

# Deep Learning with Jax

Fitting neural state-space models

---

Marco Forgione

IDSIA USI-SUPSI, Lugano, Switzerland

## State-space models

We consider state-space models in the form:

$$\begin{aligned}x(k+1) &= f(x(k), u(k); p) \\y(k) &= g(x(k); p)\end{aligned}$$

- $x(k) \in \mathbb{R}^{n_x}$  is the state vector. Latent, hidden, unobserved variable
- $u(k) \in \mathbb{R}^{n_u}$  is the input vector. External, exogenous variable
- $y(k) \in \mathbb{R}^{n_y}$  is the output vector
- $p \in \mathbb{R}^{n_p}$  is the parameter vector

I don't need to convince you that they are general and powerful!

## State-space models

We consider state-space models in the form:

$$\begin{aligned}x(k+1) &= f(x(k), u(k); p) \\y(k) &= g(x(k); p)\end{aligned}$$

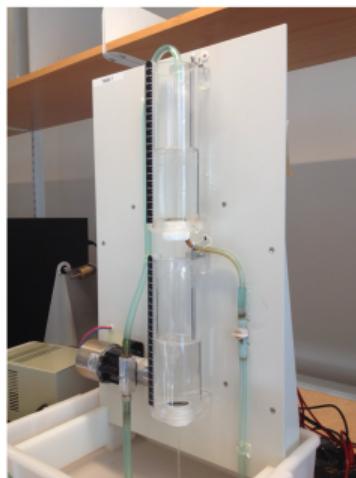
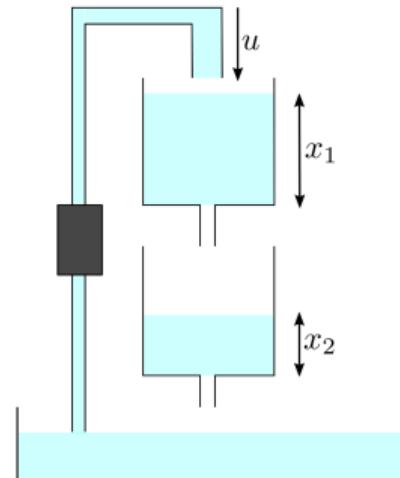
- $x(k) \in \mathbb{R}^{n_x}$  is the state vector. Latent, hidden, unobserved variable
- $u(k) \in \mathbb{R}^{n_u}$  is the input vector. External, exogenous variable
- $y(k) \in \mathbb{R}^{n_y}$  is the output vector
- $p \in \mathbb{R}^{n_p}$  is the parameter vector

I don't need to convince you that they are general and powerful!

- We will see how to fit state-space models to data using Jax
- $f, g$  represented as feed-forward neural networks.

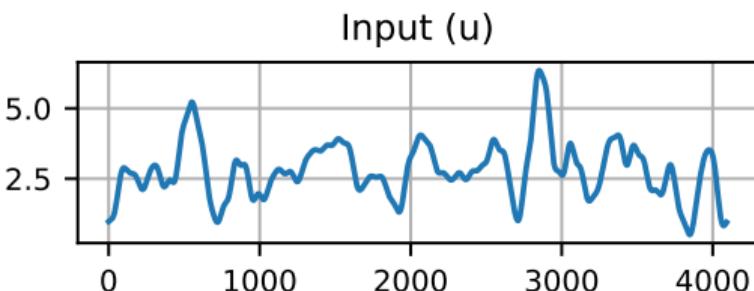
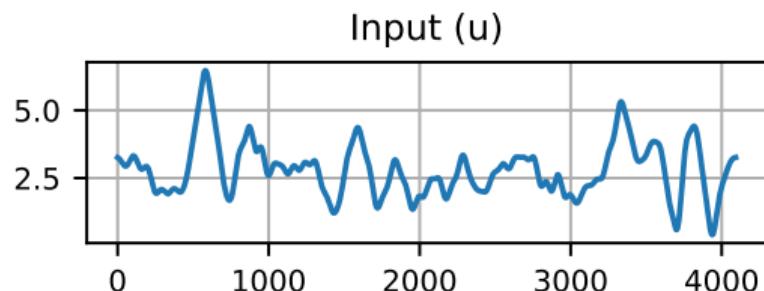
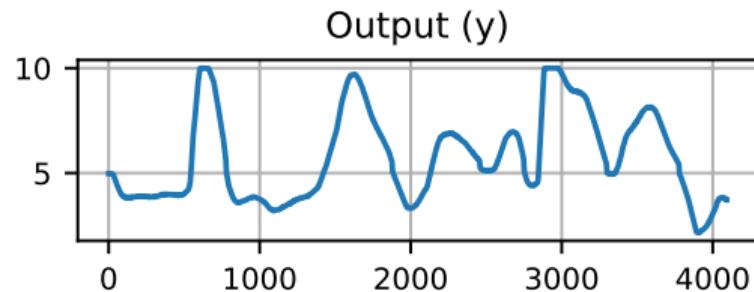
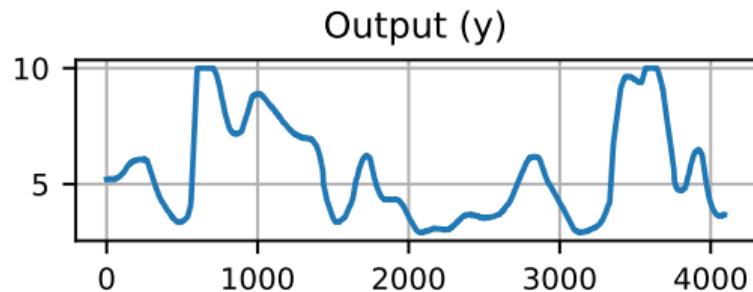
## Benchmark Dataset: Cascaded Tanks

Non-linear SISO system. Input: upper tank inlet flow  $u$ . Output: lower tank level  $x_2$ .

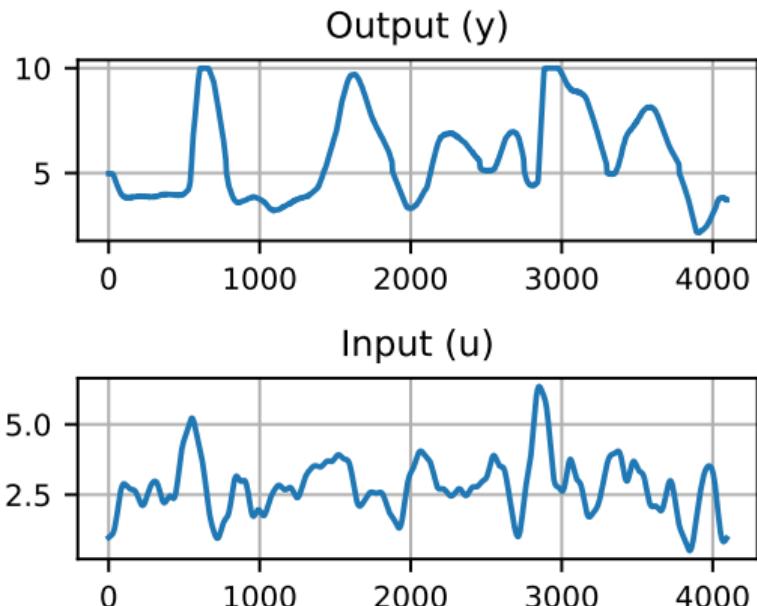
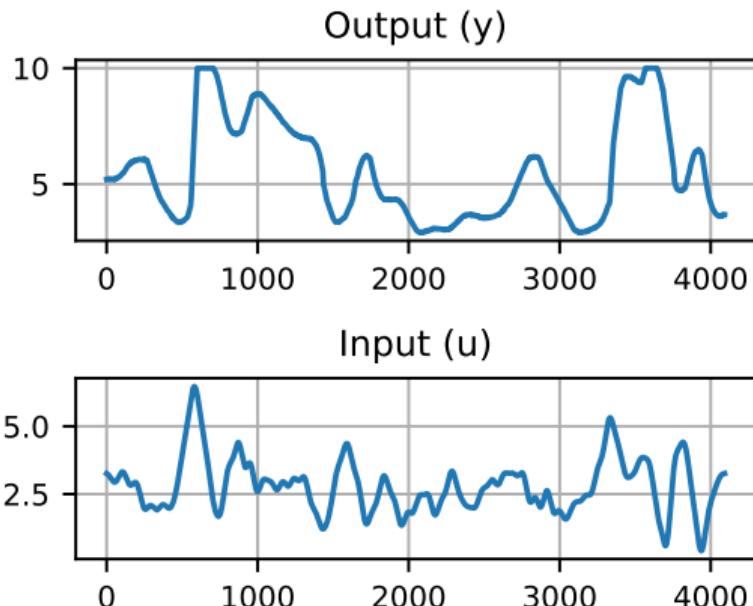


A pretty common **benchmark** for system identification algorithms.

## Training (left) and test (right) data



## Training (left) and test (right) data



```
u_tr.shape, y_tr.shape, u_te.shape, y_te.shape
```

```
((1024, 1), (1024, 1), (1024, 1), (1024, 1))
```

## Pre-processing

Learning algorithms perform better with favorable **scaling** (e.g., standardization)

```
u_mean = jnp.mean(u_tr); u_std = jnp.std(u_tr)
u_tr_sc = (u_tr - u_mean) / u_std
u_te_sc = (u_te - u_mean) / u_std # use training mean and std!
```

## Pre-processing

Learning algorithms perform better with favorable **scaling** (e.g., standardization)

```
u_mean = jnp.mean(u_tr); u_std = jnp.std(u_tr)
u_tr_sc = (u_tr - u_mean) / u_std
u_te_sc = (u_te - u_mean) / u_std # use training mean and std!
```

```
# Same shapes as before...
```

```
u_tr_sc.shape, y_tr_sc.shape
```

```
((1024, 1), (1024, 1))
```

## Pre-processing

Learning algorithms perform better with favorable **scaling** (e.g., standardization)

```
u_mean = jnp.mean(u_tr); u_std = jnp.std(u_tr)
u_tr_sc = (u_tr - u_mean) / u_std
u_te_sc = (u_te - u_mean) / u_std # use training mean and std!
```

```
# Same shapes as before...
```

```
u_tr_sc.shape, y_tr_sc.shape
```

```
((1024, 1), (1024, 1))
```

```
# ... but normalized to zero mean and unit variance
```

```
u_tr_sc.mean(), u_tr_sc.std()
```

```
(Array(-4.246831e-07, dtype=float32), Array(1., dtype=float32))
```

Same procedure applied to the output...

## Model fitting

- Among several valid options, we choose the model structure:

$$\begin{aligned}x(k+1) &= x(k) + W_2 \tanh(W_1 \text{vec}(x(k), u(k)) + b_1) + b_2 \\y(k) &= Cx(k)\end{aligned}$$

State update: previous state + FF( $x, u$ ). Output: linear.

## Model fitting

- Among several valid options, we choose the model structure:

$$\begin{aligned}x(k+1) &= x(k) + W_2 \tanh(W_1 \text{vec}(x(k), u(k)) + b_1) + b_2 \\y(k) &= Cx(k)\end{aligned}$$

State update: previous state + FF( $x, u$ ). Output: linear.

- We will implement simulation error minimization:

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

- $y^{\text{sim}}(k)$ : output *simulated* by iterating the model, starting from  $x(0)$
- Loss minimized w.r.t. the parameters  $p$  and the initial state  $x(0)$
- It enhances learning of long-term dependencies

## Define optimization variables

We will need to optimize both wrt parameters and initial condition. Let's define:

```
nu = 1; nx = 2; ny = 1; nh = 16
params = {
    "W1": jr.normal(keys[0], shape=(nh, nu+nx)), # nu + nx inputs
    "b1": jr.normal(keys[1], shape=(nh,)),
    "W2": jr.normal(keys[2], shape=(nx, nh)) * 1e-3, # nx outputs
    "b2": jr.normal(keys[3], shape=(nx,)) * 1e-3,
    "C": jr.normal(keys[4], shape=(ny, nx)), # nx inputs, ny outputs
}

x0 = jnp.zeros((nx,)) # initial state, also to be optimized
```

## Single step (left) and whole simulation (right)

We need `y_sim = sim(params, x0, u_tr_sc)`.

```
def f(p, x, u):
    xu = jnp.concatenate([x, u])
    z = jnp.tanh(p["W1"]@xu + p["b1"])
    x_new = x + p["W2"]@z + p["b2"]
    return x_new

def g(p, x):
    y = p["C"] @ x
    return y
```

```
def sim(p, x, us):
    # x: (nx), us:(T, nu)
    T = us.shape[0]; y_sim = []
    for t in range(T):
        y_sim.append(g(p, x))
        x = f(p, x, us[t])

    y_sim = jnp.stack(y_sim)
    return y_sim # (T, ny)
```

## Single step (left) and whole simulation (right)

We need `y_sim = sim(params, x0, u_tr_sc)`.

```
def f(p, x, u):
    xu = jnp.concatenate([x, u])
    z = jnp.tanh(p["W1"]@xu + p["b1"])
    x_new = x + p["W2"]@z + p["b2"]
    return x_new

def g(p, x):
    y = p["C"] @ x
    return y
```

```
def sim(p, x, us):
    # x: (nx), us:(T, nu)
    T = us.shape[0]; y_sim = []
    for t in range(T):
        y_sim.append(g(p, x))
        x = f(p, x, us[t])

    y_sim = jnp.stack(y_sim)
    return y_sim # (T, ny)
```

- For nerds: see `sim` implementation with `jax.lax.scan` (equivalent, faster)

## Loss function

We are almost done! Just need to define a loss and optimize w.r.t. `params` and `x0`:

## Loss function

We are almost done! Just need to define a loss and optimize w.r.t. `params` and `x0`:

- It is convenient to have all optimization variables in a same container

```
opt_vars = {"params": params, "x0": x0}
```

## Loss function

We are almost done! Just need to define a loss and optimize w.r.t. `params` and `x0`:

- It is convenient to have all optimization variables in a same container

```
opt_vars = {"params": params, "x0": x0}
```

- Then, write `loss_fn` with first argument `ov` with same structure as `opt_vars`

```
def loss_fn(ov, ys, us):  
    y_sim = sim(ov["params"], ov["x0"], us)  
    return jnp.mean((ys - y_sim)**2)
```

## Loss function

We are almost done! Just need to define a loss and optimize w.r.t. `params` and `x0`:

- It is convenient to have all optimization variables in a same container

```
opt_vars = {"params": params, "x0": x0}
```

- Then, write `loss_fn` with first argument `ov` with same structure as `opt_vars`

```
def loss_fn(ov, ys, us):  
    y_sim = sim(ov["params"], ov["x0"], us)  
    return jnp.mean((ys - y_sim)**2)
```

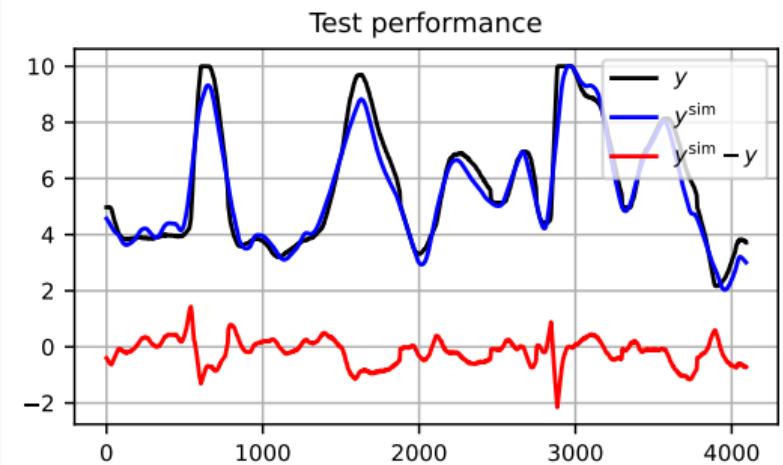
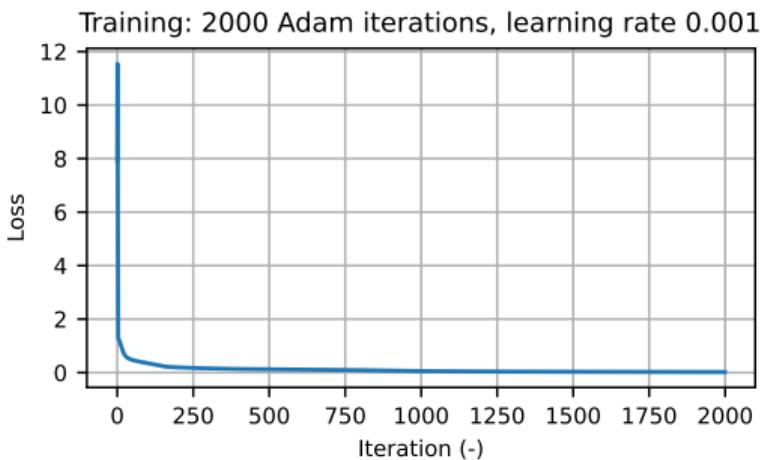
Loss function available, together with its derivatives. In a sense, the problem is *solved*.

```
loss_fn(opt_vars, y_tr_sc, u_tr_sc)
```

```
Array(7.950006, dtype=float32)
```

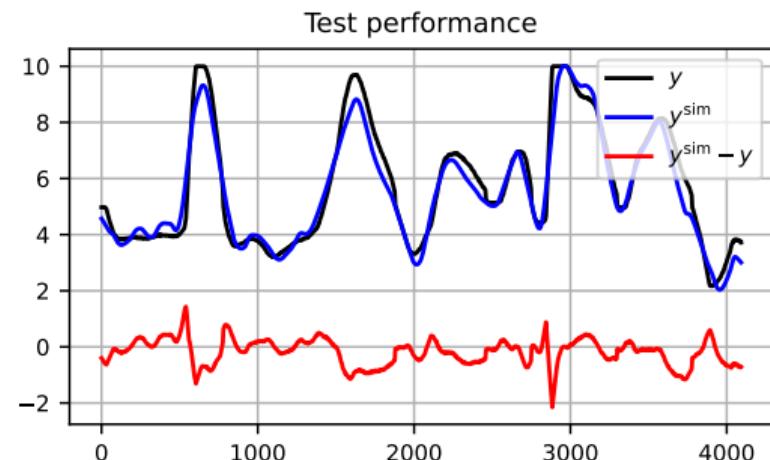
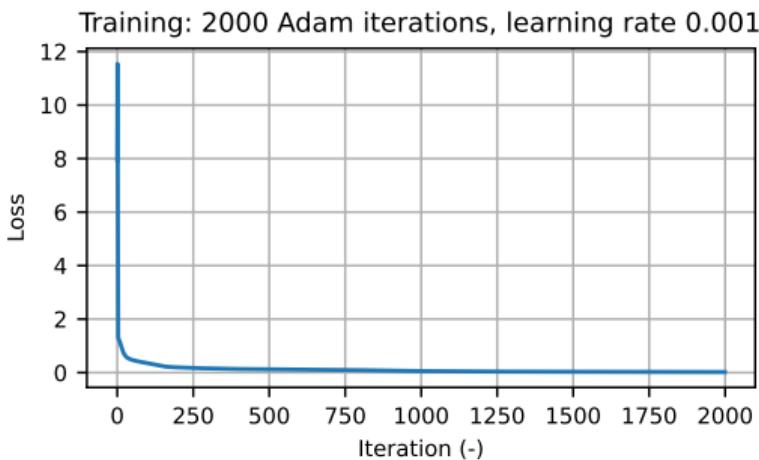
# Training loss and test performance

After training (left), we assess test-set performance (right).



## Training loss and test performance

After training (left), we assess test-set performance (right).



```
rmse = jnp.sqrt(jnp.mean((y_te_sim - y_te)**2)); rmse
```

Array(0.5024492, dtype=float32)

Not too bad! Check out the official **Benchmark Results**

## Exercise

- Reproduce the training at the previous slide
  - Hint: adapt the optimization code from the feed-forward example.
- Try out different models. You may modify:
  - Hyper-parameters and optimization settings
  - Parameter initialization
  - Number of states
  - Output equation as one of the two states
  - Output equation as another feed-forward neural net
  - ...
- Try out a different dataset