

Better Learning for ODEs



Decomposing Sequences

We can address the first two issues using a **reformulation**

Let's consider the sequence of measurements $\{y_k\}_{k=0}^n$

- We can view it as a sequence of pairs $\{(y_{k-1}, y_k)\}_{k=1}^n$
- ...Each referring to a **distinct** ODE, i.e. $\dot{\hat{y}}_k = f(\hat{y}_k, t; \theta)$
- ...With all ODEs sharing the **same** parameter vector θ

With this approach, we can reformulate the training problem as:

$$\begin{aligned} & \operatorname{argmin}_{\omega} \sum_{k=1}^n L(\hat{y}_k(t_k), y_k) \\ & \text{subject to: } \dot{\hat{y}}_k = f(\hat{y}_k, t; \theta) \quad \forall k = 1..n \\ & \quad \quad \hat{y}_k(t_{k-1}) = y_{k-1} \quad \forall k = 1..n \end{aligned}$$



In practice, we assume we are dealing with **multiple** initial value problems

Decomposing Sequences

Let's examine again the new training problem:

$$\operatorname{argmin}_{\omega} \sum_{k=1}^n L(\hat{y}_k(t_k), y_k)$$

$$\text{subject to: } \dot{\hat{y}}_k = f(\hat{y}_k, t; \theta) \quad \forall k = 1..n$$

$$\hat{y}_k(t_{k-1}) = y_{k-1} \quad \forall k = 1..n$$

There a few things to keep in mind:

- The approach is viable only if we have measurements for the **full state**
- ...And we are also assuming that the original loss is **separable**
- Finally, the new training problem is **not exactly equivalent** to the old one
- ...Since by re-starting at each step we are disregarding compound errors



Preparing the Data

Our implementation can naturally deal with the reformulation

We just need to properly prepare the data

- Each ODE can be seen as a different **example**

```
In [2]: ns = len(data.index)-1
```

- The sequence for each example contains **only two** measurements
- ...Corresponding to consecutive evaluation points

```
In [3]: tr_T = np.vstack((data.index[:-1], data.index[1:])).T  
print(tr_T[:3])
```

```
[[0. 1.]  
 [1. 2.]  
 [2. 3.]]
```



Preparing the Data

Our implementation can naturally deal with the reformulation

We just need to properly prepare the data

- The first measurement represents the initial state

```
In [4]: tr_y0 = np.array(data.iloc[:-1]).reshape(-1, 1)
        print(tr_y0[:2])
```

```
[[0.]
 [1.41003718]]
```

- The second to the final state, which we need for defining a target tensor

```
In [5]: tr_y = np.full((ns, 2, 1), np.nan)
        tr_y[:, 1, :] = data.iloc[1:]
        print(tr_y[:2])
```

```
[[[ nan]
   [1.41003718]]
```

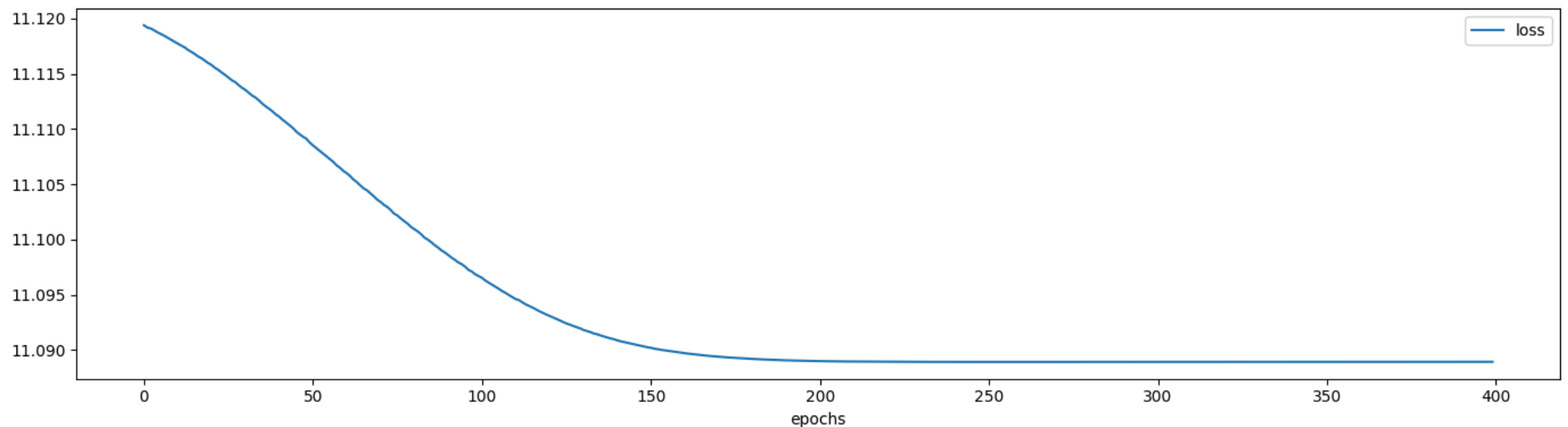
```
[[ nan]
 [2.6543906 ]]]
```



Training

Then we can perform training as usual

```
In [6]: %%time
dRC = util.RCNablaLayer(tau_ref=10, vs_ref=10)
euler = util.ODEEulerModel(dRC)
history = util.train_ml_model(euler, [tr_y0, tr_T], tr_y, validation_split=0.0, epochs=400)
util.plot_training_history(history, figsize=figsize)
```



Model loss: 11.0889 (training)

CPU times: user 2.51 s, sys: 622 ms, total: 3.13 s

Wall time: 2.39 s



Training

The results are the same as before (including estimation problems)

```
In [7]: print(f'tau: {tau:.2f} (real), {dRC.get_tau().numpy()[0]:.2f} (estimated)')  
        print(f'Vs: {Vs:.2f} (real), {dRC.get_vs().numpy()[0]:.2f} (estimated)')
```

```
tau: 8.00 (real), 8.51 (estimated)  
Vs: 12.00 (real), 12.00 (estimated)
```

...But there are significant computational advantages

Since we are using a **shallow** compute graph rather than a deep one...

- The training time is much lower
- Potential vanishing/exploding gradient problems are absent

Since we now have **multiple examples**...

- We can benefit from stochastic gradient descent
- We could use a validation set



Accuracy Issues

We are now ready to tackle our estimation issues

- We know we have trouble estimating the τ parameter
- Intuitively, that should translate in trouble estimating the dynamic behavior

Let's check whether this is true

- We prepare data structures to replicate our original run

```
In [8]: run_y0 = data.iloc[0].values.reshape(1, -1)
run_T = np.array([data.index])
print('y0:', run_y0)
print('T:', run_T)
```

```
y0: [[0.]]
T: [[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11. 12. 13. 14. 15. 16. 17.
    18. 19. 20. 21. 22. 23. 24. 25. 26. 27. 28. 29. 30. 31. 32. 33. 34. 35.
    36. 37. 38. 39. 40. 41. 42. 43. 44. 45. 46. 47. 48. 49. 50. 51. 52. 53.
    54. 55. 56. 57. 58. 59. 60.]]
```



Accuracy Issues

Then we can run Euler method directly using our model

As a side benefit, this will naturally use the estimate parameters

```
In [9]: run_y = euler.predict([run_y0, run_T], verbose=0)
```

Next, let's build a dataset with the original data and the predictions:

```
In [10]: data_euler = data.copy()  
data_euler['euler'] = run_y[0]  
data_euler.head()
```

Out[10]:

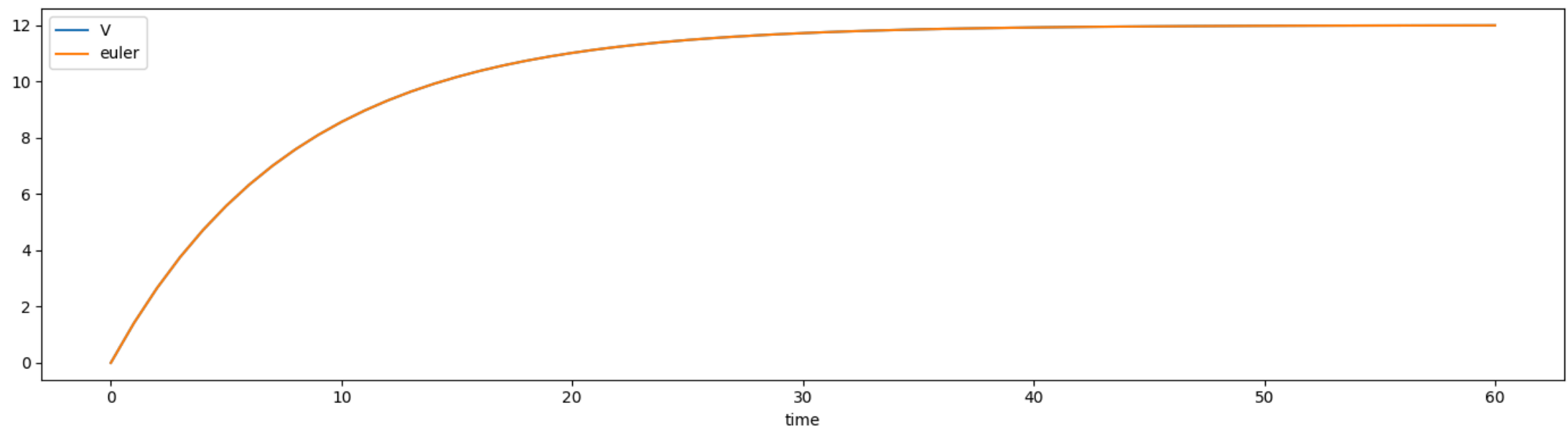
	V	euler
time		
0.0	0.000000	0.000000
1.0	1.410037	1.410051
2.0	2.654391	2.654415
3.0	3.752529	3.752560
4.0	4.721632	4.721668



Accuracy Issues

Finally, we can plot the two curves

```
In [11]: util.plot_df_cols(data_euler, figsize=figsize)
```



We have a very good match!

What is going on?



Accuracy Issues?

We formulated the training problem in terms of curve fitting

- I.e. we optimized τ and V_s so as to obtain a close fitting curve
- ...Constructed using Euler method

The problem is that Euler method is **inaccurate**

- If using wrong parameters will lead to a better fitting curve
- ...Our approach will not hesitate to do just that

Is this a problem?

If we just care about the curve, not at all

- It can actually be an advantage, if properly exploited

If we care about estimating parameters, then yes

- ...But it also suggests an easy fix (using a more accurate integration method)



Improving Parameter Estimation

For sake of simplicity, we will keep using Euler method

...And we will just increase the number of steps to improve its accuracy

- First, we introduce **more evaluation points** for each measurement pair

```
In [12]: nsteps = 11
tr_T2 = np.vstack(np.linspace(data.index[:-1], data.index[1:], nsteps)).T
print(tr_T2[:2])
```

```
[[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
 [1.  1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2. ]]
```

- Second, we update the target sequences to match the size

```
In [13]: tr_y2 = np.full((ns, nsteps, 1), np.nan)
tr_y2[:, -1, :] = data.iloc[1:]
print(str(tr_y2[:2]).replace('\n', ', ').replace(' ', '').replace(',,', '\n'))
```

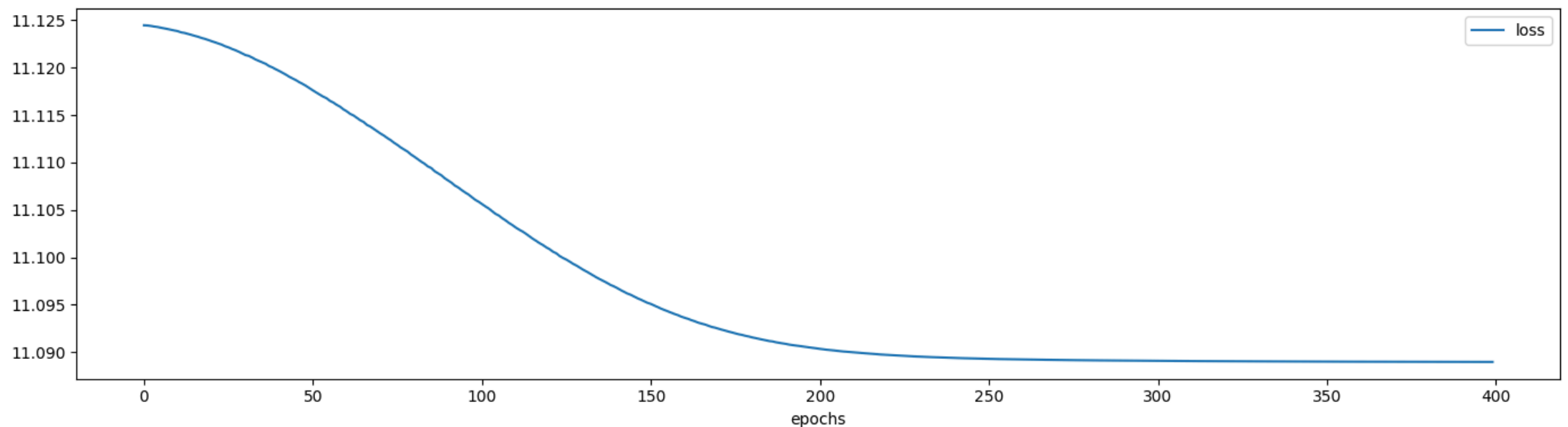
```
[[[nan], [nan], [nan], [nan], [nan], [nan], [nan], [nan], [nan], [nan], [1.41003718]]
 [[nan], [nan], [nan], [nan], [nan], [nan], [nan], [nan], [nan], [nan], [2.6543906]]]
```



Improving Parameter Estimation

Then, we can train as usual

```
In [14]: %%time
dRC2 = util.RCNablaLayer(tau_ref=10, vs_ref=10)
euler2 = util.ODEEulerModel(dRC2)
history = util.train_ml_model(euler2, [tr_y0, tr_T2], tr_y2, validation_split=0.0, epochs=400)
util.plot_training_history(history, figsize=figsize)
```



Model loss: 11.0890 (training)

CPU times: user 2.9 s, sys: 766 ms, total: 3.67 s

Wall time: 2.73 s



Improving Parameter Estimation

This approach leads to considerably better estimates

```
In [15]: print(f'tau: {tau:.2f} (real), {dRC2.get_tau().numpy()[0]:.2f} (estimated)')  
         print(f'Vs: {Vs:.2f} (real), {dRC2.get_vs().numpy()[0]:.2f} (estimated)')
```

```
tau: 8.00 (real), 8.05 (estimated)  
Vs: 12.00 (real), 12.00 (estimated)
```

- The results can be improved by using additional steps
- ...Or by switching to a different integration method (e.g. RK4)

Overall, when using this approach...

...It's important to be aware that integration methods are **approximate**

- This can easily lead to incorrectly estimated parameters
- Which may or may not be a problem, depending on your priorities

