Universal ODEs

One More Step

Is it really worth it?

So far, we have just used gradient descent to train ODEs

- We could have achieved the same results with other methods
- What is the added value of using a "neural" engine?

Is is worth it? Why?

One More Step

Is it really worth it?

So far, we have just used gradient descent to train ODEs

- We could have achieved the same results with other methods
- What is the added value of using a "neural" engine?

There are several advantages

High-dimensionality is not a problem

- We can train ODEs with multiple parameters
- E.g. V_s or τ that vary over time

We can approximate ODEs with weaker methods

- We managed to use Euler method to obtain good curves
- ...And weaker methods are computationally cheaper

Universal Ordinary Differential Equations

The real deal is the ability to incorporate black-box functions

This is sometimes called a Universal Ordinary Differential Equation (UDE)

$$\dot{y} = f(y, t, U(y, t))$$

- y, t, and f are as usual
- ...Except that some of its parameters come from a second function U
- U is a trainable universal approximator (typically a NN)

This is an example of Physics Informed Neural Networks

- f encodes (interpretable) knowledge about the system behavior
- U can be trained to learn implicit knowledge from data

The result is a very flexible hybrid framework

An Example

As ax example, let's consider the SIR model we have encountered

This is a simple, but locally effective, epidemic model

$$
\begin{aligned}
\dot{S} &= -\beta \frac{1}{N} S I \\
\dot{I} &= +\beta \frac{1}{N} S I - \gamma I \\
\dot{R} &= +\gamma I
\end{aligned}
$$

Say we want to control the epidemic via Non-Pharmaceutical Interventions

- E.g. using masks, social distancing, etc.
- These typically have an effect on β, and they change over time

They can (partially) explain multiple waves observed in a real epidemic

SIR with NPIs

We can model the effect of NPIs via a UDE model

$$\begin{aligned}\dot{S} &= -\beta(t)\frac{1}{N}SI \\ \dot{I} &= +\beta(t)\frac{1}{N}SI - \gamma I \\ \dot{R} &= +\gamma I\end{aligned}$$

- Where $U(y, t)$ corresponds to $\beta(t)$

In practice, depending on t certain NPIs will be active and affect β

- The connection is complex and cannot be modeled by an expert
- ...But given enough data, it could be learned

Use Case Parameters

Let's see the approach in action on a synthetic use case

...Where we will have access to ground truth information

- We will assume that initially 1% of the population is infected
- ...That the recovery time is 10 days ($\gamma = 1/10$)
- ...And that the "natural" β value is 0.23

In [3]:
```
S0, I0, R0 = 0.99, 0.01, 0.00
beta_base, gamma = 0.23, 1/10
```

We assume that NPIs cut that number by a measure-specific factor

Assuming I is the set of active NPIs, the ground truth function $\hat{\beta}(t)$ is:

$$\hat{\beta}(t) = \beta \prod_{i \in I} e_i$$

Non-Pharmaceutical Interventions

We will consider the following NPIs

```
In [5]: npis = [
            util.NPI('masks-indoor', effect=0.75, cost=1),
            util.NPI('masks-outdoor', effect=0.9, cost=1),
            util.NPI('dad', effect=0.7, cost=3),
            util.NPI('bar-rest', effect=0.6, cost=3),
            util.NPI('transport', effect=0.6, cost=4)
        ]
```

For sake of simplicity, we will sample NPI values at random

- We will change them at random every week
- ...With a restriction on the number of NPIs that can be simultanously active
- We will update the $\beta(t)$ value accordingly
- ...And simulate the epidemics by integrating a SIR model

The Dataset

Let's use this approach of build a 52-week dataset

In [11]:
```
nweeks = 52
data = util.gen_SIR_NPI_dataset(S0, I0, R0, beta_base, gamma, npis, nweeks, steps_per_day=5, max
data.iloc[:8]
```

Out[11]:

	S	I	R	week	masks-indoor	masks-outdoor	dad	bar-rest	transport	beta
0.0	0.990000	0.010000	0.000000	0	0	0	1	1	0	0.0966
1.0	0.989046	0.009956	0.000998	0	0	0	1	1	0	0.0966
2.0	0.988098	0.009911	0.001991	0	0	0	1	1	0	0.0966
3.0	0.987154	0.009866	0.002980	0	0	0	1	1	0	0.0966
4.0	0.986216	0.009820	0.003964	0	0	0	1	1	0	0.0966
5.0	0.985283	0.009773	0.004944	0	0	0	1	1	0	0.0966
6.0	0.984356	0.009725	0.005919	0	0	0	1	1	0	0.0966
7.0	0.983434	0.009677	0.006889	1	0	0	0	1	1	0.0828

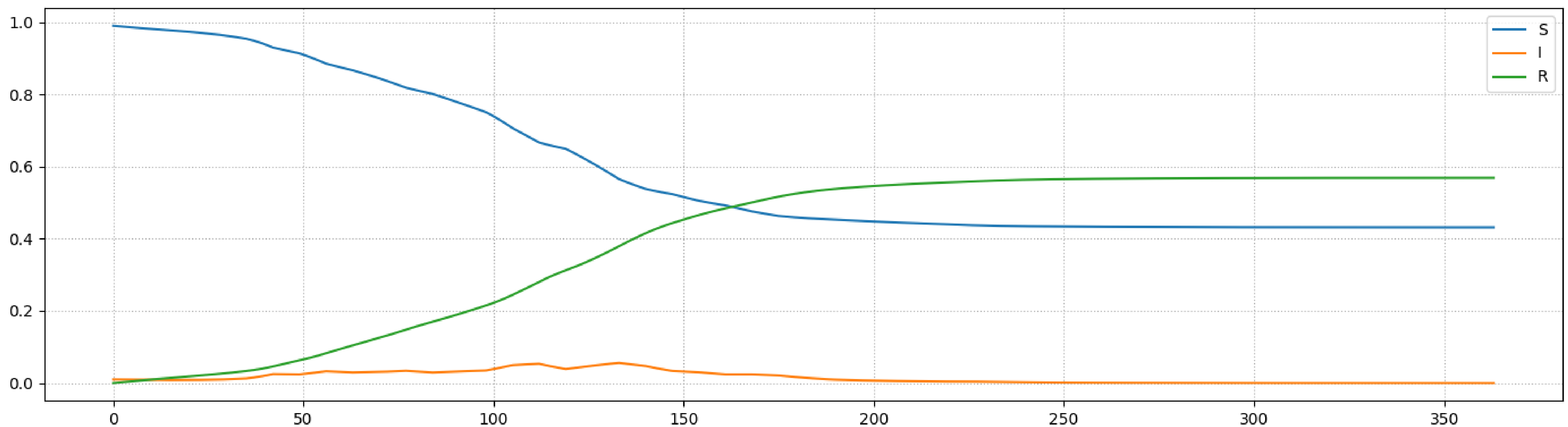
- Despite the results are obtained using an accurate method
- ...We still assume access to a single measurement per day

This is typically the case in real-world epidemics

The Dataset

Let's plot the S, I, R component from the dataset

In [12]:
```
util.plot_df_cols(data[['S', 'I', 'R']], figsize=figsize)
```

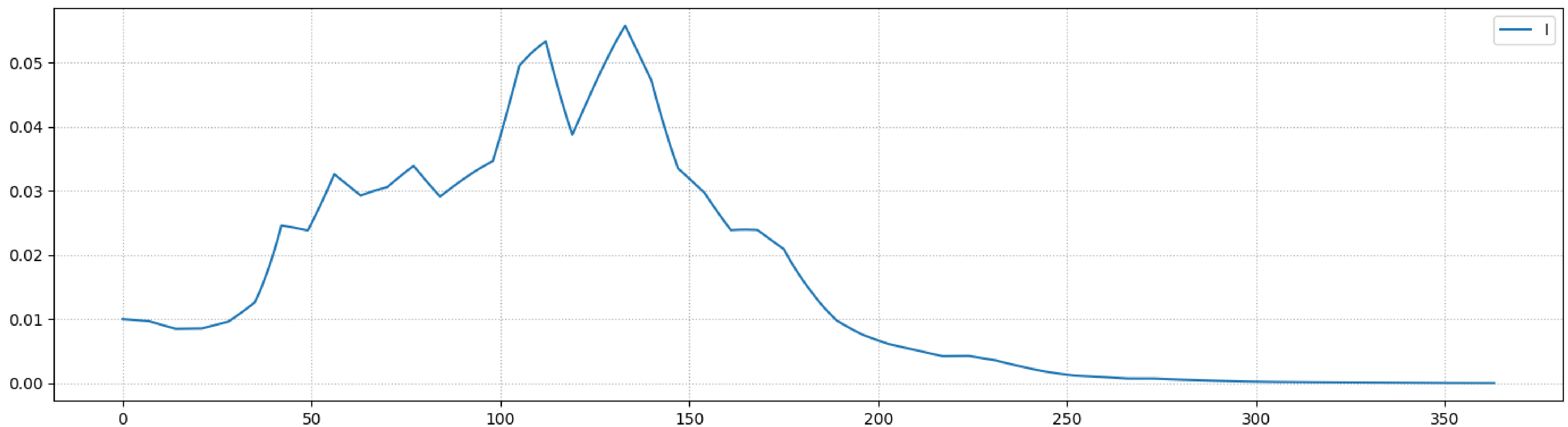


- There is still a single wave
- ...Due to how we sampled the NPIs (and their effects)

The Dataset

Locally, the behavior is more complex

```
In [13]: util.plot_df_cols(data[['I']], figsize=figsize)
```



- When $\frac{\hat{\beta}(t)}{\gamma} > 1$ we have a true epidemic behavior
- When $\frac{\hat{\beta}(t)}{\gamma} \leq 1$, the number of new cases always drops

The Implementation

In principle, our previous code should be enough

...I.e. we could use a custom layer for the UDE:

$$\begin{aligned}\dot{S} &= -\beta(t)\frac{1}{N}SI \\ \dot{I} &= +\beta(t)\frac{1}{N}SI - \gamma I \\ \dot{R} &= +\gamma I\end{aligned}$$

In practice, things are slightly more complicated

- Formally, the input for $\beta(t)$ is time
- ...But the input we care about are the active NPIs

The Implementation

Therefore, a more accurate formulation would be

$$\begin{aligned}
\dot{S} &= -\beta(NPI(t))\frac{1}{N}SI \\
\dot{I} &= +\beta(NPI(t))\frac{1}{N}SI - \gamma I \\
\dot{R} &= +\gamma I
\end{aligned}$$

In principle, our custom layer should:

- Take as input S, I, R and t
- Use t to retrieve $NPI(t)$
- ...And then compute the gradient

In practice, it's easier to supply the NPIs as additional inputs

Custom NPI-SIR Layer

We provide the SIR layer with a $\beta(NPI)$ model at construction time

```
class NPISIRNablaLayer(keras.layers.Layer):
    def __init__(self, beta_pred, gamma_ref=0.1, fixed_gamma=None):
        ...
        self.beta_pred = beta_pred # NPI-to-beta model
        ...

    def call(self, inputs):
        y, t, npis = inputs # unpack
        ...
        beta = self.beta_pred(npis) # obtain beta
        ...
```

- This is the `beta_pred` parameter in the `NPISIRNablaLayer` class
- In the `call` method we unpack the auxiliary input

Custom NPI-SIR Layer

We provide the SIR layer with a $\beta(NPI)$ model at construction time

```
class NPISIRNablaLayer(keras.layers.Layer):
    def __init__(self, beta_pred, gamma_ref=0.1, fixed_gamma=None):
        ...
        self.beta_pred = beta_pred # NPI-to-beta model
        ...

    def call(self, inputs):
        y, t, npis = inputs # unpack
        ...
        beta = self.beta_pred(npis) # obtain beta
        ...
```

- `npis` is a vector representing active NPI, using a 0/1 encoding
- ...And we use the `beta_pred` model to obtain β

Modified Euler Method Model

Then, we modify our custom model

We introduce a flag to tell the model we plan to use auxiliary inputs

```
class ODEEulerModel(keras.Model):
    def __init__(self, f, auxiliary_input=False, **params):
        ...

    def call(self, inputs, training=False):
        if self.auxiliary_input:
            y, T, aux = inputs
        else:
            y, T = inputs
        ...
```

- We unpack all inputs in the `call`, `train_step`, and `test_step` method
- We have the initial state `y`, the evaluation points `T`, and `aux`

Modified Euler Method Model

Then, we modify our custom model

We introduce a flag to tell the model we plan to use auxiliary inputs

```
class ODEEulerModel(keras.Model):
    def __init__(self, f, auxiliary_input=False, **params):
        ...

    def call(self, inputs, training=False):
        if self.auxiliary_input:
            y, T, aux = inputs
        else:
            y, T = inputs
        ...
```

- We need an NPI vector for each evaluation point (except the last)
- Hence `aux` should have `len(T)-1` elements

Preparing the Training Data

The data structures for the initial state are as usual

```
In [14]: tr_y0 = data[['S', 'I', 'R']].values[:-1];
         print(tr_y0[:2])

[[0.99       0.01       0.        ]
 [0.98904622 0.00995598 0.0009978 ]]
```

The same goes for the evaluation points

```
In [15]: euler_steps = 5
         tr_T = np.linspace(data.index[:-1], data.index[1:], euler_steps).T
         print(tr_T[:2])

[[0.   0.25 0.5  0.75 1.  ]
 [1.   1.25 1.5  1.75 2.  ]]
```

- We choose to use 5 euler steps per time unit, for a better approximation
- Since our goal is estimating β, accuracy is important

Preparing the Training Data

NPI vectors stay constant for every time unit

```
In [16]: npi_names = [n.name for n in npis]
         tmp = data[npi_names].values[:-1]
         ns = len(tr_y0)
         tr_npi = np.tile(tmp, euler_steps-1).reshape(ns, -1, len(npi_names))
         print(tr_npi[:2])

[[[0 0 1 1 0]
  [0 0 1 1 0]
  [0 0 1 1 0]
  [0 0 1 1 0]]

 [[0 0 1 1 0]
  [0 0 1 1 0]
  [0 0 1 1 0]
  [0 0 1 1 0]]]
```

- We obtain the NPI values for each time unit
- ...And we repeat them for every intermediate Euler step
- Since NPIs are input, they are not needed for the last step

Preparing the Training Data

The target data is as usual

```
In [17]: ns = len(tr_y0)
         tr_y = np.full((ns, euler_steps, 3), np.nan)
         tr_y[:, -1, :] = data[['S', 'I', 'R']].values[1:]
         print(tr_y[:2])
```

```
[[[        nan         nan         nan]
  [        nan         nan         nan]
  [        nan         nan         nan]
  [        nan         nan         nan]
  [0.98904622 0.00995598 0.0009978 ]]

 [[        nan         nan         nan]
  [        nan         nan         nan]
  [        nan         nan         nan]
  [        nan         nan         nan]
  [0.98809759 0.00991123 0.00199117]]]
```

- Most entries are null, since we have only one measurement per time unit

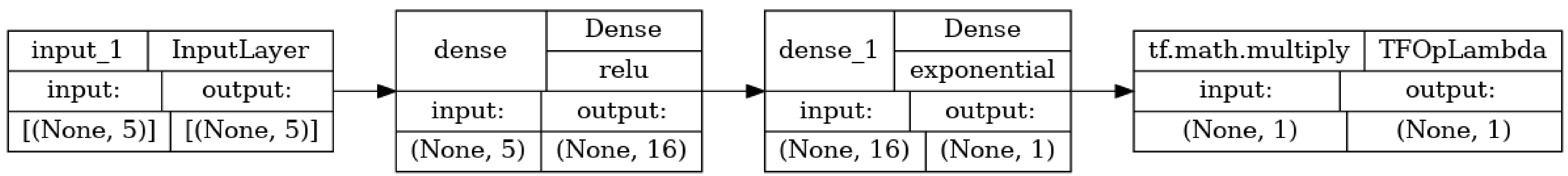
Building the Model

We start by building the $\beta(NPI)$ model

In [18]:
```
beta_pred = util.build_nn_model(input_shape=len(npi_names), output_shape=1,
                                hidden=[16], output_activation='exponential', scale=0.1)
util.plot_nn_model(beta_pred)
```

```
2022-11-28 14:59:23.894431: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFl
ow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following C
PU instructions in performance-critical operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
```

Out[18]:



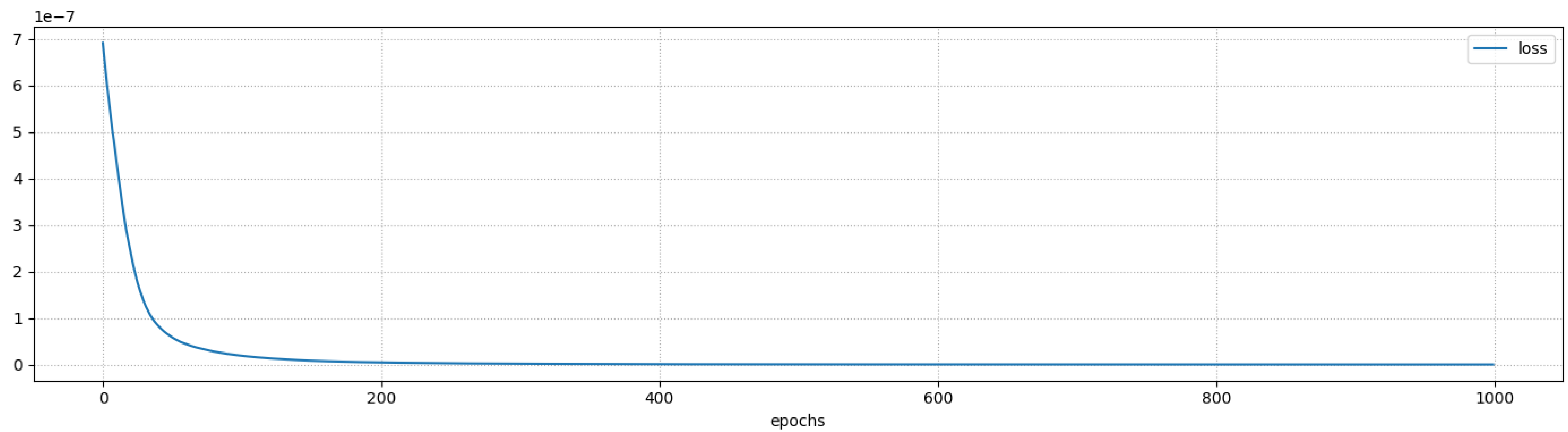
- We use an exponential activation to ensure non-negative β values
- ...And a scaling factor to make the initial guess more reasonable

Then we build an instance of the modified SIR layer and feed it to the Euler model:

Training

Now we can perform training as usual

```
In [20]: %%time
history = util.train_nn_model(euler, [tr_y0, tr_T, tr_npi], tr_y, loss='mse', validation_split=(
util.plot_training_history(history, figsize=figsize)
```




```
Final loss: 0.0000 (training)
CPU times: user 17.3 s, sys: 2.21 s, total: 19.5 s
Wall time: 11.2 s
```

Evaluation

Let's the the quality of estimate curves

We will use the `call` method to have the same conditions as training

- We prepare the initial state

```
In [21]: run_y0 = data[['S', 'I', 'R']].iloc[0].values
         run_y0 = np.array([run_y0])
         print(run_y0)

         [[0.99 0.01 0.  ]]
```

- Then all the evaluation points (in a whole year)

```
In [22]: run_T = np.arange(0, data.index[-1]+1/euler_steps, 1/euler_steps)
         run_T = np.array([run_T])
         print(run_T)

         [[0.000e+00 2.000e-01 4.000e-01 ... 3.626e+02 3.628e+02 3.630e+02]]
```

Evaluation

Let's the the quality of estimate curves

Finally, we prepare the NPI vectors

In [23]:
```
run_npis = np.tile(data[npi_names].values, euler_steps).reshape(-1, len(npi_names))
run_npis = np.array([run_npis])
print(run_npis)
```

```
[[[0 0 1 1 0]
  [0 0 1 1 0]
  [0 0 1 1 0]
  ...
  [1 0 0 1 0]
  [1 0 0 1 0]
  [1 0 0 1 0]]]
```

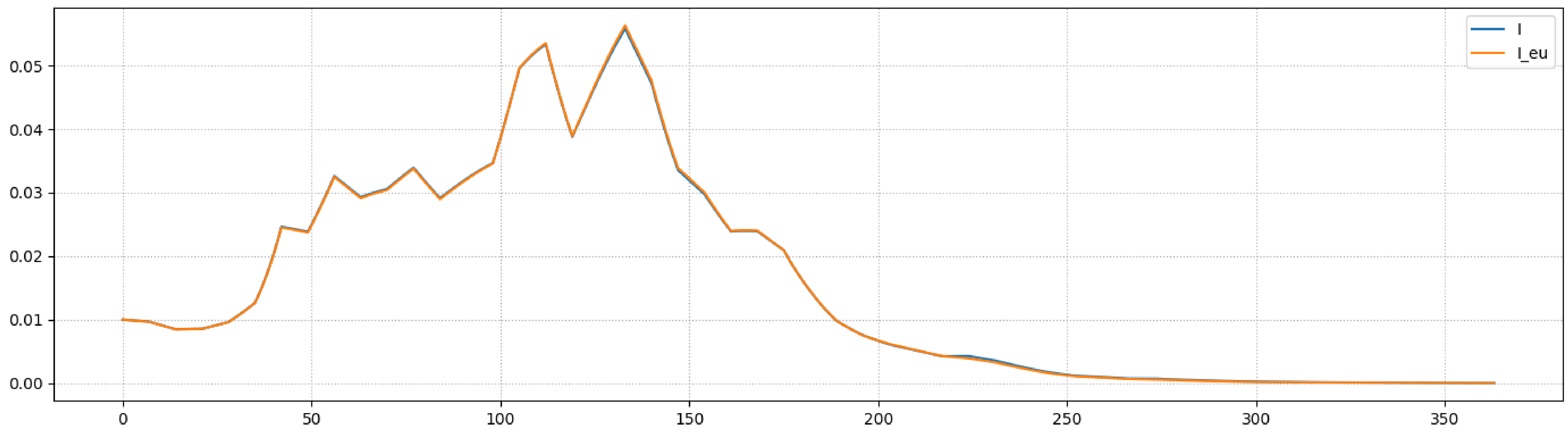
...And finally we integrate the ODE

In [24]:
```
run_y = euler([run_y0, run_T, run_npis])
```

Evaluation

Let's plot the original measurement and the estimated curve

In [25]:
```
run_cmp = pd.DataFrame(data=run_y[0].numpy(), index=run_T[0], columns=['S_eu', 'I_eu', 'R_eu'])
run_cmp[['S', 'I', 'R']] = data[['S', 'I', 'R']]
run_cmp[['S', 'I', 'R']] = run_cmp[['S', 'I', 'R']].interpolate()
util.plot_df_cols(run_cmp[['I', 'I_eu']], figsize=figsize)
```

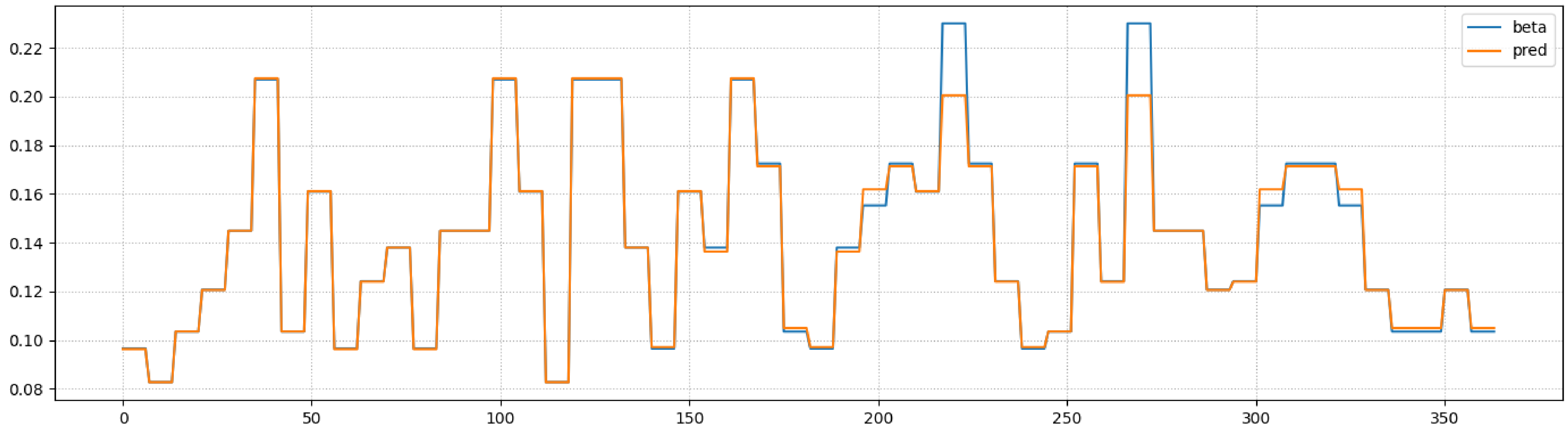


- There is a pretty good match

Beta Estimation

Finally, let's make a qualitative check of our β estimates

In [26]:
```
beta_cmp = data[['beta']].copy()
beta_cmp['pred'] = beta_pred.predict(data[npi_names], verbose=0)
util.plot_df_cols(beta_cmp, figsize=figsize)
```



- The estimates are quite good, except for the later part of the sequence
- This is due mostly to our choosing the plain MSE as a loss function

Final Considerations

Keep in mind that this example has limitations

- In practice, a SIR model may not be the best match
- The NPIs actually tested may cover the input space poorly
- Some state component may not be measurable (e.g. S)

But the UDE approach is very flexible and can be quite effective

What is the connection with constraints?

- A simple approach to account for constraints in ML
- ...Is to enforce them at an architectural level

UDEs are one interesting case of this more general template