients with
reverse-mode AD

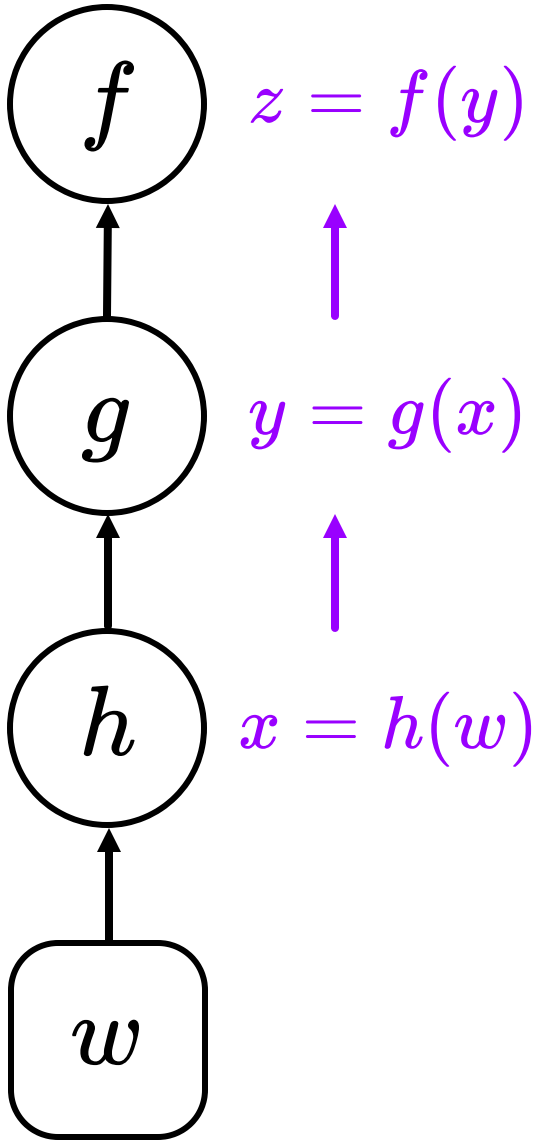
First we need to evaluate the program
with a **forward pass**



Let's compute gradients with **reverse-mode AD**

First we need to evaluate the program with a **forward pass**

We need to store intermediate values for when we perform the **backwards pass**

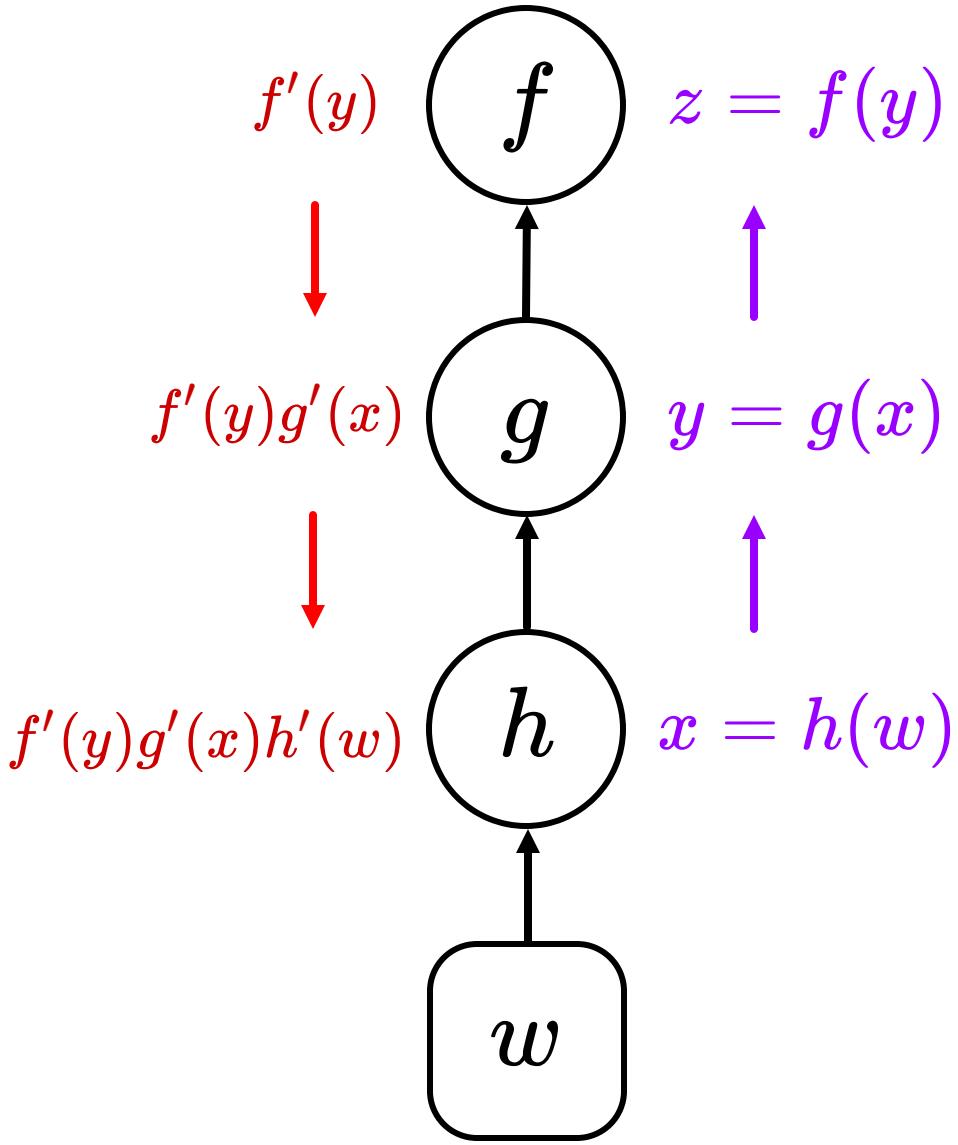


Let's compute gradients with **reverse-mode AD**

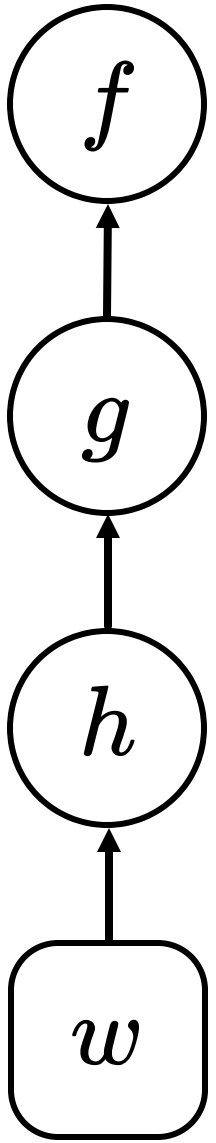
First we need to evaluate the program with a **forward pass**

We need to store intermediate values for when we perform the **backwards pass**

Now we can apply the **chain rule**!

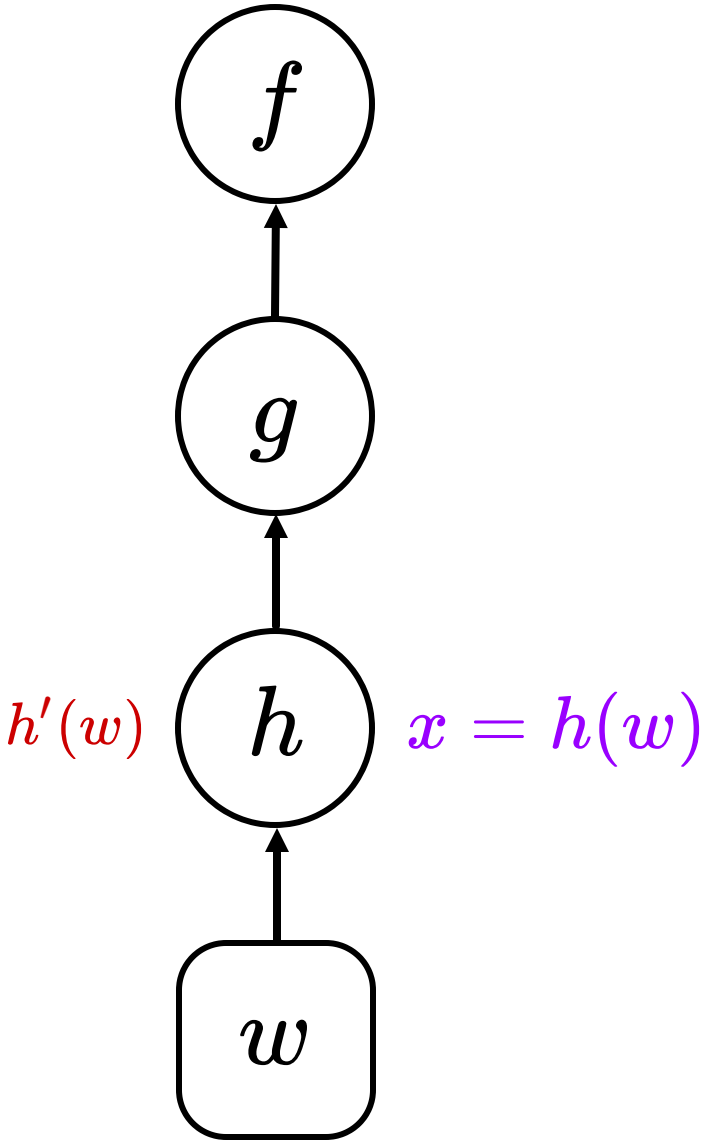


Let's compute gradients with
forward-mode AD



Let's compute gradients with
forward-mode AD

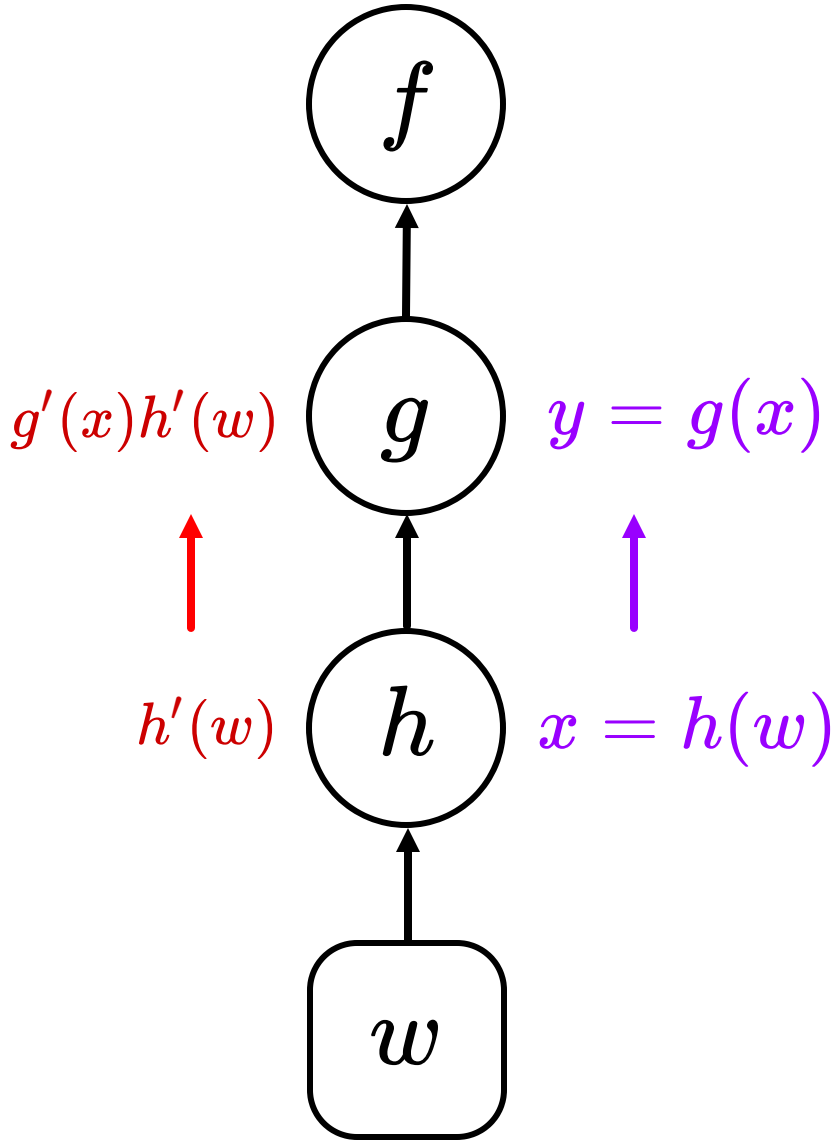
This time we compute gradients
after every operation



Let's compute gradients with
forward-mode AD

This time we compute gradients
after every operation

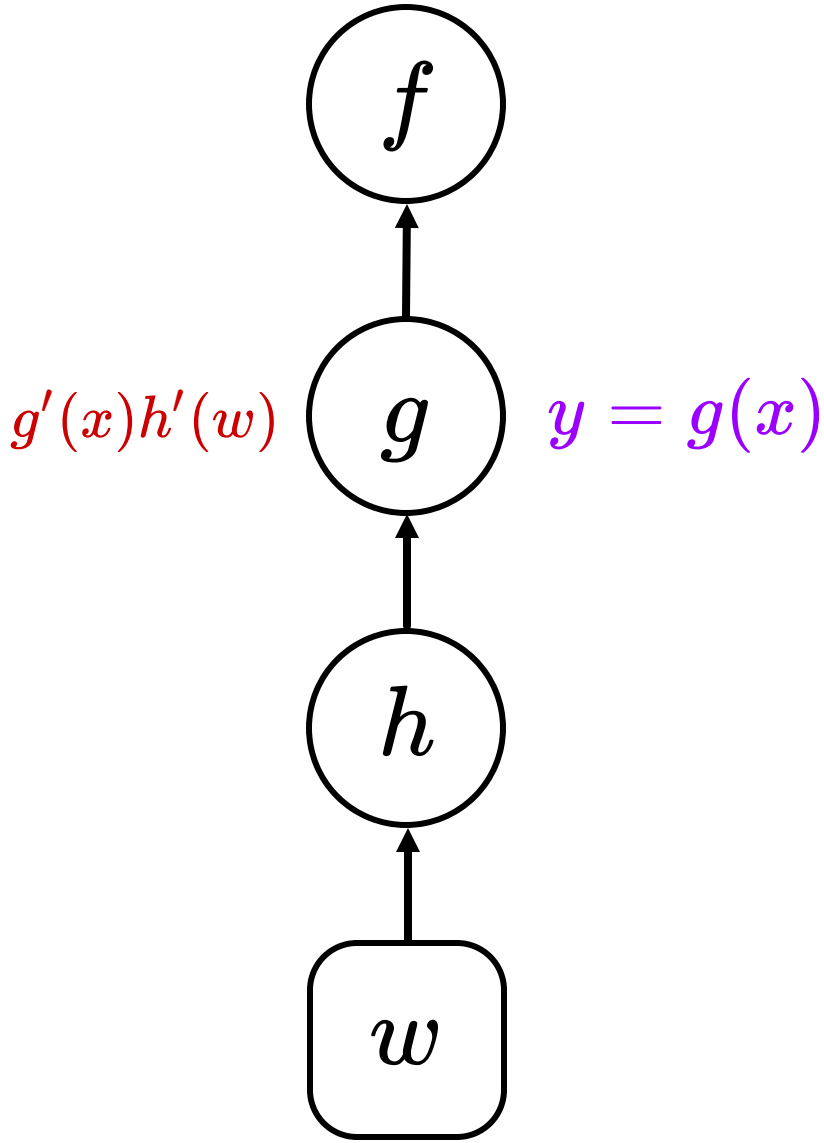
After the next operation, we can
delete old values and gradients



Let's compute gradients with
forward-mode AD

This time we compute gradients
after every operation

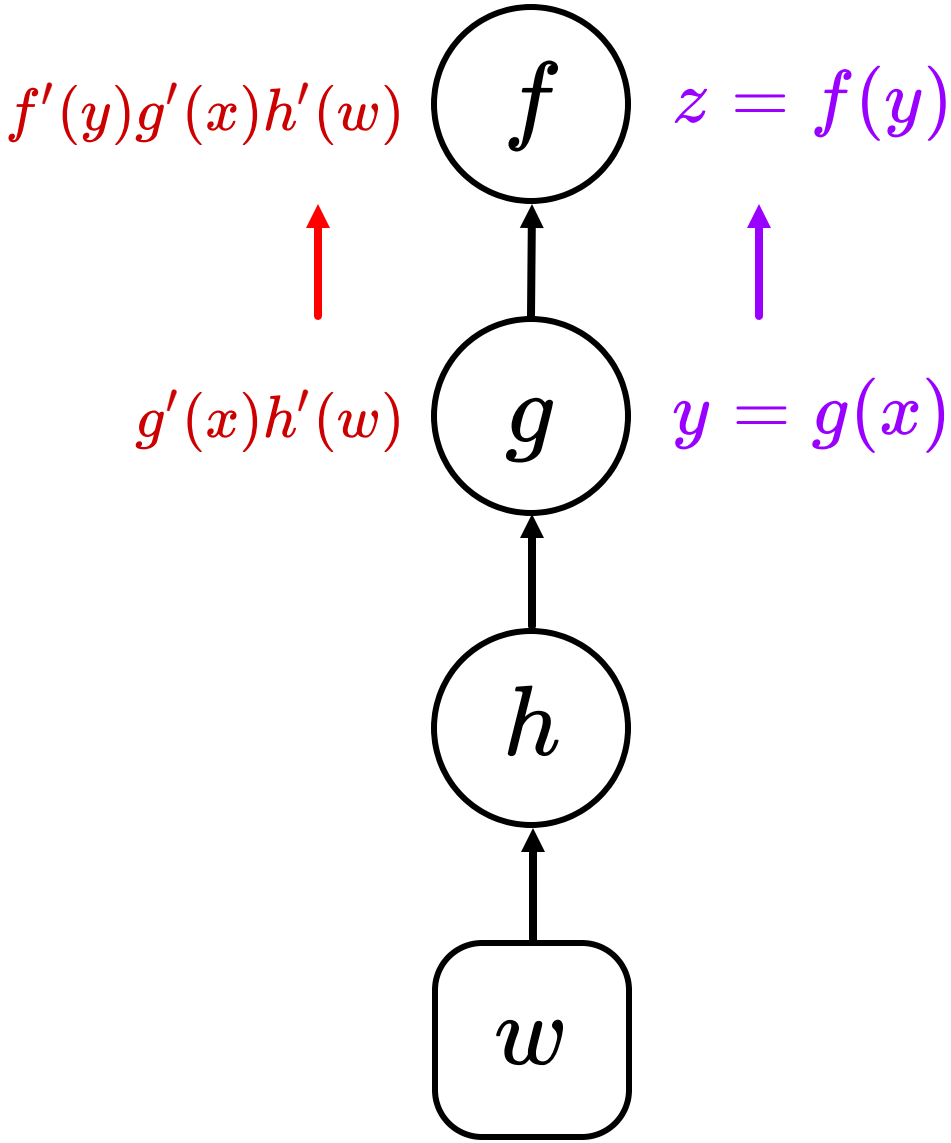
After the next operation, we can
delete old values and gradients



Let's compute gradients with
forward-mode AD

This time we compute gradients
after every operation

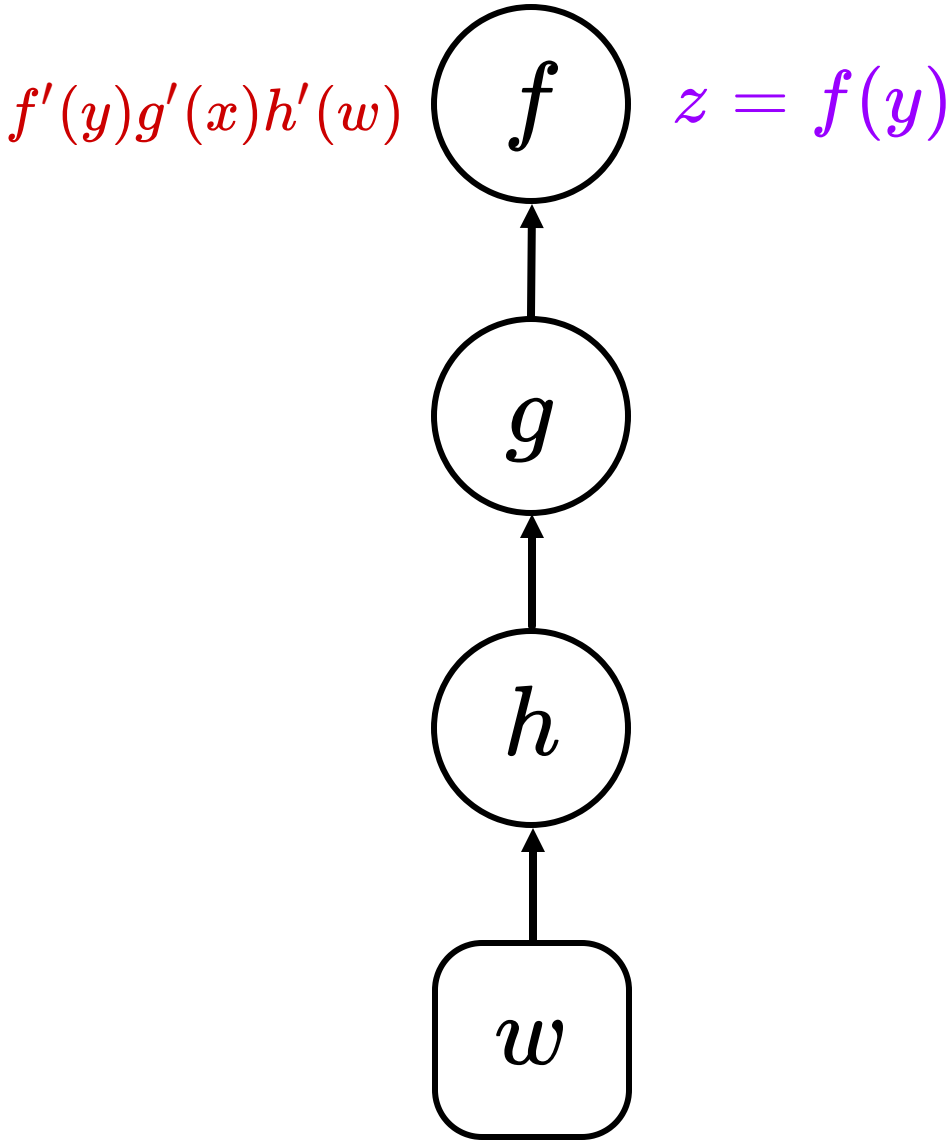
After the next operation, we can
delete old values and gradients



Let's compute gradients with
forward-mode AD

This time we compute gradients
after every operation

After the next operation, we can
delete old values and gradients

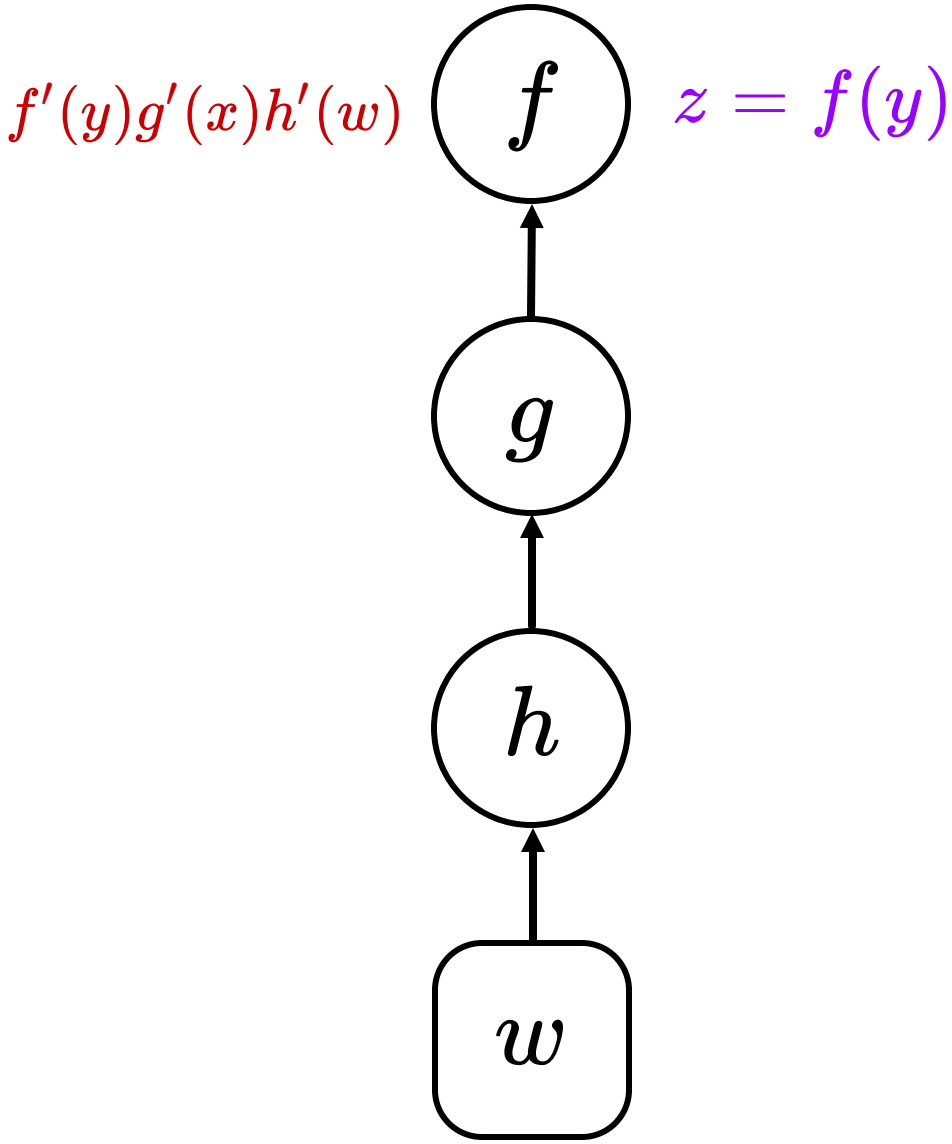


Let's compute gradients with
forward-mode AD

This time we compute gradients
after every operation

After the next operation, we can
delete old values and gradients

Far more memory efficient for large-scale computational graphs!



Agenda

A Brief Overview of Automatic Differentiation

Differentiating Through Discrete and Stochastic Programs

Building Differentiable Programs and ABMs

Generalised Variational Inference with Differentiable ABMs

$$z \sim \mathcal{N}(1, 2)$$
$$y = \theta^2(z + 4)$$

$$z \sim \mathcal{N}(1, 2)$$
$$y = \theta^2(z + 4)$$

$$\frac{df}{dx} = \lim_{h \to 0} \frac{f(\theta + h) - f(\theta)}{h}$$

$$z \sim \mathcal{N}(1, 2)$$

$$y = \theta^2(z + 4)$$

$$\frac{df}{d\theta} = \lim_{h \to 0} \frac{f(\theta + h) - f(\theta)}{h}$$

Random

$$z \sim \mathcal{N}(1, 2)$$
$$y = \theta^2(z + 4)$$

$$\frac{df}{dx} = \lim_{h \to 0} \frac{f(\theta + h) - f(\theta)}{h}$$

← **Random**

We can extend the theory of derivatives to stochastic maps by defining **sample derivatives**

$$z \sim \mathcal{N}(1, 2)$$
$$y = \theta^2(z + 4)$$

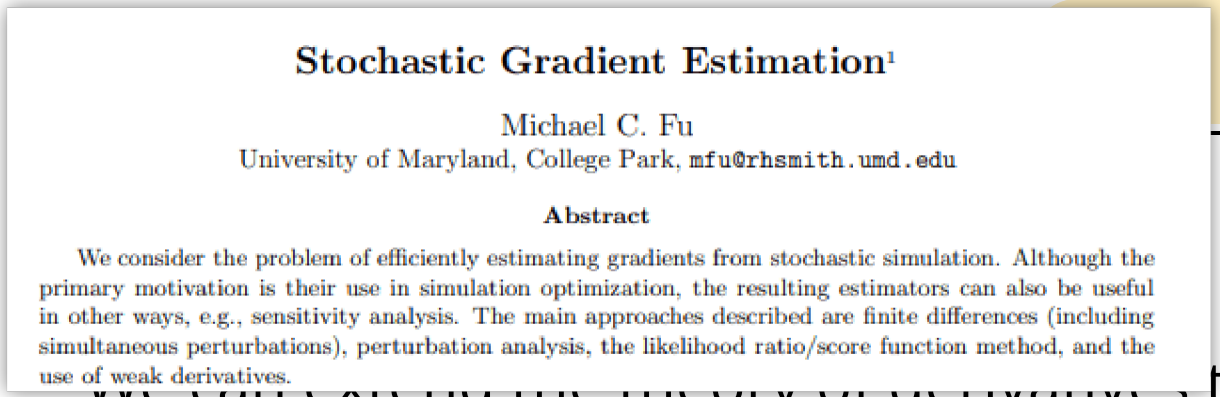
$$\frac{df}{dx} = \lim_{h \to 0} \frac{f(\theta + h) - f(\theta)}{h}$$

← **Random**

We can extend the theory of derivatives to stochastic maps by defining **sample derivatives**

We will **avoid this theory today**, but many methods gradient estimation techniques are based on sample derivatives.

$$z \sim \mathcal{N}(1, 2)$$
$$y = \theta^2(z + 4)$$



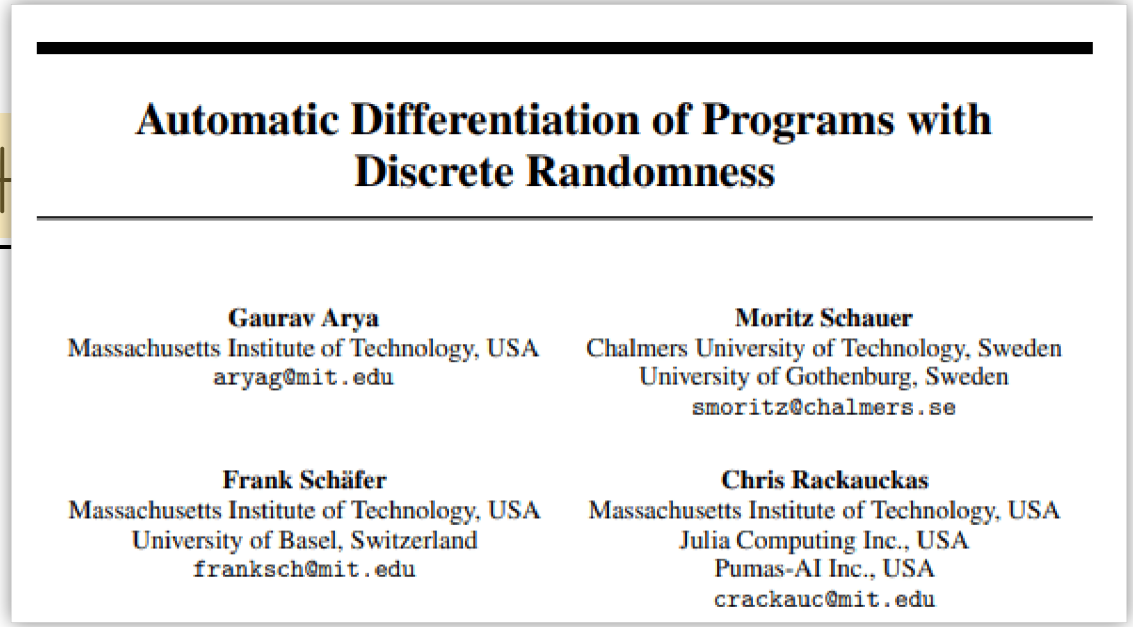
Stochastic Gradient Estimation[1]

Michael C. Fu
University of Maryland, College Park, mfu@rhsmith.umd.edu

Abstract

We consider the problem of efficiently estimating gradients from stochastic simulation. Although the primary motivation is their use in simulation optimization, the resulting estimators can also be useful in other ways, e.g., sensitivity analysis. The main approaches described are finite differences (including simultaneous perturbations), perturbation analysis, the likelihood ratio/score function method, and the use of weak derivatives.

+



Automatic Differentiation of Programs with Discrete Randomness

Gaurav Arya
Massachusetts Institute of Technology, USA
aryag@mit.edu

Moritz Schauer
Chalmers University of Technology, Sweden
University of Gothenburg, Sweden
smoritz@chalmers.se

Frank Schäfer
Massachusetts Institute of Technology, USA
University of Basel, Switzerland
franksch@mit.edu

Chris Rackauckas
Massachusetts Institute of Technology, USA
Julia Computing Inc., USA
Pumas-AI Inc., USA
crackauc@mit.edu

We can extend the theory of derivatives to **derivatives**

We will **avoid this theory today**, but many methods gradient estimation techniques are based on sample derivatives.

In practice, we are typically interested in the **average** behaviour of a model

$$\nabla_\theta \mathbb{E}_{z \sim \mathcal{N}(1,2)} [\theta^2 (z + 4)]$$

In practice, we are typically interested in the **average** behaviour of a model

$$\nabla_\theta \mathbb{E}_{z \sim \mathcal{N}(1,2)} [\theta^2 (z + 4)]$$

This is an instance of the more **general gradient estimation problem**:

$$\nabla_\theta \mathbb{E}_{p(z)} [f_\theta (z)]$$

$$\nabla_\theta \mathbb{E}_{p(z)}[f_\theta(z)]$$

$$\nabla_\theta \mathbb{E}_{p(z)}[f_\theta(z)] = \nabla_\theta \int p(z) f_\theta(z) dz$$

$$\begin{aligned}\nabla_\theta \mathbb{E}_{p(z)}[f_\theta(z)] &= \nabla_\theta \int p(z) f_\theta(z) dz \\ &= \int p(z) \nabla_\theta f_\theta(z) dz\end{aligned}$$

$$
\begin{aligned}
\nabla_\theta \mathbb{E}_{p(z)}[f_\theta(z)] &= \nabla_\theta \int p(z) f_\theta(z) dz \\
&= \int p(z) \nabla_\theta f_\theta(z) dz \\
&= \mathbb{E}_{z \sim p(z)}[\nabla_\theta f_\theta(z)]
\end{aligned}
$$

$$\nabla_\theta \mathbb{E}_{p(z)}[f_\theta(z)] = \nabla_\theta \int p(z) f_\theta(z) dz$$

$$= \int p(z) \nabla_\theta f_\theta(z) dz$$

$$= \mathbb{E}_{z \sim p(z)}[\nabla_\theta f_\theta(z)]$$

The gradient of the expectation is equal to the expectation of the gradient!

Monte Carlo Gradients

$$\nabla_\theta \mathbb{E}_{p(z)}[f_\theta(z)] = \mathbb{E}_{z\sim p(z)}[\nabla_\theta f_\theta(z)]$$

$$= \frac{1}{m}\sum_{i=1}^{m} \nabla_\theta f_\theta(z_i) \quad , \quad z_i \sim p$$

We can use any previous approach
to approximate these gradients!

Finite Differences

$$\nabla_\theta \mathbb{E}_{p(z)}[f_\theta(z)] \approx \frac{\frac{1}{m}\sum_{i=1}^{m} f_{\theta+\epsilon}(z_{i,1}) - \frac{1}{m}\sum_{i=1}^{m} f_\theta(z_{i,2})}{\epsilon},$$

Finite Differences

$$\nabla_\theta \mathbb{E}_{p(z)}[f_\theta(z)] \approx \frac{\frac{1}{m}\sum_{i=1}^{m} f_{\theta+\epsilon}(z_{i,1}) - \frac{1}{m}\sum_{i=1}^{m} f_\theta(z_{i,2})}{\epsilon},$$

Dividing through by the perturbation **increases the variance** of the Monte Carlo sum!

Finite Differences

$$\nabla_\theta \mathbb{E}_{p(z)}[f_\theta(z)] \approx \frac{\frac{1}{m}\sum_{i=1}^{m} f_{\theta+\epsilon}(z_{i,1}) - \frac{1}{m}\sum_{i=1}^{m} f_\theta(z_{i,2})}{\epsilon},$$

Dividing through by the perturbation **increases the variance** of the Monte Carlo sum!

We can reduce variance by **increasing the number of samples**, but this requires more function evaluations.

Finite Differences

$$\nabla_\theta \mathbb{E}_{p(z)}[f_\theta(z)] \approx \frac{\frac{1}{m}\sum_{i=1}^{m} f_{\theta+\epsilon}(z_{i,1}) - \frac{1}{m}\sum_{i=1}^{m} f_\theta(z_{i,2})}{\epsilon},$$

Dividing through by the perturbation **increases the variance** of the Monte Carlo sum!

We can reduce variance by **increasing the number of samples**, but this requires more function evaluations.

We can also **synchronise noise terms** but this is difficult to implement!

$$z_{i,1} = z_{i,2}$$

Parameterized Noise

So far we have assumed that the random noise **does not** depend on the parameters

$$\nabla_\theta \mathbb{E}_{p(z)}\left[f_\theta(z)\right]$$

Parameterized Noise

So far we have assumed that the random noise **does not** depend on the parameters

$$\nabla_\theta \mathbb{E}_{p(z)}\left[f_\theta(z)\right]$$

This is **unrealistic for ABMs**, where parameters often affect the chance of an agent changing state or taking an action.

Parameterized Noise

So far we have assumed that the random noise **does not** depend on the parameters

$$\nabla_\theta \mathbb{E}_{p_\theta(z)}[f_\theta(z)]$$

This is **unrealistic for ABMs**, where parameters often affect the chance of an agent changing state or taking an action.

So we need to solve a more **general problem**!

$$\nabla_\theta \mathbb{E}_{p_\theta(z)}[f_\theta(z)]$$

$$\nabla_\theta \mathbb{E}_{p_\theta(z)}[f_\theta(z)] = \nabla_\theta \left[\int_z p_\theta(z) f_\theta(z) \mathrm{d}z \right]$$

$$\nabla_\theta \mathbb{E}_{p_\theta(z)}[f_\theta(z)] = \nabla_\theta \left[\int_z p_\theta(z) f_\theta(z) \mathrm{d}z \right] = \int_z \nabla_\theta \left[p_\theta(z) f_\theta(z) \right] \mathrm{d}z$$

$$\nabla_\theta \mathbb{E}_{p_\theta(z)}[f_\theta(z)] = \nabla_\theta \left[\int_z p_\theta(z) f_\theta(z) \mathrm{d}z \right] = \int_z \nabla_\theta \left[p_\theta(z) f_\theta(z) \right] \mathrm{d}z$$

$$= \int_z f_\theta(z) \nabla_\theta p_\theta(z) \mathrm{d}z + \int_z p_\theta(z) \nabla_\theta f_\theta(z) \mathrm{d}z$$

$$
\begin{aligned}
\nabla_\theta \mathbb{E}_{p_\theta(z)}\left[f_\theta(z)\right] &= \nabla_\theta \left[\int_z p_\theta(z) f_\theta(z) \mathrm{d}z\right] = \int_z \nabla_\theta \left[p_\theta(z) f_\theta(z)\right] \mathrm{d}z \\
&= \int_z f_\theta(z) \nabla_\theta p_\theta(z) \mathrm{d}z + \int_z p_\theta(z) \nabla_\theta f_\theta(z) \mathrm{d}z \\
&= \int_z f_\theta(z) \nabla_\theta p_\theta(z) \mathrm{d}z + \mathbb{E}_{p_\theta(z)}\left[\nabla_\theta f_\theta(z)\right]
\end{aligned}
$$

$$
\begin{aligned}
\nabla_\theta \mathbb{E}_{p_\theta(z)}\left[f_\theta(z)\right] &= \nabla_\theta \left[\int_z p_\theta(z) f_\theta(z) \mathrm{d}z\right] = \int_z \nabla_\theta \left[p_\theta(z) f_\theta(z)\right] \mathrm{d}z \\
&= \int_z f_\theta(z) \nabla_\theta p_\theta(z) \mathrm{d}z + \int_z p_\theta(z) \nabla_\theta f_\theta(z) \mathrm{d}z \\
&= \int_z f_\theta(z) \nabla_\theta p_\theta(z) \mathrm{d}z + \mathbb{E}_{p_\theta(z)}\left[\nabla_\theta f_\theta(z)\right]
\end{aligned}
$$

We now have an **extra term** that **prevents exchange** of the expectation of the gradient.

Indirect Sampling

We can sample from p_θ in two ways. We can sample **directly**:

$$x \sim p_\theta(x)$$

Or we could **sample from some auxiliary distribution** and apply a **smooth transform** so that the output is distributed according to p_θ

$$x = g_\theta(\varepsilon),\ \varepsilon \sim q(\varepsilon)$$

Indirect Sampling

We can sample from p_θ in two ways. We can sample **directly**:

$$x \sim p_\theta(x)$$

Or we could **sample from some auxiliary distribution** and apply a **smooth transform** so that the output is distributed according to p_θ

$$x = g_\theta(\varepsilon),\ \varepsilon \sim q(\varepsilon)$$

Reparameterization Trick

$$\nabla_\theta \mathbb{E}_{p_\theta(z)}[f_\theta(z)]$$

Reparameterization Trick

$$\nabla_\theta \mathbb{E}_{p_\theta(z)}[f_\theta(z)] = \nabla_\theta \mathbb{E}_{\varepsilon \sim q(\varepsilon)}[f_\theta(g_\theta(\varepsilon))]$$

Reparameterization Trick

$$\nabla_\theta \mathbb{E}_{p_\theta(z)}[f_\theta(z)] = \nabla_\theta \mathbb{E}_{\varepsilon \sim q(\varepsilon)}[f_\theta(g_\theta(\varepsilon))]$$

$$= \mathbb{E}_{\varepsilon \sim q(\varepsilon)}[\nabla_\theta f_\theta(g_\theta(\varepsilon))]$$

Reparameterization Trick

$$\nabla_\theta \mathbb{E}_{p_\theta(z)}[f_\theta(z)] = \nabla_\theta \mathbb{E}_{\varepsilon \sim q(\varepsilon)}[f_\theta(g_\theta(\varepsilon))]$$

$$= \mathbb{E}_{\varepsilon \sim q(\varepsilon)}[\nabla_\theta f_\theta(g_\theta(\varepsilon))]$$

$$\approx \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta f_\theta(g_\theta(\varepsilon_i)), \quad \varepsilon_i \sim q$$

Let's apply the **reparameterization trick** to this stochastic program

$$z \sim \mathcal{N}(\theta, 4)$$
$$y = \theta^2(4 + z)$$

Let's apply the **reparameterization trick** to this stochastic program

$$\varepsilon \sim \mathcal{N}(0, 1)$$
$$z = \theta + 2\varepsilon$$
$$y = \theta^2(4 + z)$$

Let's apply the **reparameterization trick** to this stochastic program

$$\varepsilon \sim \mathcal{N}(0, 1)$$
$$y = \theta^2(4 + \theta + 2\varepsilon)$$

Let's apply the **reparameterization trick** to this stochastic program

$$\varepsilon \sim \mathcal{N}(0, 1)$$
$$y = \theta^2(4 + \theta + 2\varepsilon)$$

It's now easy to **compute the gradient**

$$\mathbb{E}_{\epsilon \sim \mathcal{N}(0,1)} \left[\nabla_\theta \theta^2(4 + \theta + 2\epsilon) \right] = 8\theta + 3\theta^2$$

Discrete Randomness

The reparameterization trick relies on a **smooth transformation**

This is fine for most continuous distributions but **not for discrete ones**

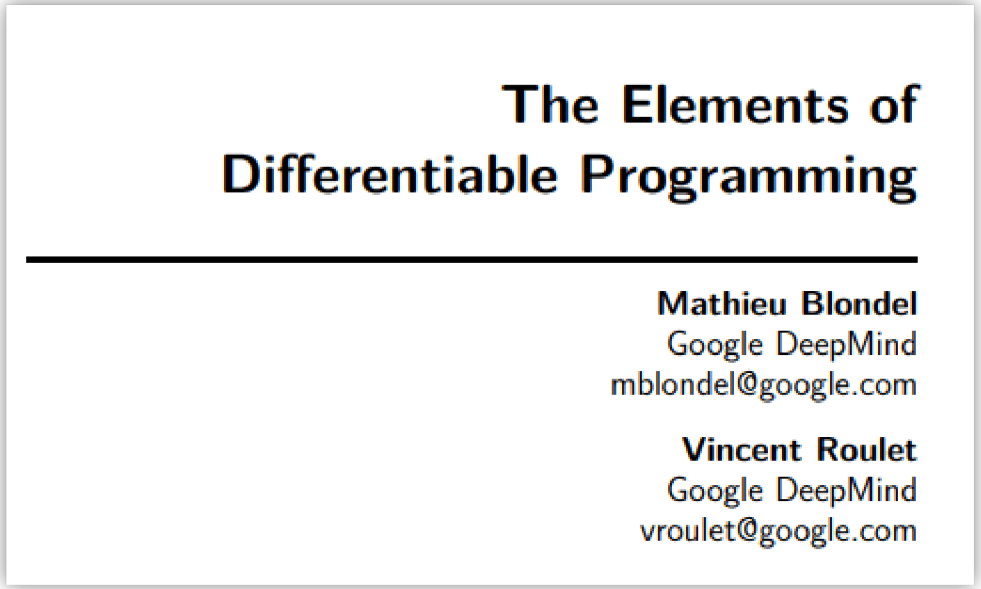
Thus we need **alternative tricks** to deal with discrete distributions

Discrete Randomness

The reparameterization trick relies on a **smooth transformation**

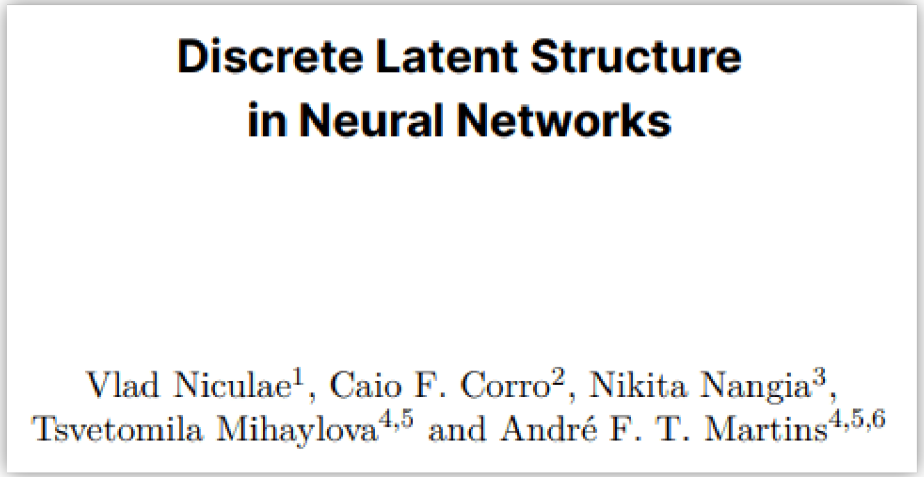
This is fine for most continuous distributions but **not for discrete ones**

Thus we need **alternative tricks** to deal with discrete distributions


The Elements of Differentiable Programming

Mathieu Blondel
Google DeepMind
mblondel@google.com

Vincent Roulet
Google DeepMind
vroulet@google.com


Discrete Latent Structure in Neural Networks

Vlad Niculae[1], Caio F. Corro[2], Nikita Nangia[3], Tsvetomila Mihaylova[4,5] and André F. T. Martins[4,5,6]

Gumbel-Max

Consider the categorical distribution $\mathrm{Cat}(\pi)$ with parameters

$$\pi = (\pi_1, \ldots, \pi_n)$$

We can reparametrize the this distribution with the **Gumbel distribution**

$$\arg\max_i (g_i + \log \pi_i) \sim \mathrm{Cat}(\pi), \quad g_i \sim \mathrm{Gumbel}(0)$$

Unfortunately, $\arg\max$ is **not differentiable**

Gumbel-Softmax

To fix this, we can replace $\arg\max$ with the $\mathrm{softmax}$ operator

$$y_i = \frac{\exp((\log \pi_i + g_i)/\tau)}{\sum_{j=1}^{n} \exp((\log \pi_j + g_j)/\tau)} \qquad g_i \sim \mathrm{Gumbel}(0)$$

τ is a free parameter called the **temperature**

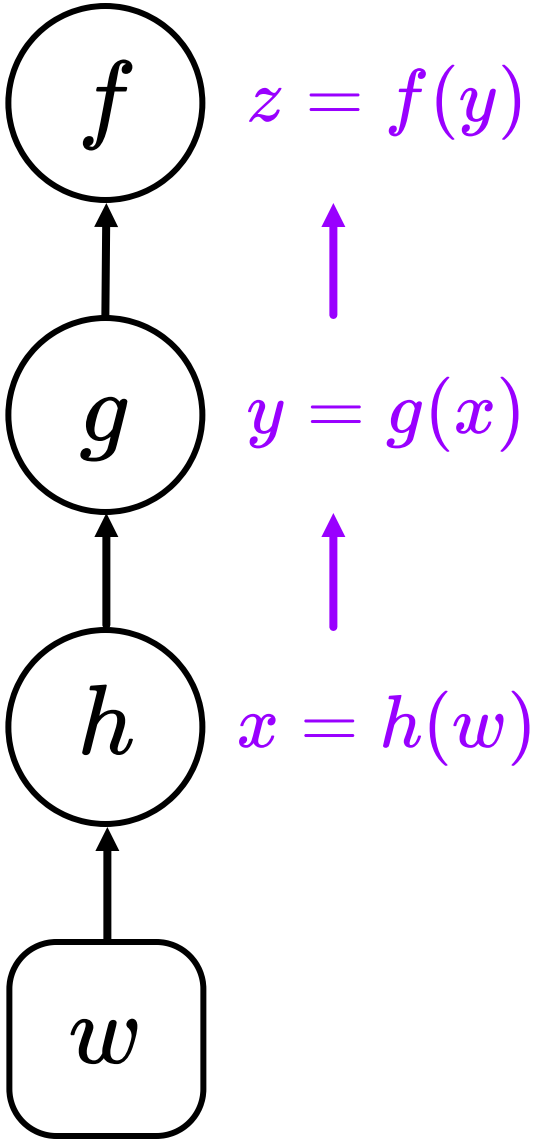
The Temperature

As the **temperature** goes to zero the Gumbel-Softmax distribution converges to the true Categorical distribution, so the **bias goes to zero**.

However, the **variance** of the resulting gradient estimator also **diverges to infinity**.

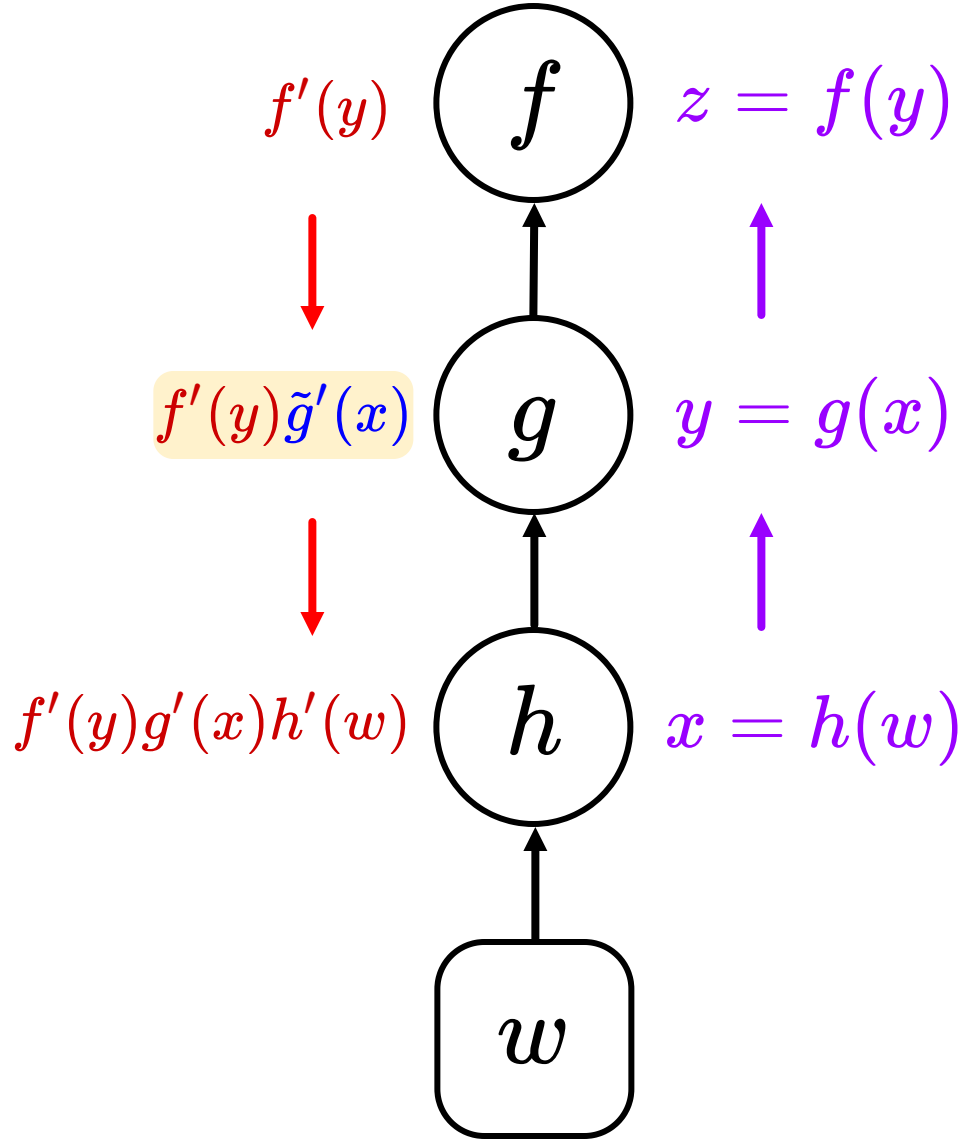
In other words, we have a **bias-variance trade-off.**

The GS trick is an example of a **surrogate gradient** method



The GS trick is an example of a **surrogate gradient** method

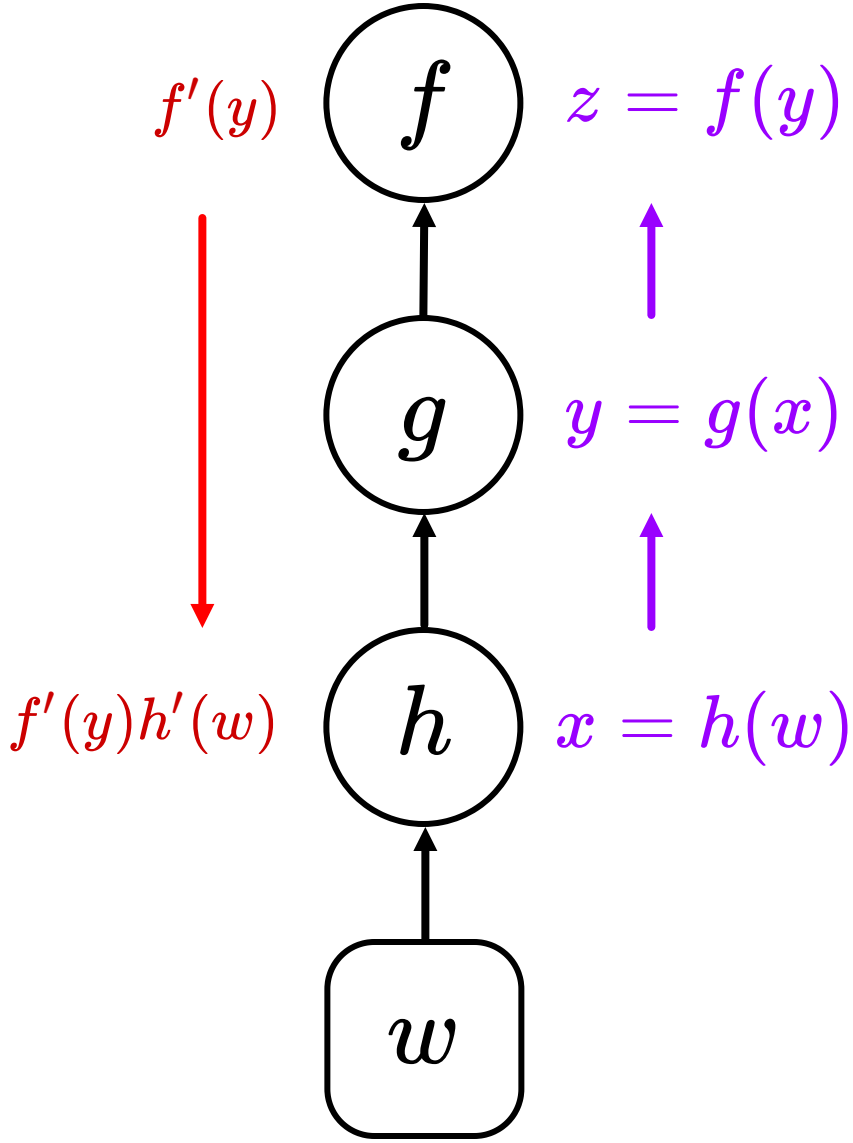
Idea: Replace a non-differentiable function with a **smooth approximation** when applying the chain rule



The GS trick is an example of a **surrogate gradient** method

Idea: Replace a non-differentiable function with a **smooth approximation** when applying the chain rule

Choosing the identity gives the **straight-through** gradient estimator



Agenda

A Brief Overview of Automatic Differentiation

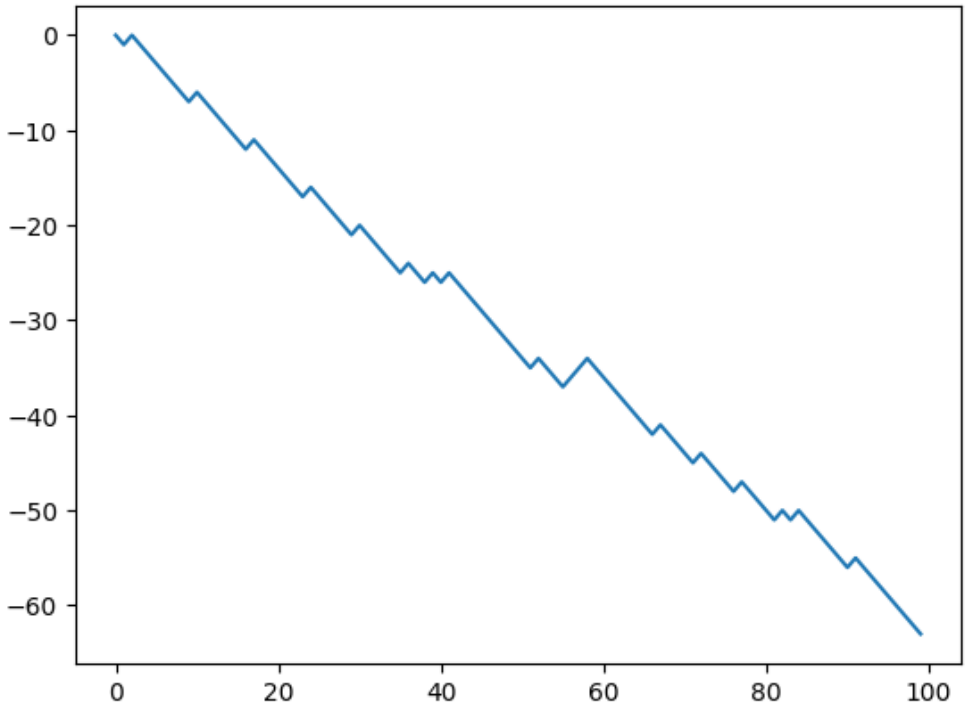
Differentiating Through Discrete and Stochastic Programs

Building Differentiable Programs and ABMs

Generalised Variational Inference with Differentiable ABMs

Random Walk

$$x_{t+1} = x_t + \begin{cases} 1 & \text{if } \xi = 1 \\ -1 & \text{if } \xi = 0 \end{cases}, \quad \xi \sim \text{Bern}(\theta)$$



Agenda

A Brief Overview of Automatic Differentiation

Differentiating Through Discrete and Stochastic Programs

Building Differentiable Programs and ABMs

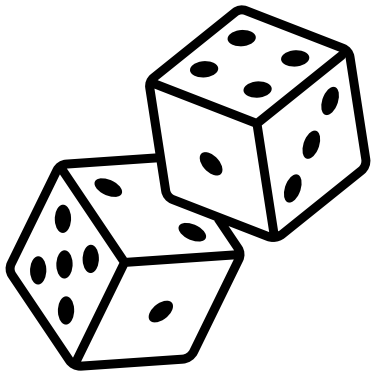
Generalised Variational Inference with Differentiable ABMs

Uncertainty Quantification

Traditional calibration methods return a **single point** from the parameter space

Often we **do not have enough data** to recover ABM parameters exactly

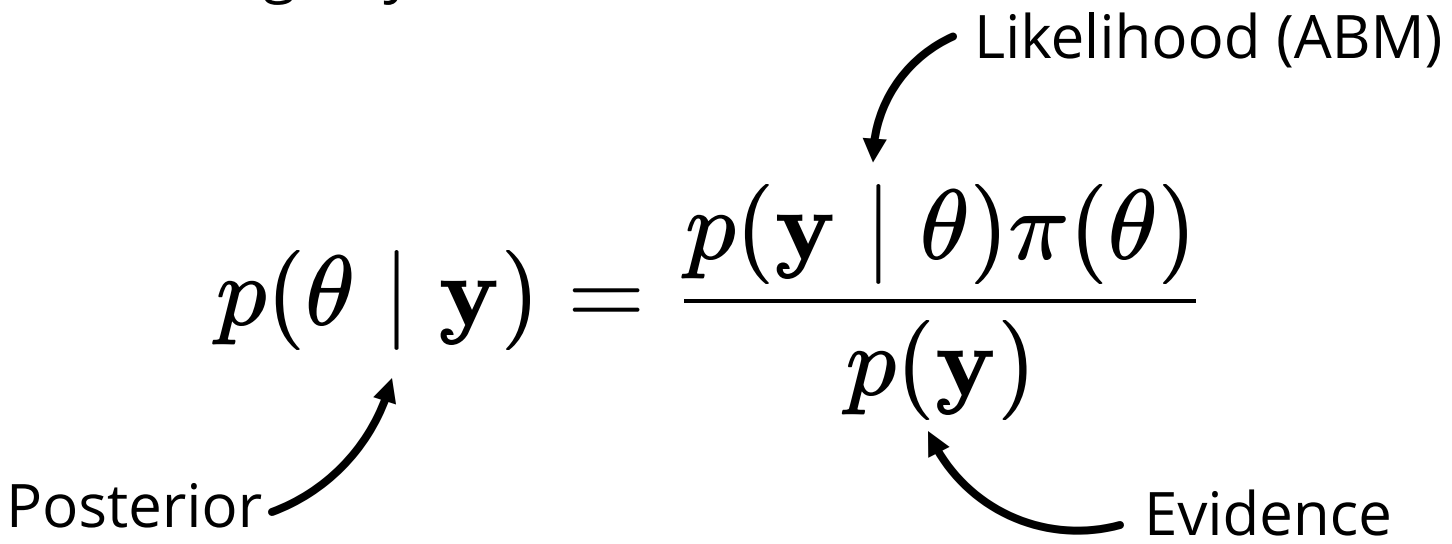
We want a method that naturally **quantifies our uncertainty** over ABM parameters



Bayesian Inference

Specify our initial beliefs with a **prior** $\pi(\theta)$ over the parameter space

Update our beliefs using Bayes rule



Variational Inference

The evidence is **intractable** to compute

To circumvent this, we can approximate the posterior with a **variational family**

$$\min_{\phi} \mathrm{D}_{\mathrm{KL}}(q_\phi(\theta), p(\theta \mid \mathbf{y}))$$

Since we don't have access to the posterior we optimise a **variational lower bound** instead

$$\min_{\phi} \mathbb{E}_{\theta \sim q_\phi(\theta)}[\log p(\mathbf{y} \mid \theta)] - \mathrm{D}_{\mathrm{KL}}(q_\phi, \pi)$$

GVI

$$\min_{\phi} \mathbb{E}_{\theta \sim q_\phi(\theta)}[\log p(\mathbf{y} \mid \theta)] - \mathrm{D}_{\mathrm{KL}}(q_\phi, \pi)$$

Evaluating the likelihood is generally **intractable in the case of ABMs**

To address this, we can **replace the log likelihood with a loss function** capturing our belief of how well the parameters match the data

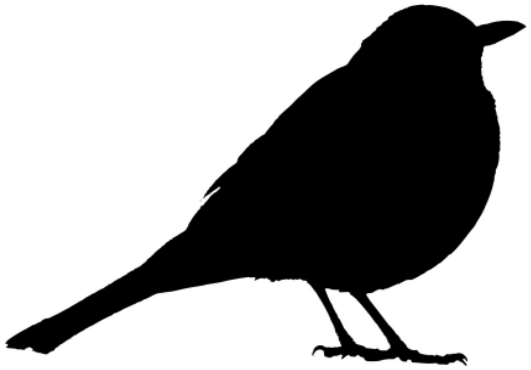
GVI

$$\min_{\phi} \mathbb{E}_{\theta \sim q_\phi(\theta)}[\ell(\mathbf{y}, \theta)] - \mathrm{D}_{\mathrm{KL}}(q_\phi, \pi)$$

Evaluating the likelihood is generally **intractable in the case of ABMs**

To address this, we can **replace the log likelihood with a loss function** capturing our belief of how well the parameters match the data

This is called **generalised variational inference**



BlackBIRDS

https://github.com/arnauqb/blackbirds

Volatility Clustering

Population of N agents trading a single asset over T time steps

Agents are essentially trying to forecast the logarithmic return r_t

$$r_t = \log S_t / S_{t-1}$$

Asset price on previous time step

Each agent receives a **common signal** on each time step about log returns

$$\epsilon_t \sim \mathcal{N}(0, \sigma^2)$$

Free parameter

Volatility Clustering

Using their common signal, each agent i makes an order based on their own personal threshold

$$\rho_i(t) = \mathbb{I}_{\epsilon_t > \nu_i(t)} - \mathbb{I}_{\epsilon_t < -\nu_i(t)}$$

Order

Personal threshold

$\rho_i(t)$	Order
-1	**Sell**
0	**Inactive**
$+1$	**Buy**

Volatility Clustering

The true return updates as follows

$$r_t = \frac{\sum_{i=1}^{N} \rho_i(t)}{N\eta}$$

Return

Free Parameter

Agents update their thresholds at random

$$\nu_i(t) = \mathbb{I}_{u_i(t)<1/10}|r_t| + \mathbb{I}_{u_i(t)\geq 1/10}\nu_i(t-1)$$

Uniform random variable