Neural State-Space models

Theory

Marco Forgione, Dalle Molle Institute for Artificial Intelligence, Lugano, Switzerland

Milano, July 3rd, 2025

State-space models

They are defined in continuous/discrete time by:

$$\dot{x}(t) = f(x(t), u(t); \theta)$$
$$y(t) = g(x(t); \theta)$$

$$x(k+1) = f(x(k), u(k); \theta)$$
$$y(k) = g(x(k); \theta)$$

- x is the state vector. It is a latent (=hidden, unobserved) quantity
- u is the input vector (=external/exogenous variable, covariate)
- y is the output vector
- θ is a vector of unknown parameters to be estimated

- Powerful and flexible
- Familiar to engineer/researcher
- Suitable to embed physics
- Might be harder to train than other sequence models

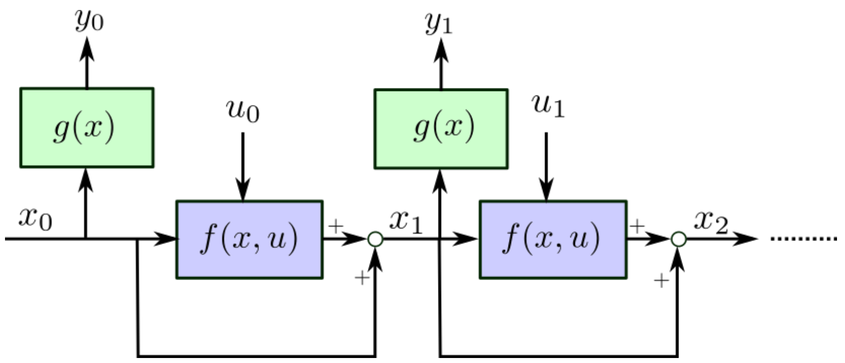
Neural state-space models

In our context, f and g are feed-forward neural networks, e.g.

$$x(k+1) = x(k) + f(x(k), u(k); \theta)$$
$$y(k) = g(x(k); \theta)$$

$$f(z;\theta) = W_3\sigma(W_2\sigma(W_1 z + b_1) + b_2) + b_3$$
$$z = [x\ u]^\top$$
$$\theta = \mathrm{vec}(W_1, W_2, W_3, b_1, b_2, b_3)$$

Overall, we define a **neural state-space** model, aka **recurrent neural network** (RNN)

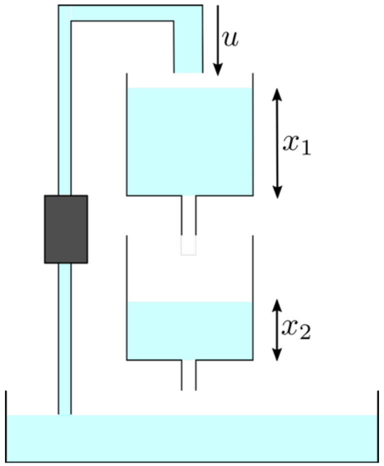


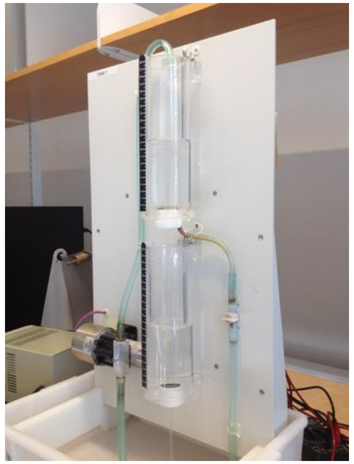
With respect to classic RNNs (Vanilla RNN, LSTM), focus on:

- Embedding prior (physical) knowledge
- Parsimonious representations suitable for control

Physics-inspired neural state-space models

Cascaded Tanks System. Input: upper tank inlet flow u. Output: lower tank level.





Schoukens, M. et al. "Cascaded tanks benchmark combining soft and hard nonlinearities." Workshop on nonlinear system identification benchmarks, 2016.

Intuitive physics:

- The system has two states: levels x_1, x_2
- The state x_2 is measured: $y = x_2$
- State x_1 does not depend on x_2
- State x_2 does not depend on u directly

Observations above are embedded in a **physics-inspired** neural state-space model:

$$\dot{x}_1 = f_1(x_1, u; \theta)$$
$$\dot{x}_2 = f_2(x_1, x_2; \theta)$$
$$y = x_2$$

Identification results in Forgione, M. and Piga, D. "[Continuous-time system identification with neural networks: Model structures and fitting criteria](#)." European Journal of Control, 2021

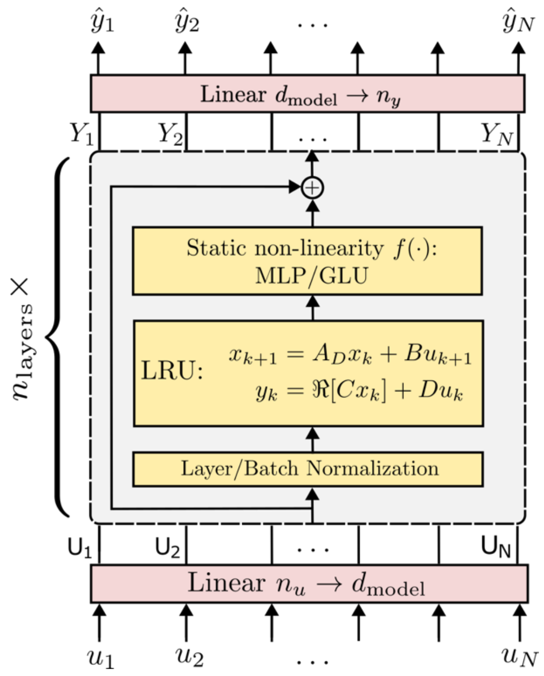
Physics-inspired neural state-space models

Several contributions aim at embedding **prior knowledge** in the architecture.

- (Port) Hamiltonian neural networks
  S. Greydanus et al., Hamiltonian Neural Networks, 2019
  S. Moradi et al., Port-Hamiltonian Neural Networks with Output Error Noise Models, 2025

- Lagrangian neural networks
  M. Cranmer et al., Lagrangian Neural Networks, 2020

- Structured state-space sequence models
  A. Gu et al., Efficiently Modeling Long Sequences with Structured State Spaces, 2022



From M. Forgione, M. Mejari, and D. Piga Model order reduction of deep structured state-space models: A system-theoretic approach, 2024

… and many, many more!

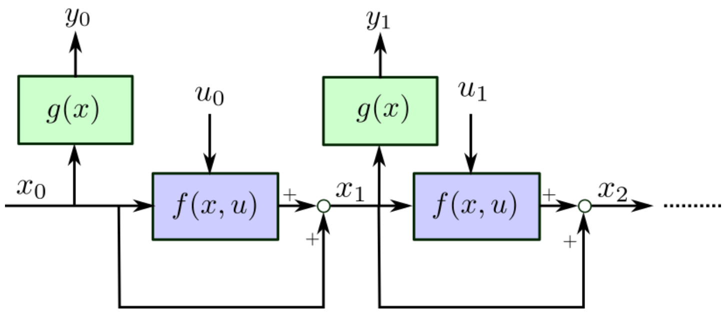
Simulation error minimization

In principle, **simulation error minimization** can be a valid approach:

$$\hat{\theta}, \hat{x}(0) = \arg\min_{\theta, x(0)} \frac{1}{T} \sum_{k=0}^{T-1} \left\| \hat{y}^{\mathrm{sim}}(k) - y(k) \right\|^2$$

$$x(k+1) = x(k) + f(x(k), u(k);\ \theta)$$
$$\hat{y}^{\mathrm{sim}}(k) = g(x(k);\ \theta)$$

- Minimization wrt both θ (model) and x_0 (initial state)
- It is related to maximum likelihood under certain hypotheses
- It favors learning of **long-term** dependencies
- We stick with it for the exercise session

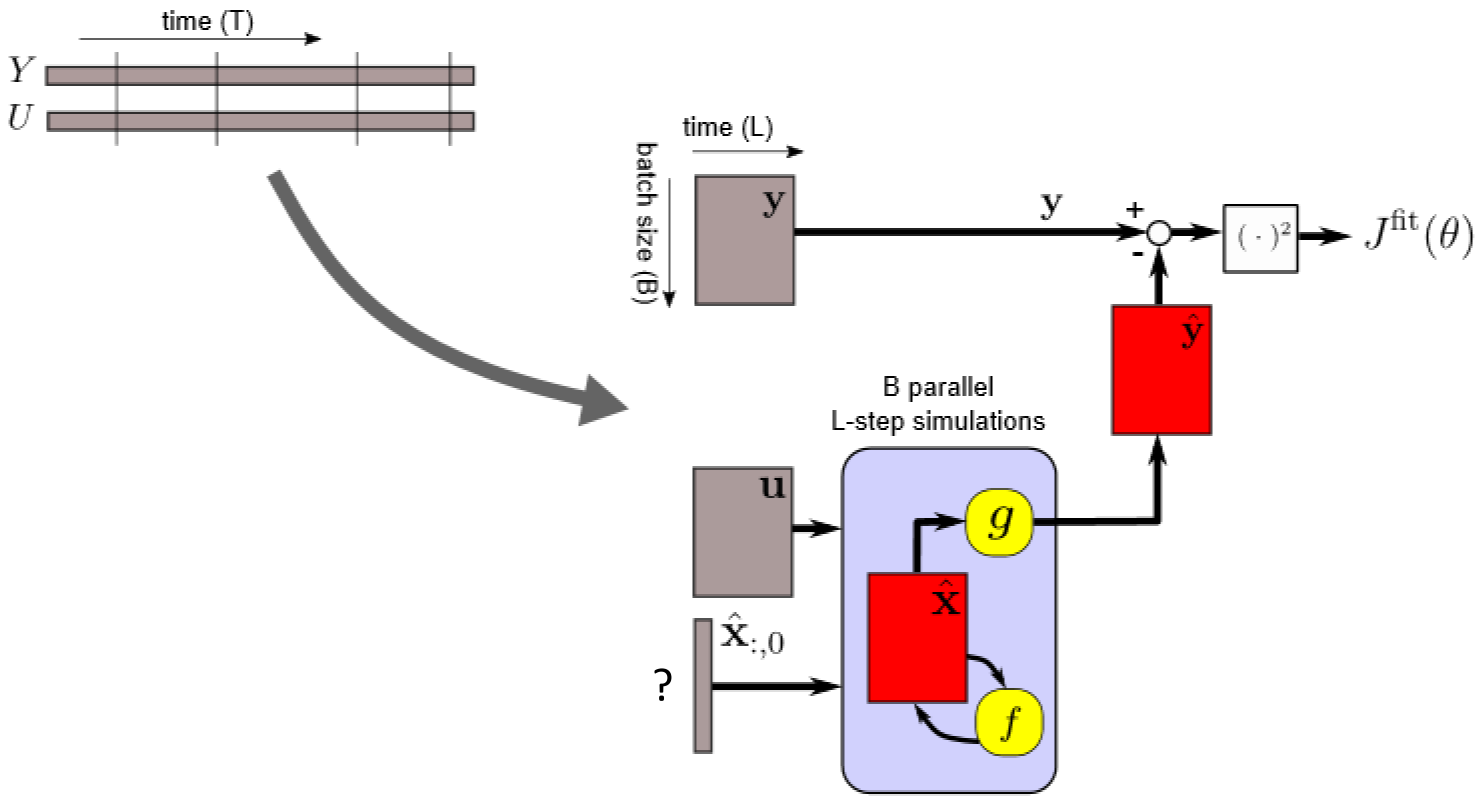


- It might be **computationally heavy**
  - Processing of a long sequence only gives one gradient update
- It might be **numerically hard**
  - Unstable/chaotic dynamics cannot be predicted on the long term
- It might not match the intended model usage
  - MPC only needs estimates up to the prediction horizon

For these reasons, alternative methods based on simulation on **shorter sub-sequences** have been developed

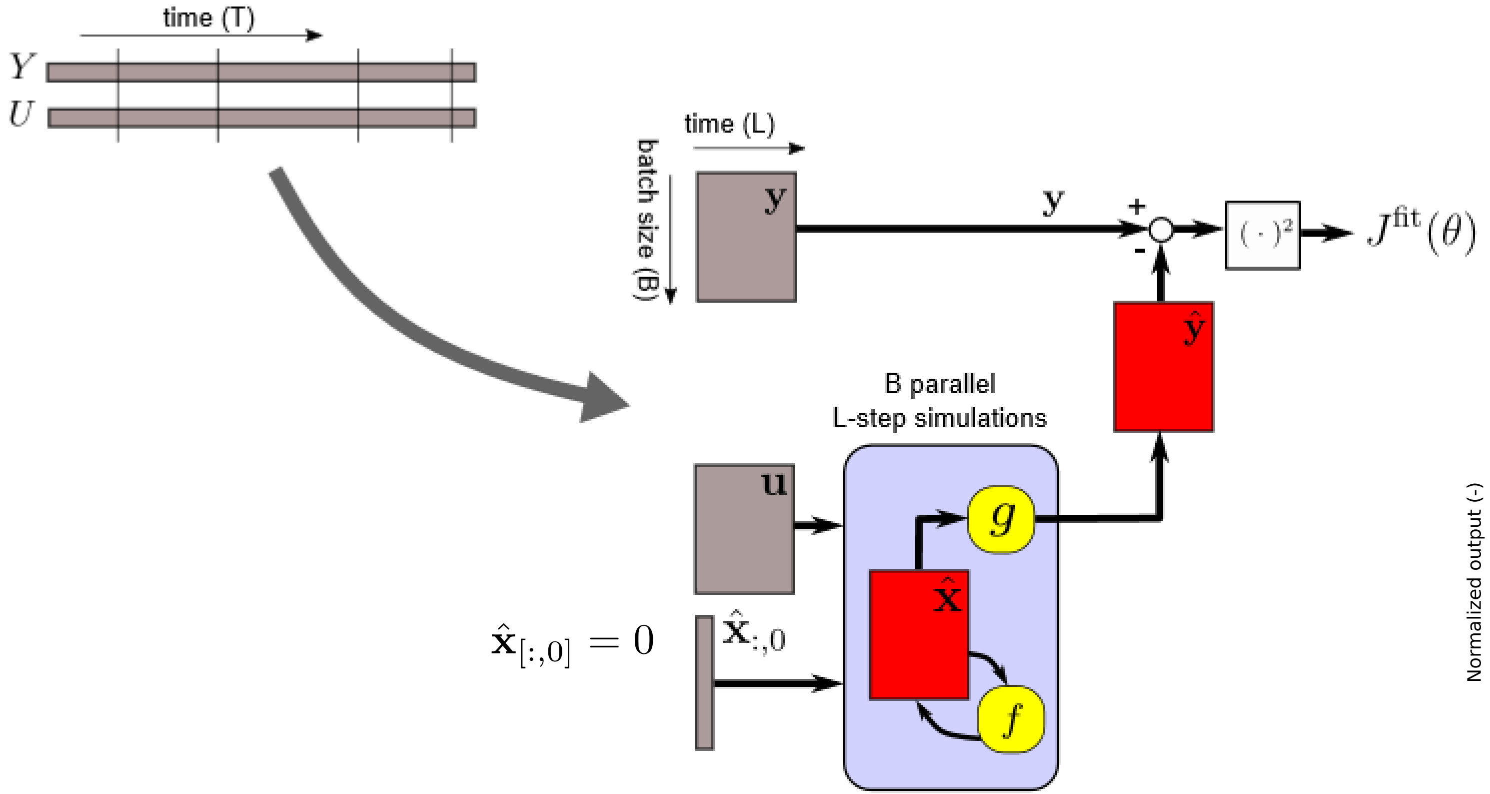
Truncated simulation error minimization

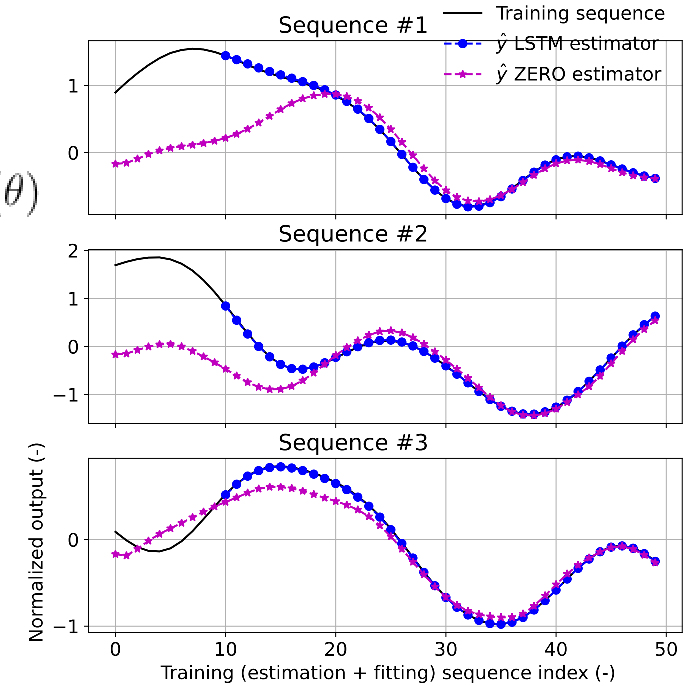
Different ***L*-step-ahead error minimization** schemes have been developed:



- Simulate B (batch size) shorter sequences of length L extracted from the longer (length T) training dataset.
- Main variation is: how do we deal with all possible initial conditions?

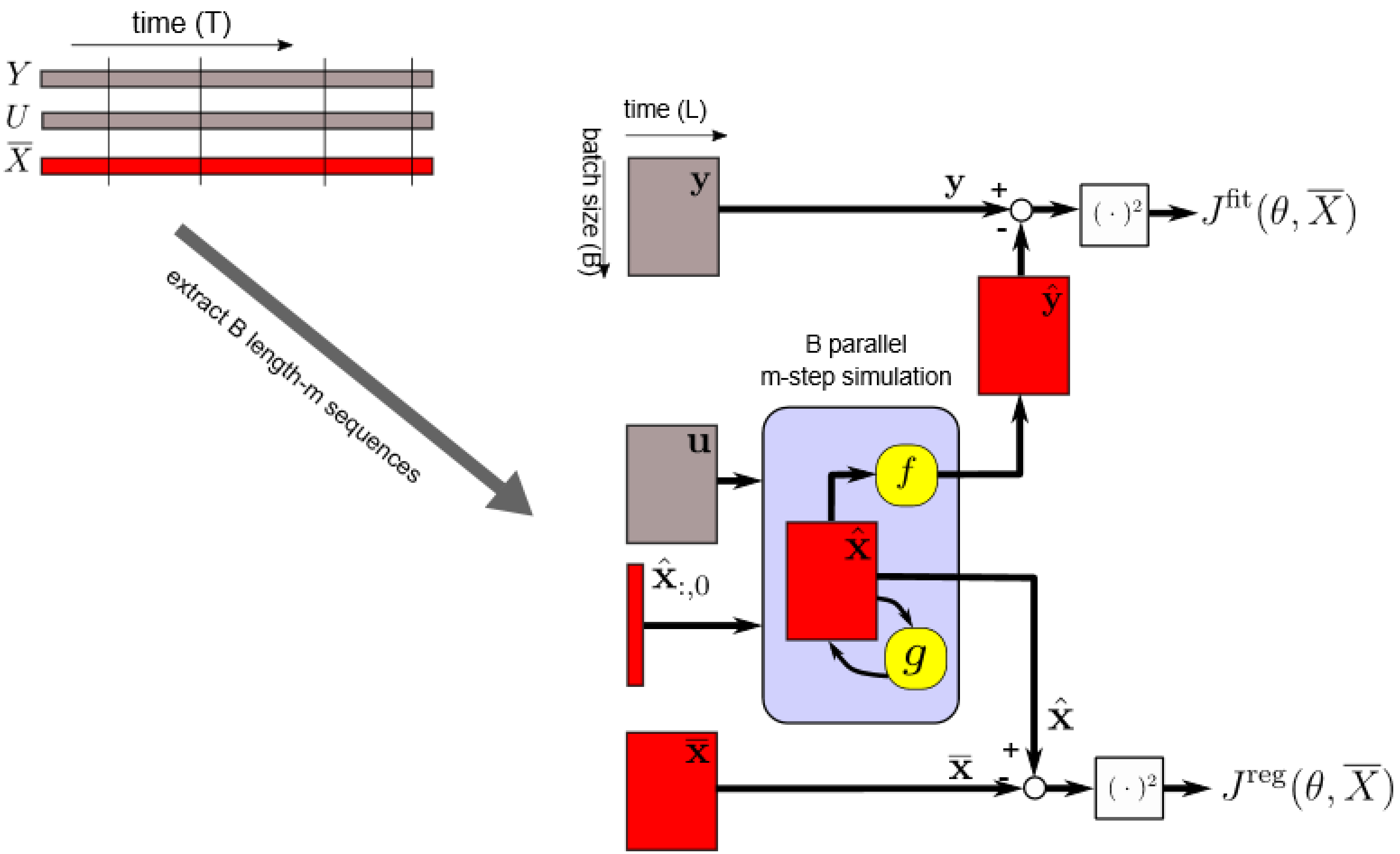
Zero initialization strategy





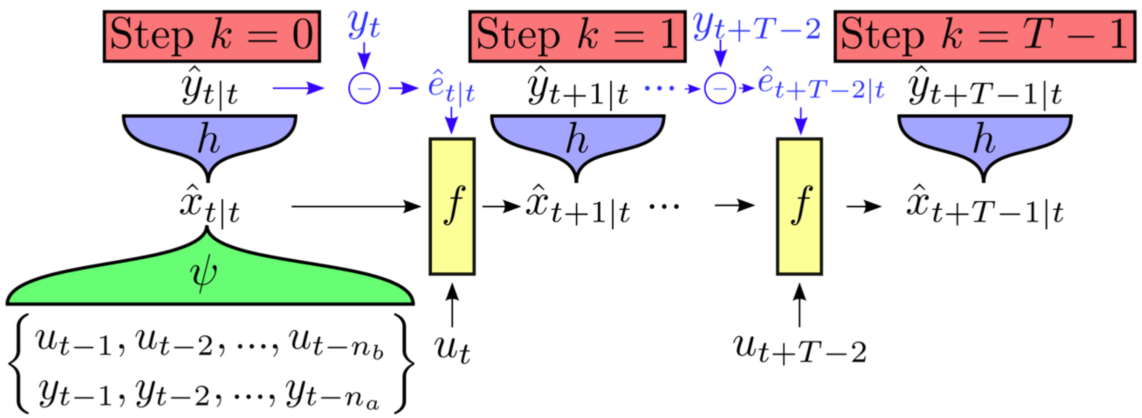
- Initialize from arbitrary (ZERO) state and discard a few initial samples from the loss
- Described as a baseline in M.Forgione, M.Mejari, and D.Piga, [Learning neural state-space models: do we need a state estimator?](), 2022
- Very simple, yet effective when the system memory is short (wrt sequence length m)

Learn the hidden state sequence

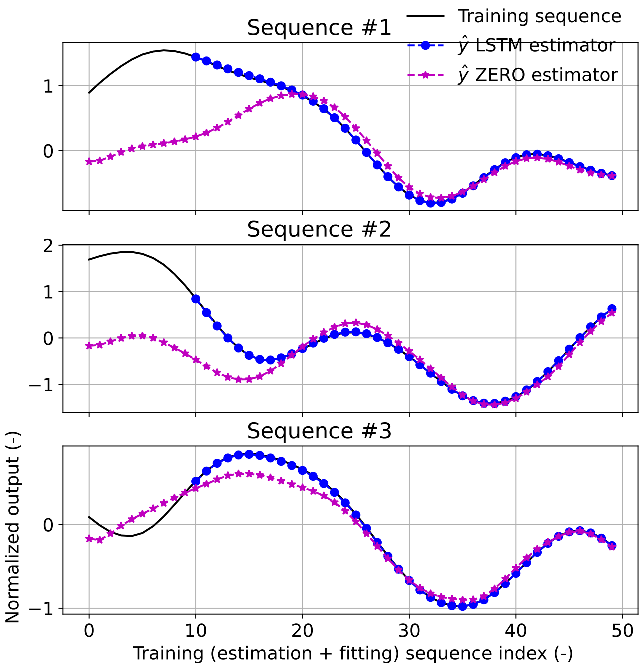


- Learn the sequence of states along with the system mappings f and g
- Regularize the learned state sequence with the systep mapping
- Described in M.Forgione and D.Piga, [Continuous-time system identification with neural networks](), 2021

Learn a state estimator along with the model (SUBNET)



From G. Beintema, M. Schoukens, R.Tóth Deep subspace encoders for nonlinear system identification, 2022



- Use the initial samples of each sequence to reconstruct the initial state with a (learned) state estimator
- See G. Beintema, M. Schoukens, R.Tóth Deep subspace encoders for nonlinear system identification, 2022

PyTorch Implementation

Unlike with standard RNNs, we must work at a slightly lower level

Custom state-update function $f(x, u)$

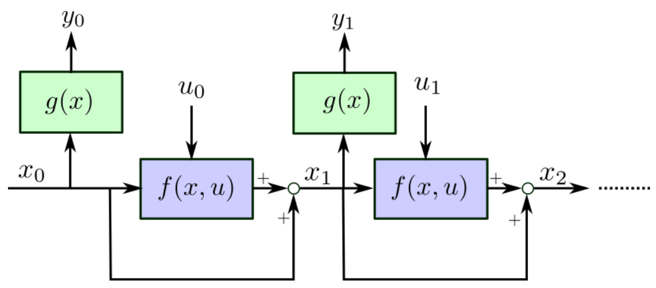
```
class NeuralStateUpdate(nn.Module):

    def __init__(self, n_x=2, n_u=1, n_feat=32):
        super(NeuralStateUpdate, self).__init__()

        self.net = nn.Sequential(
            nn.Linear(n_x+n_u, n_feat),
            nn.Tanh(),
            nn.Linear(n_feat, n_x),
        )

        for m in self.net.modules():
            if isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, mean=0, std=1e-2)
                nn.init.constant_(m.bias, val=0)

    def forward(self, x, u):
        z = torch.cat((x, u), dim=-1)
        dx = self.net(z)
        return dx
```

Custom code to unroll $f(x, u)$ over time

```
class StateSpaceSimulator(nn.Module):
    def __init__(self, f_xu):
        super().__init__()
        self.f_xu = f_xu

    def forward(self, x_0, u):
        B, n_x = x_0.shape
        _, T, _ = u.shape # B, T, n_u
        x = torch.empty((B, T, n_x))
        x_step = x_0

        # manually unroll f_xu over time
        for t in range(T):
            x[:, t, :] = x_step
            dx = self.f_xu(x_step, u[:, t, :])
            x_step = x_step + dx

        return x
```

Note:

- we actually implement: $x(k+1) = x(k) + f(x(k), u(k); \theta)$
- we make a custom weight initialization



Code adapted from https://github.com/forgi86/pytorch-ident

PyTorch Implementation

We can then instantiate a model that behaves very similarly to a standard PyTorch RNN

```
n_x = 2; n_u = 1;
f_xu = NeuralStateUpdate(n_x, n_u, n_feat=32)
simulator = StateSpaceSimulator(f_xu)
```

The model accepts a batch of initial states (B, n_x) and input sequences (B, T, n_u) and returns a batch of simulated states (B, T, n_x)

```
B, T = 32, 1024;
batch_x0 = torch.zeros((B, n_x))
batch_u = torch.randn((B, T, n_u)) # replace with actual training input
batch_x_sim = simulator(batch_x0, batch_u) # B, T, n_x
batch_x_sim.shape
```

```
torch.Size([32, 1024, 2])
```

We may finally process batch_x_sim through a feed-forward network to simulate the output equation $y = g(x)$. Alternatively, we may set the output to a subset of the simulated states.

Conclusions

- Neural state-space models are just custom RNNs
- Beyond standard Vanilla RNN, LSTM, GRU

Recent research has explored:

- State initialization for training with mini-batches
- Embedding of physical knowledge

We focus on physical knowledge in the exercises

Thank you for your attention