Training an ML Model

The Dataset

So far, we have introduced our simulator

The rest of our plan is as follows

- We learn an ML model
- We embed the model in a larger optimization problem
- We obtain a solution, i.e. a set of action to control the epidemics

But which data are we going to use for training?

The Dataset

So far, we have introduced our simulator

The rest of our plan is as follows

- We learn an ML model
- We embed the model in a larger optimization problem
- We obtain a solution, i.e. a set of action to control the epidemics

But which data are we going to use for training?

Since we have a simulator, we can build our dataset

- This means we can generate as much data as we wish
- ...But also that we are responsible for how to generate it

Building Our Dataset

We need to define the structure of the dataset

- We will focus on Non-Therapeutic Interventions (mask mandates, etc.)
- NPIs affect the β parameter in a SIR model (γ will be fixed)
- We will focus on making predictions at weekly intervals

Therefore, we can cover our needs with...

- For the input:
  - The state (S_0, I_0, R_0) at the beginning of a week
  - The value of β, i.e. a proxy for the NPIs currently in effect
- For output: the state after one week (S, I, R)

Given an input (S, I, R, β), we can get the output via simulation

Which input configurations should we generate?

Building Our Dataset

A training set should be representative of the test distribution

- We do not have a fixed test distribution (no test set)
- ...But we know that the ML model will be used by an optimizer

The optimizer will seek to minimize the total infections. So, we will need:

- High accuracy on the best configurations, so as to find them
- High accuracy on the worst configurations, so as to avoid them

I.e. to be safe the model should work all across the board

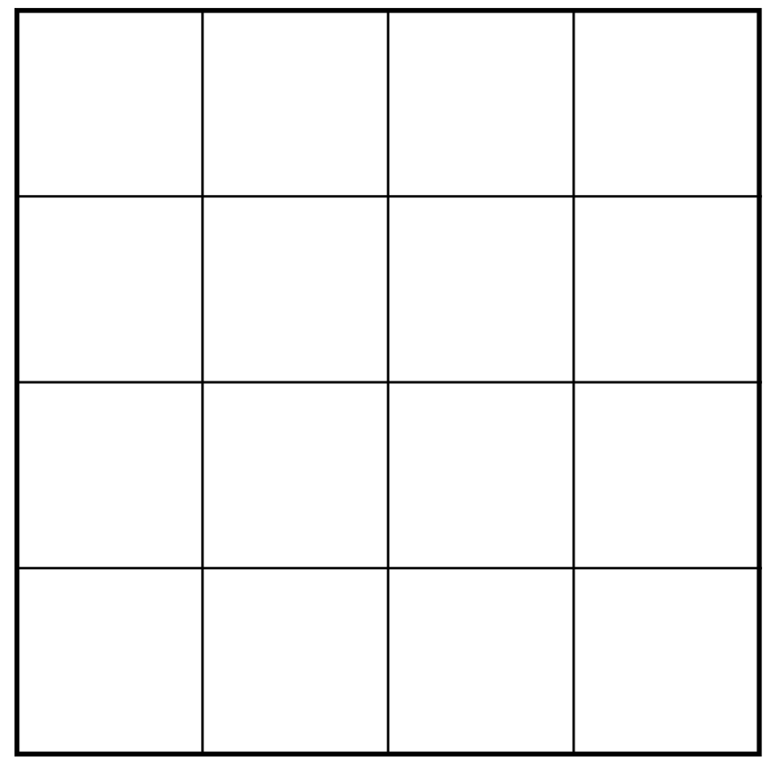
Hence, we need to cover reasonably well all the input space

- The simplest approach would be use use a regular grid (factorial design)
- ...But factorial design has poor scalability

Latin Hypercube Sampling

The method we will use is called Latin Hypercube Sampling

Suppose we want to sample m points for n attributes with fixed ranges



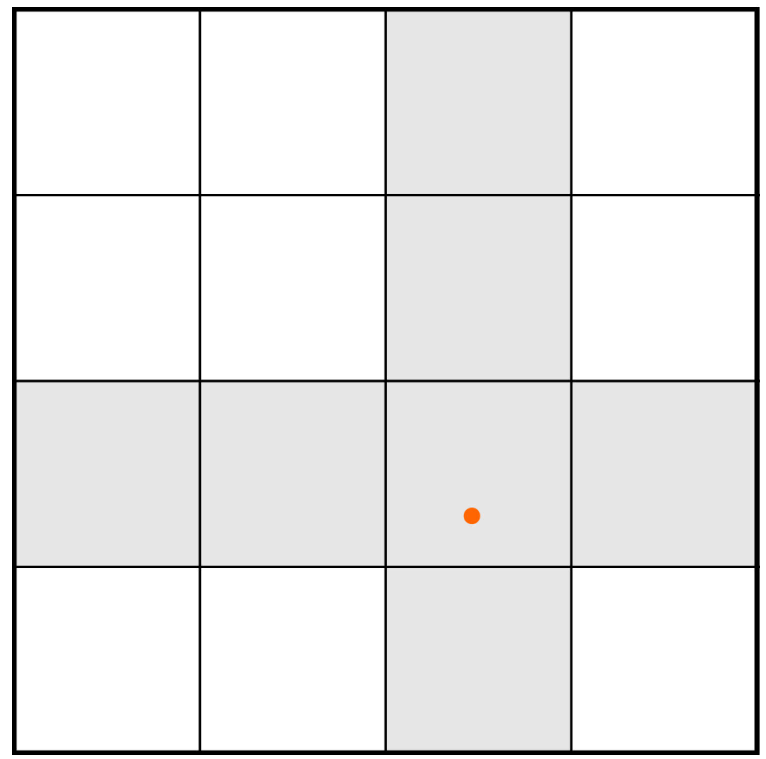
- We can view the sampling space as a hypercube
- ...Then we divide each dimension in n segments

In the example we will to sample 4 points for 2 attributes

Latin Hypercube Sampling

The method we will use is called **Latin Hypercube Sampling**

Suppose we want to sample m points for n attributes with fixed ranges

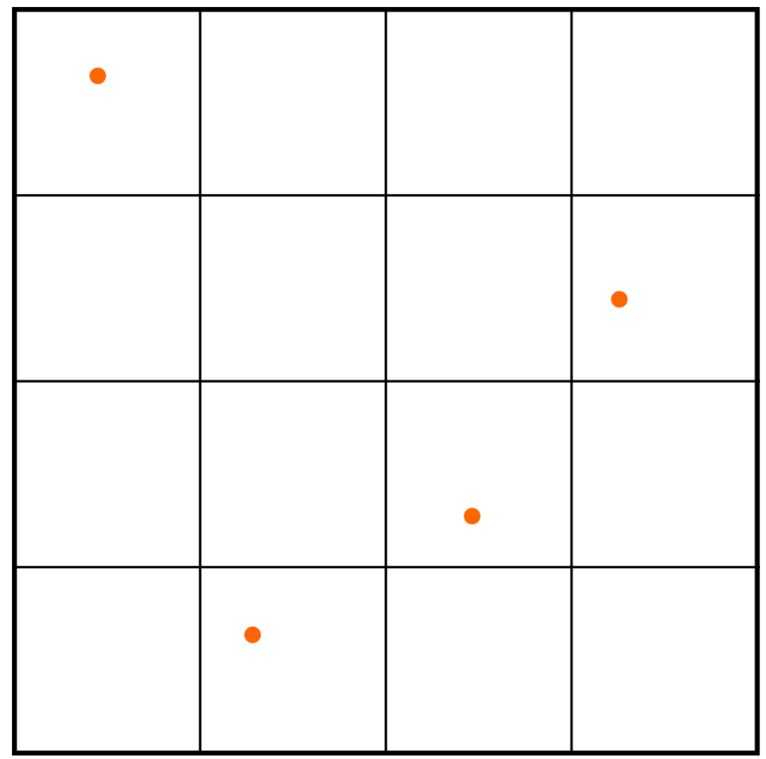


- We sample the first point uniformly at random
- ...Then we cover the row and column that contain the sample

Latin Hypercube Sampling

The method we will use is called Latin Hypercube Sampling

Suppose we want to sample m points for n attributes with fixed ranges



- When we take additional samples, we exclude all covered row/columns
- ...So we end up with a pattern similar to that of the figure

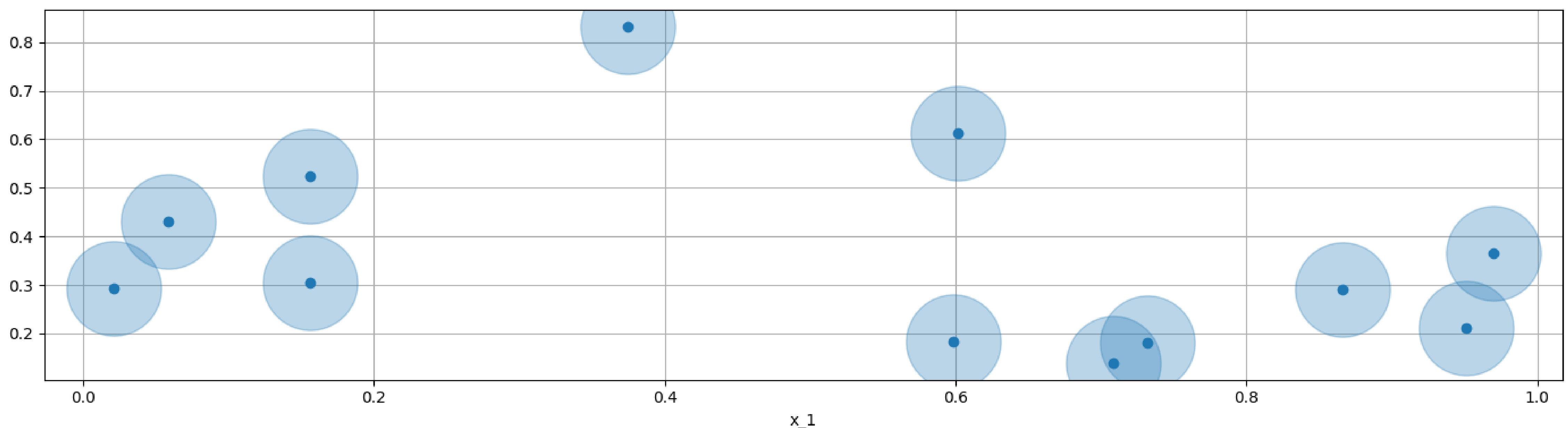
LHS can cover quite uniformly a given space with relatively few samples

Latin Hypercube Sampling

Let's see a practical example

Here is the result of uniform sampling, for reference

In [2]:
```
test_nsamples, test_ranges = 12, [(0., 1.), (0., 1.)]
X = util.sample_points(test_ranges, test_nsamples, mode='uniform', seed=42)
util.plot_2D_samplespace(X, figsize=figsize)
```

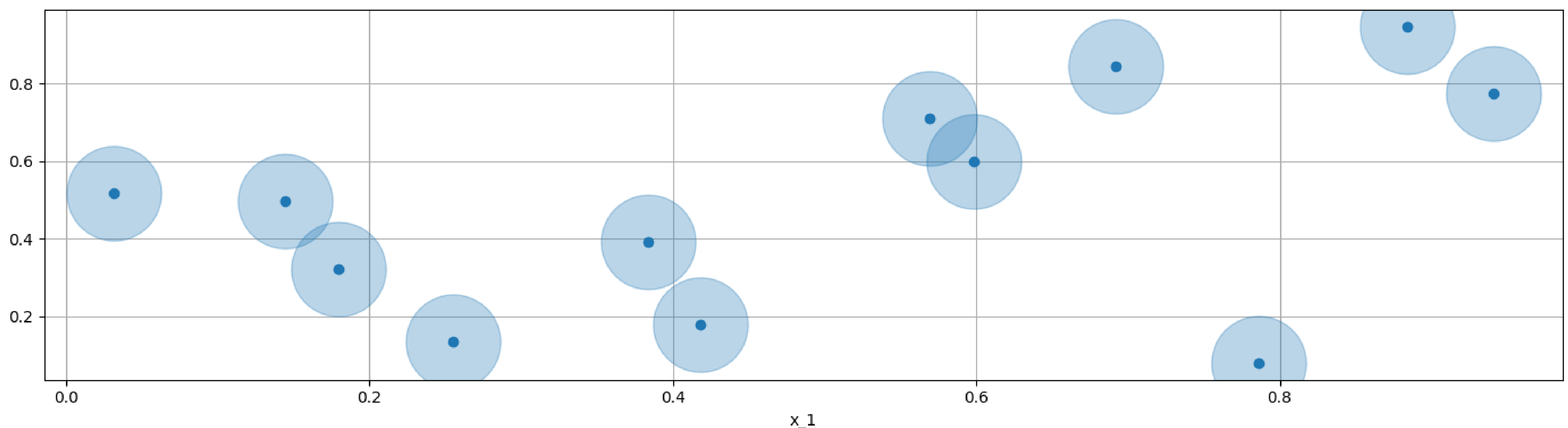


Latin Hypercube Sampling

Let's see a practical example

...And here is the result of classical LHS:

In [3]:
```
test_nsamples, test_ranges = 12, [(0., 1.), (0., 1.)]
X = util.sample_points(test_ranges, test_nsamples, mode='lhs', seed=42)
util.plot_2D_samplespace(X, figsize=figsize)
```

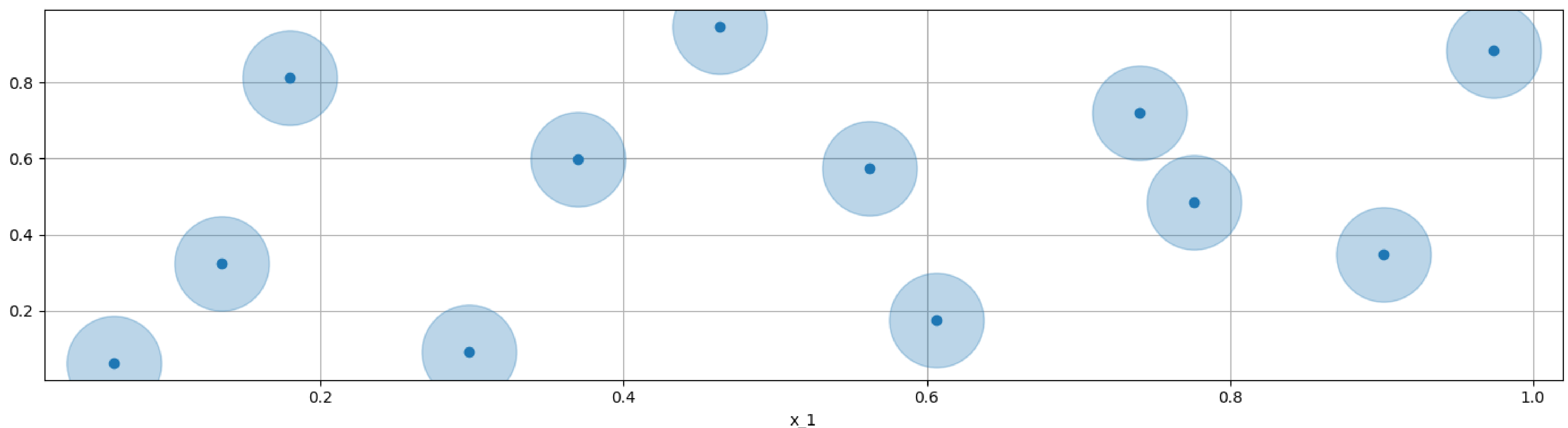


Latin Hypercube Sampling

The process can be further improved

E.g. we can sample multiple times to maximize the minimum distance

In [4]:
```
test_nsamples, test_ranges = 12, [(0., 1.), (0., 1.)]
X = util.sample_points(test_ranges, test_nsamples, mode='max_min', seed=42)
util.plot_2D_samplespace(X, figsize=figsize)
```



Dataset Input

We are now ready to generate our dataset input

```
In [5]: n_tr, n_ts = 10000, 2000
        sir_tr_in = util.generate_SIR_input(max_samples=n_tr, mode='max_min', seed=42, normalize=True, r
        sir_ts_in = util.generate_SIR_input(max_samples=n_ts, mode='max_min', seed=42, normalize=True, r
        sir_tr_in.head()
```

Out[5]:

	S	I	R	beta
0	0.352572	0.512310	0.135119	0.167925
1	0.175114	0.707867	0.117019	0.100628
2	0.024010	0.836985	0.139005	0.048884
3	0.139277	0.247573	0.613150	0.367367
4	0.491191	0.195846	0.312963	0.067067

- We sample S, I, R, β from $[0, 1]^3 \times [0, .4]$
- ...Then S, I, R are normalized so that their sum is 1

This will reduce in some redundancy in the dataset

Dataset Output

We obtain the corresponding output via simulation

In [6]:
```
%%time
gamma = 1/14
sir_tr_out = util.generate_SIR_output(sir_tr_in, gamma, 7)
sir_ts_out = util.generate_SIR_output(sir_ts_in, gamma, 7)
sir_tr_out.head()
```

```
CPU times: user 4.2 s, sys: 6.64 ms, total: 4.21 s
Wall time: 4.21 s
```

Out[6]:

	S	I	R
0	0.201814	0.425756	0.372430
1	0.115945	0.474359	0.409696
2	0.019150	0.511369	0.469481
3	0.078295	0.196566	0.725139
4	0.453265	0.148189	0.398546

- We picked $\gamma = 1/14$ (this will be fixed in our use case)
- For each configuration, we simulate one week

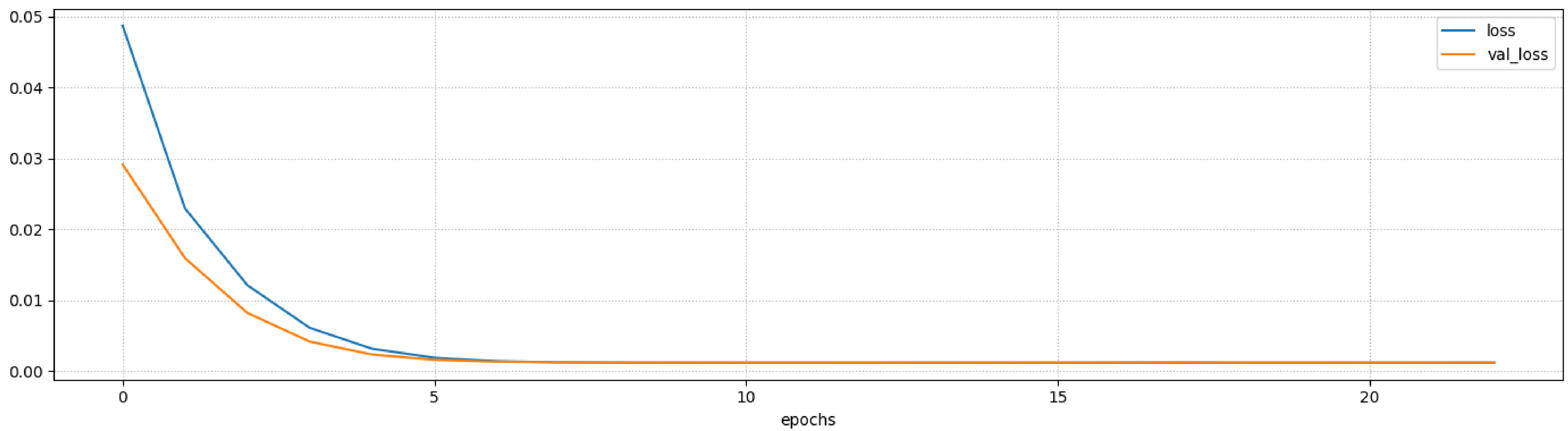
Training a Model

We try with Linear Regression

In [8]:

```
nn0 = util.build_nn_model(input_shape=4, output_shape=3, hidden=[], name='LR')
history0 = util.train_nn_model(nn0, sir_tr_in, sir_tr_out, loss='mse', verbose=0, validation_spl
util.plot_training_history(history0, figsize=figsize)
util.print_ml_metrics(nn0, sir_tr_in, sir_tr_out, 'training')
util.print_ml_metrics(nn0, sir_ts_in, sir_ts_out, 'test')
```

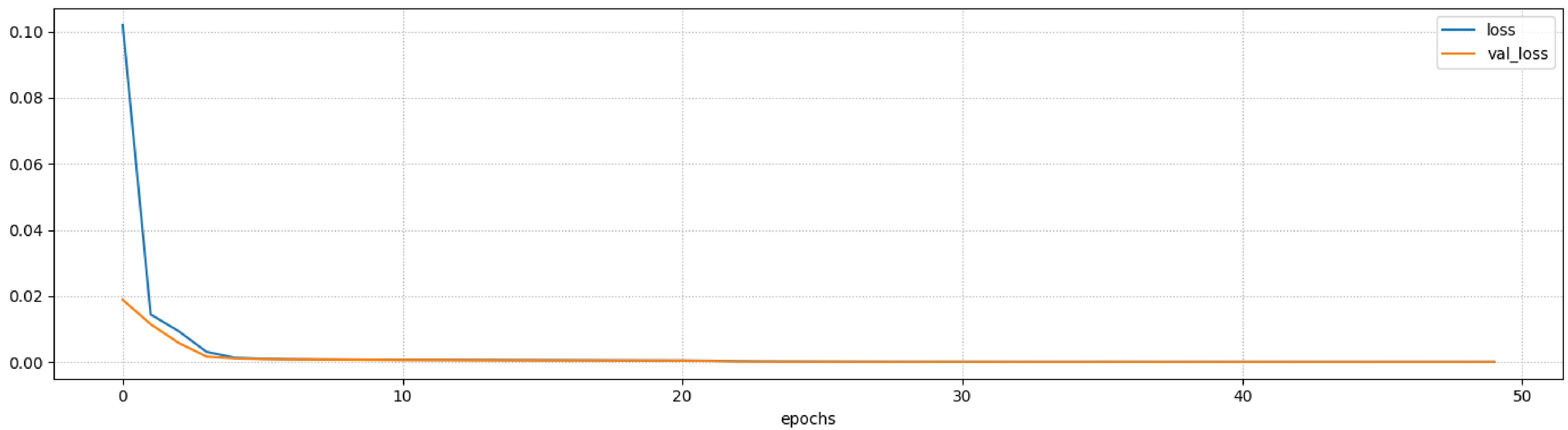
```
2022-12-12 17:06:20.411632: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFl
ow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following C
PU instructions in performance-critical operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
```



Training a Model

...And with a shallow Neural Network

In [9]:
```
nn1 = util.build_nn_model(input_shape=4, output_shape=3, hidden=[8], name='MLP')
history1 = util.train_nn_model(nn1, sir_tr_in, sir_tr_out, loss='mse', verbose=0, validation_spl
util.plot_training_history(history1, figsize=figsize)
util.print_ml_metrics(nn1, sir_tr_in, sir_tr_out, 'training')
util.print_ml_metrics(nn1, sir_ts_in, sir_ts_out, 'test')
```




```
Final loss: 0.0001 (training), 0.0001 (validation)
R2: 0.99, MAE: 0.0072, RMSE: 0.01 (training)
R2: 0.99, MAE: 0.0072, RMSE: 0.01 (test)
```

Considerations and Next Steps

We will save both models for later

```
In [10]: util.save_ml_model(nn0, 'nn0')
         util.save_ml_model(nn1, 'nn1')
```

- The network is much better in terms of accuracy
- ...But the Linear Regressor is simpler!

Hence, the approaches provide different trade offs

We are halfway there

We now have our ML model(s)!

- We need to understand how they can be embedded in an optimization model
- ...And we need to define our optimization model itself