Sequence-Space Jacobian meets Deep Learning: Exploiting the Random Walk for HANK

Hanno Kase[1], Rodolfo Rigato[1], Matthias Rottner[3]

SEM, August 2, 2024

[1]European Central Bank
[2]Bank for International Settlements, Deutsche Bundesbank

Motivation

Challenges in Bayesian Estimation of Complex Models

- Time-consuming process limits empirical exploration and usefulness.
- Restricts the estimation of certain parameters, particularly those requiring the resolution of the model's steady state and the recalculation of Jacobians.

Combining innovations to speed things up

- **Approximate Posterior**: Utilize a DNN to approximate the true posterior.
- **Efficient Training Data**: Employ a (parallel) Metropolis-Hastings algorithm to generate training data and explore relevant regions of the parameter space.
- **Waste Recycling**: Use all generated draws.

Metropolis-Hastings

Objective: Sample from a target distribution $\pi(x)$ when direct sampling is difficult.

Steps:

1. **Initialization:** Choose an initial value x_0, number of samples T set $t = 0$.
2. **Proposal:** Generate a candidate x^* from a proposal distribution $q(x^*|x_t)$.
3. **Acceptance Criterion:**
   - Compute the acceptance probability:
   $$\alpha = \min\left(1, \frac{\pi(\mathbf{x}^*)q(x_t|x^*)}{\pi(x_t)q(x^*|x_t)}\right)$$
   - Accept or reject the candidate:
     - Accept x^* with probability α: set $x_{t+1} \leftarrow x^*$.
     - Otherwise, set $x_{t+1} \leftarrow x_t$.
4. **Iterate:** Increment t and repeat from Step 2 until $t = T$

You CAN have your $\pi(x)$ and eat it!

- Problem: To compute $\pi(x^*)$ we usually need to solve the model again

You CAN have your $\pi(x)$ and eat it!

- Problem: To compute $\pi(x^*)$ we usually need to solve the model again
  Solution: We can use a deep neural network to approximate $\pi(x^*)$

You CAN have your $\pi(x)$ and eat it!

- Problem: To compute $\pi(x^*)$ we usually need to solve the model again
  Solution: We can use a deep neural network to approximate $\pi(x^*)$
- Problem: How to create the training data?

You CAN have your $\pi(x)$ and eat it!

- Problem: To compute $\pi(x^*)$ we usually need to solve the model again
  Solution: We can use a deep neural network to approximate $\pi(x^*)$
- Problem: How to create the training data?
  Solution: Use the Metropolis-Hastings algorithm

You CAN have your $\pi(x)$ and eat it!

- Problem: To compute $\pi(x^*)$ we usually need to solve the model again
  Solution: We can use a deep neural network to approximate $\pi(x^*)$
- Problem: How to create the training data?
  Solution: Use the Metropolis-Hastings algorithm
- Problem: Only 20-30% of the proposals end up being accepted

You CAN have your $\pi(x)$ and eat it!

- Problem: To compute $\pi(x^*)$ we usually need to solve the model again
  Solution: We can use a deep neural network to approximate $\pi(x^*)$
- Problem: How to create the training data?
  Solution: Use the Metropolis-Hastings algorithm
- Problem: Only 20-30% of the proposals end up being accepted
  Solution: We can use both rejected and accepted candidates as training data

Simple HANK: Environment

Model:

- One asset HANK model with sticky wages, three aggregate shocks
- Solved using Sequence-Space Jacobian toolkit (Auclert et al. 2021)
  - Computing the steady state and household Jacobians takes **1.0s**
  - Soving the model and computing the log-likelihood takes **0.7s**

Estimation:

- US data from 1966 to 2019
  - Three time-series: GDP growth, GDP deflator, Fed funds rate
- Estimate 11 parameters
  - Metropolis-Hastings + Deep learning
  - Multi-proposal Metropolis-Hastings + Deep learning

Metropolis-Hastings + Deep Learning

Steps:

1. Find posterior mode using a solver
2. Sample from the posterior using the Metropolis-Hastings
   - Store candidates x_i^* and log-posteriors $\log \pi(x_i^*)$, $i = 1, 2, ..., N$
3. Shuffle and split into training and validation samples
4. Train a deep neural network to approximate $\log \hat{\pi}(x^*) = \Psi_{DNN}(x^*)$
   - Supervised training to minimize loss $\mathcal{L} = \frac{1}{B} \sum_{i=1}^{B} \left(\log \pi(x_i) - \log \hat{\pi}(x_i)\right)^2$
   - Fully connected feed-forward neural network
     - 3 hidden layers, 128 neurons each, CELU activation function
   - trained for 1000 epochs, $B = 100$
     - AdamW and cosine annealing learning rate scheduler
5. Sample from posterior using Metropolis-Hastings and $\Psi_{DNN}(x)$
   - Very fast sampling at a rate of **10 000 it/second**

Approximate Posterior $\Psi_{DNN}(x)$

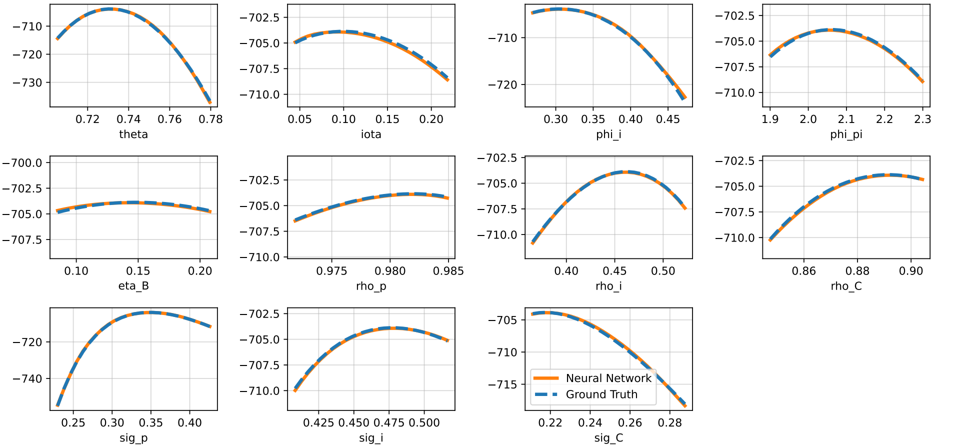

Figure 1: True and approximate log posterior for different parameters

Comparing posterior distributions

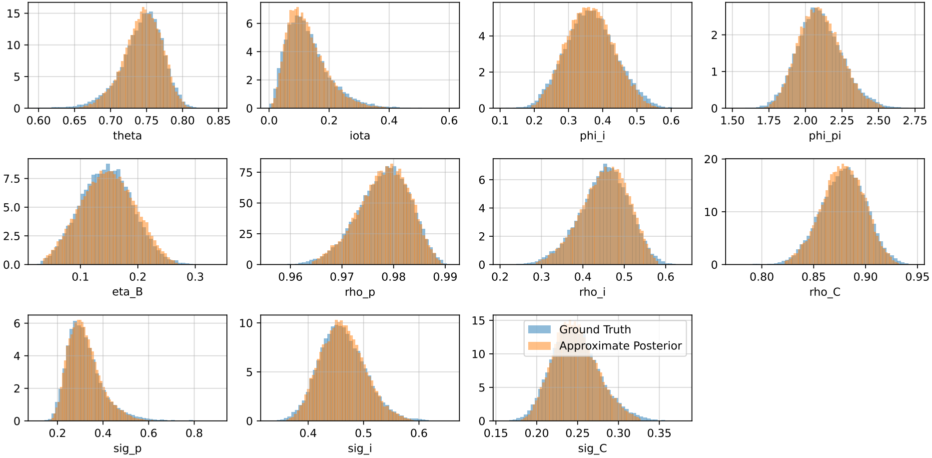

Figure 2: True and approximate posterior distributions for different parameters

How much faster?

- Standard Metropolis-Hastings for 200 000 samples: **39h**

How much faster?

- Standard Metropolis-Hastings for 200 000 samples: **39h**
- Generating training data 50 000 samples: **~9h 45min**
- Training the neural network: **~10min**
- Deep Learning Metropolis-hastings for 200 000 samples: **20sec**

How much faster?

- Standard Metropolis-Hastings for 200 000 samples: **39h**
- Generating training data 50 000 samples: **~9h 45min**
- Training the neural network: **~10min**
- Deep Learning Metropolis-hastings for 200 000 samples: **20sec**

Saved about 29 hours!

How much faster?

- Standard Metropolis-Hastings for 200 000 samples: **39h**
- Generating training data 50 000 samples: **~9h 45min**
- Training the neural network: **~10min**
- Deep Learning Metropolis-hastings for 200 000 samples: **20sec**

Saved about 29 hours!

- Virtually identical results:
  - Probability of wrong accept, $\max\{\hat{\alpha} - \alpha, 0\}$: **0.38%**
  - Probability of wrong reject, $\max\{\alpha - \hat{\alpha}, 0\}$: **0.42%**

How much faster?

- Standard Metropolis-Hastings for 200 000 samples: **39h**
- Generating training data 50 000 samples: **~9h 45min**
- Training the neural network: **~10min**
- Deep Learning Metropolis-hastings for 200 000 samples: **20sec**

Saved about 29 hours!

- Virtually identical results:
  - Probability of wrong accept, $\max\{\hat{\alpha} - \alpha, 0\}$: **0.38%**
  - Probability of wrong reject, $\max\{\alpha - \hat{\alpha}, 0\}$: **0.42%**
- Estimation with solving for the deterministic steady state would take ~20h

Parallel Multi-proposal Metropolis-Hastings + Deep Learning

🗽 Free to choose the flavor of Metropolis-Hastings.

- Standard Metropolis-Hastings: **39h**
- Multi-proposal Metropolis-Hastings (Calderhead 2014 or Schwedes et al. 2021)
    - 64 CPU (AMD EPYC 7V12): **1h 57min**
    - Macbook M1 Pro (6P+2E): **7h**

The main idea of the multi-proposal algorithms:

- Generate multiple candidates x_i^* where $i = 1, 2, ..., N_p$
- In parallel compute $\pi(x_i^*)$ for $i = 1, 2, ..., N_p$
    a. Either construct a transition matrix and simulate a Markov chain for N_{draws}
    b. Construct a distribution $\left(\frac{\pi(x_1^*)}{\sum_{i=1}^{N_p}(\pi(x_i^*)}, ..., \frac{\pi(x_{N_p}^*)}{\sum_{i=1}^{N_p}(\pi(x_i^*)}\right)$ to sample $(x_1^*, ..., x_{N_p}^*)$

Sample generated by the multi-proposal Metropolis-Hastings algorithm

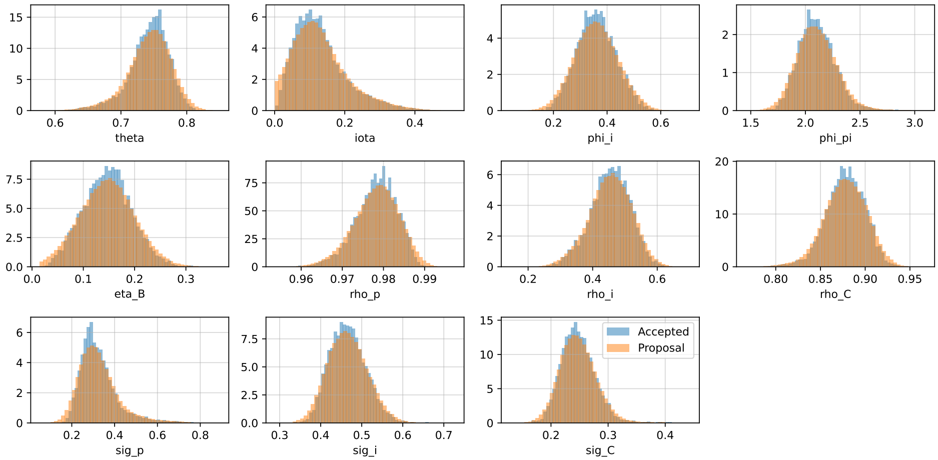

Figure 3: Distribution of proposed and accepted draws from the multi-proposal Metropolis-Hastings algorithm

Multi-proposal Metropolis-Hastings + Deep Learning

Steps:

1. Find posterior mode using a solver
2. Sample from the posterior using the multi-proposal Metropolis-Hastings
    - Store candidates x_i and posterior density $\pi(x_i)$, $i = 1, 2, ..., N$
    - Drop 20% of candidates with low posterior density
        - Some candidates have a very low likelihood
        - Leaving them out compresses the range of values and eases training
        - Wouldn't matter for the final posterior distribution
3. Shuffle and split into training and validation samples
4. Train a deep neural network to approximate $\log \hat{\pi}(x) = \Psi_{DNN}(x)$
    - Same configuration as before...
5. Sample from posterior using Metropolis-Hastings and $\Psi_{DNN}(x)$
    - Very fast sampling at a rate of **10 000 it/second**

Approximate Posterior $\Psi_{DNN}(x)$

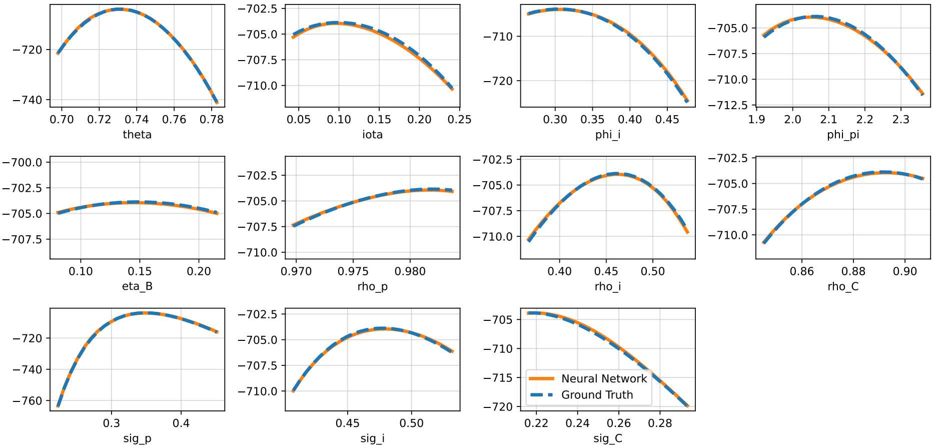

Figure 4: True and approximate log posterior for different parameters

Comparing posterior distributions

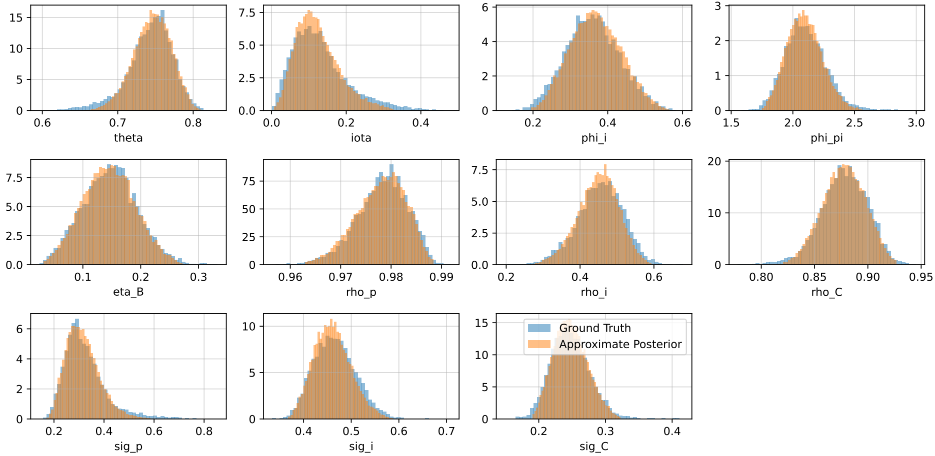

Figure 5: True and approximate posterior distributions for different parameters

More speed! 🏎️

- Generating training data 50 000 samples:
  - On a laptop: **~1h 45min**
  - On a 64 core server CPU: **~29min**
- Training the neural network: **~10min**
- Deep Learning Metropolis-hastings for 200 000 samples: **20sec**

More speed! 🏎️

- Generating training data 50 000 samples:
  - On a laptop: **~1h 45min**
  - On a 64 core server CPU: **~29min**
- Training the neural network: **~10min**
- Deep Learning Metropolis-hastings for 200 000 samples: **20sec**

Estimate a simple HANK in 40min (or 2h on a laptop)!

More speed! 🏎️

- Generating training data 50 000 samples:
  - On a laptop: **~1h 45min**
  - On a 64 core server CPU: **~29min**
- Training the neural network: **~10min**
- Deep Learning Metropolis-hastings for 200 000 samples: **20sec**

Estimate a simple HANK in 40min (or 2h on a laptop)!

- Almost identical results:
  - Probability of wrong accept, $\max\{\hat{\alpha} - \alpha, 0\}$: **0.33%**
  - Probability of wrong reject, $\max\{\alpha - \hat{\alpha}, 0\}$: **0.42%**

More speed! 🏎️

- Generating training data 50 000 samples:
  - On a laptop: **~1h 45min**
  - On a 64 core server CPU: **~29min**
- Training the neural network: **~10min**
- Deep Learning Metropolis-hastings for 200 000 samples: **20sec**

Estimate a simple HANK in 40min (or 2h on a laptop)!

- Almost identical results:
  - Probability of wrong accept, $\max\{\hat{\alpha} - \alpha, 0\}$: **0.33%**
  - Probability of wrong reject, $\max\{\alpha - \hat{\alpha}, 0\}$: **0.42%**
- Estimation with solving for the deterministic steady state would take ~1.5h

Conclusion

By combining these innovations

- Approximate Posterior: Utilize a DNN to approximate the true posterior.
- Efficient Training Data: Employ a (parallel) Metropolis-Hastings algorithm to generate training data and explore relevant regions of the parameter space.
- Waste Recycling: Use all generated draws.

We can speed up the estimation of complex macroeconomic models significantly!

Conclusion

By combining these innovations

- **Approximate Posterior**: Utilize a DNN to approximate the true posterior.
- **Efficient Training Data**: Employ a (parallel) Metropolis-Hastings algorithm to generate training data and explore relevant regions of the parameter space.
- **Waste Recycling**: Use all generated draws.

We can speed up the estimation of complex macroeconomic models significantly!

- Future directions:
  - Applications: Quantitative HANK, forecasting
  - Approximate household Jacobians?

Thank you!

References

Auclert, A., B. Bardóczy, M. Rognlie, and L. Straub. 2021. “Using the Sequence-Space Jacobian to Solve and Estimate Heterogeneous-Agent Models.” *Econometrica* 89 (5): 2375–2408.

Calderhead, B. 2014. “A General Construction for Parallelizing Metropolis-Hastings Algorithms.” *Proceedings of the National Academy of Sciences* 111 (49): 17408–17413.

Schwedes, T., and B. Calderhead. 2021. “Rao-Blackwellised Parallel MCMC.” In *Proceedings of The 24th International Conference on Artificial Intelligence and Statistics*, 3448–3456. PMLR.

Neural Network

- Single neuron i in layer l with width H_l and activation function σ:

$$x_i^l = \sigma\left(\sum_j W_{ij}^l x_j^{l-1} + b_i^l\right), \quad 1 \leq i \leq H_l, \quad 1 \leq j \leq H_{l-1}$$

- Single layer:

$$\mathbf{x}^l = \sigma\left(\mathcal{A}^l(\mathbf{x}^{l-1})\right), \quad \mathcal{A}^l\left(\mathbf{x}^{l-1}\right) = \mathbf{W}^l\mathbf{x}^{l-1} + \mathbf{b}^l$$

- The entire network with L hidden layers:

$$\psi(\mathbf{x}) = \left(\mathcal{A}^{L+1} \circ \sigma \circ \mathcal{A}^L \circ \sigma \circ \mathcal{A}^{L-1} \circ \ldots \circ \sigma \circ \mathcal{A}^1\right)(\mathbf{x})$$

- Weights and biases of the network:

$$\theta = \{\mathbf{W}^l, b^l\}_{l=1}^{L+1}$$

Training a Neural Network

- Suppose we want to approximate $\mathbf{f}: \mathbf{x} \mapsto \mathbf{y}$ using our neural network $\psi(\mathbf{x}; \theta)$
- We have a dataset of pairwise samples $\mathcal{S} = \{(\mathbf{x}_i, \mathbf{y}_i) : 1 \leq i \leq N\}$
- **Training** is adjusting θ so that $\psi(\mathbf{x}, \theta)$ starts approximating $\mathbf{f}$:

$$\theta^* = \arg\min_\theta \mathcal{L}(\theta), \quad \text{where} \quad \mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \psi(\mathbf{x}; \theta))$$

- Usually done using (some variation of) gradient descent algorithm:

$$\theta_{k+1} = \theta_k - \eta \frac{\partial \mathcal{L}}{\partial \theta}(\theta_k)$$

- Where η is the learning rate
- The gradients are efficiently calculated using the backpropagation algorithm