Bayesian (Surrogate-Based) Optimization

Bayesian Optimization

We will use an approach known as Surrogate-Based Bayesian Optimization

- It is designed to optimize blackbox functions
- I.e. functions with an unknown structure, that can only be evaluated

Formally, they address problems in the form:

$$\min_{x \in B} f(x)$$

- Where B is a box, i.e. a specification of bounds for each component of x
- In our case, the decision variable x would be θ
- ...And the function to be optimized would be the cost

The functions are typically assumed to be expensive to evaluate

Why a Surrogate

Since evaluating f is expensive, it should be done infrequently

The main trick to achieve this is using a surrogate model

- After each evaluation we train a Machine Learning model
- ...Then we perform optimization on the ML model
- ...Since it can be evaluate much more quickly

The process is usually start by sampling a few random points

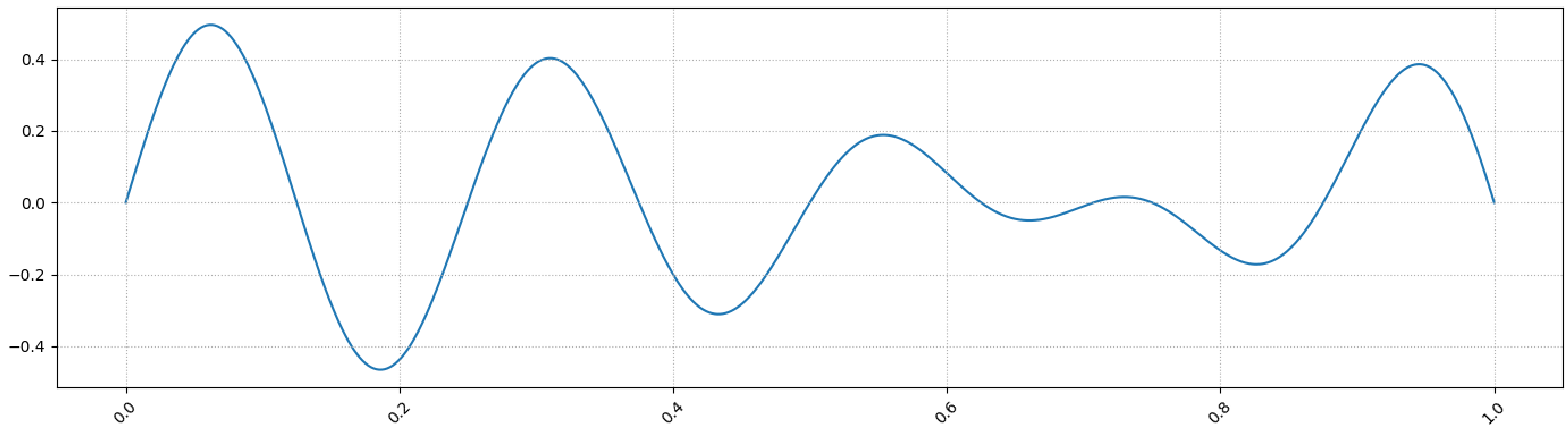
This is where the name stems from

- Since we use the ML model instead of the function, we call it a surrogate
- Moreover, we optimize over prior information (i.e. the current model)
- ...And we refine the model based on the evaluation (posterior)
- Hence we call it Bayesian Optimization

A Running Example

Let's assume we want to minimize the following function over $[0, 1]$

```
In [23]: bbf = lambda x: (0.5 - x**2) * np.sin(2 * 4 * np.pi * x)
         xrange = np.linspace(0, 1, 1000)
         target = pd.Series(index=xrange, data=bbf(xrange))
         util.plot_series(target, figsize=figsize)
```



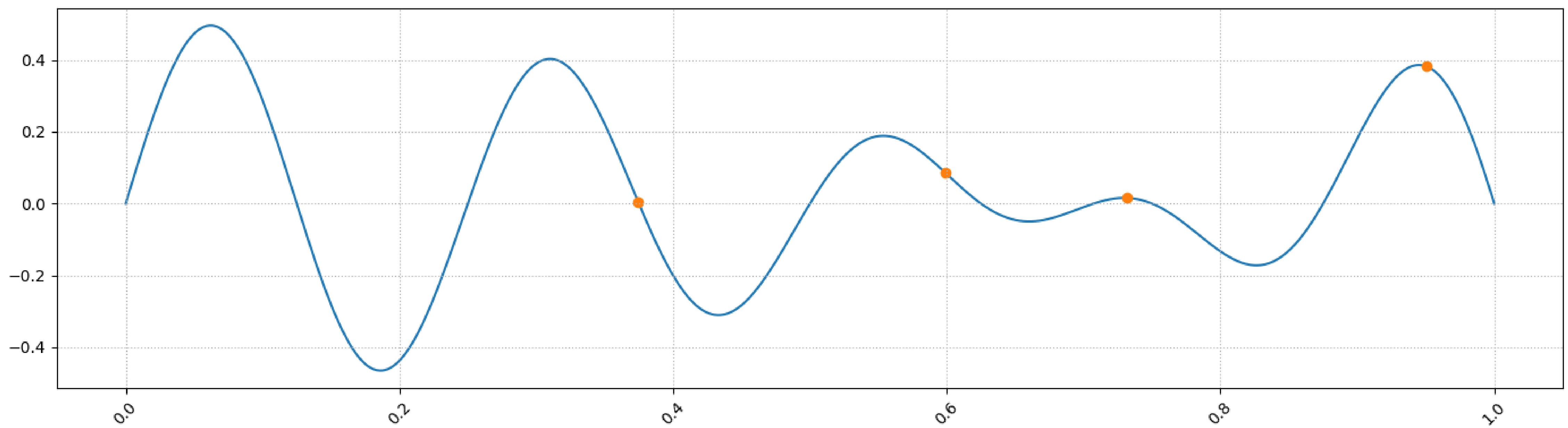
- There multiple local minima, and the global minimum is $\simeq 0.19$

A Running Example

Let's start by sampling a few points at random

In [24]:

```
np.random.seed(42)
xtr = np.sort(np.random.random(4))
ytr = bbf(xtr)
util.plot_series(target, figsize=figsize)
plt.scatter(xtr, ytr, color='tab:orange');
```



Which properties should our surrogate model have?

Properties of a Good Surrogate

Our surrogate model should

Properties of a Good Surrogate

Our surrogate model should

Approximate very accurately all evaluted points

- Assuming the function is deterministic
- ...The available evaluations are exact values

Properties of a Good Surrogate

Our surrogate model should

Approximate very accurately all evaluted points

- Assuming the function is deterministic
- ...The available evaluations are exact values

Reflect our confidence level on unexplored regions

- If we have few samples in a certain region
- ...We might want to search there just to see what the function looks like

Can you think of a ML model with these properties?

Gaussian Processes Surrogate

Gaussian Processes check all the boxes!

- They can interpolate very well known measurements
- They provide a confidence level that decays with distance from observations

Let's try to train a simple GP for our example

In [25]:
```
kernel = RBF(0.01, (1e-3, 1e3)) + WhiteKernel(1e-3, (1e-6, 1e-2))
gpr = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=9, normalize_y=True)
gpr.fit(xtr.reshape(-1, 1), ytr);
gpr.kernel_
```

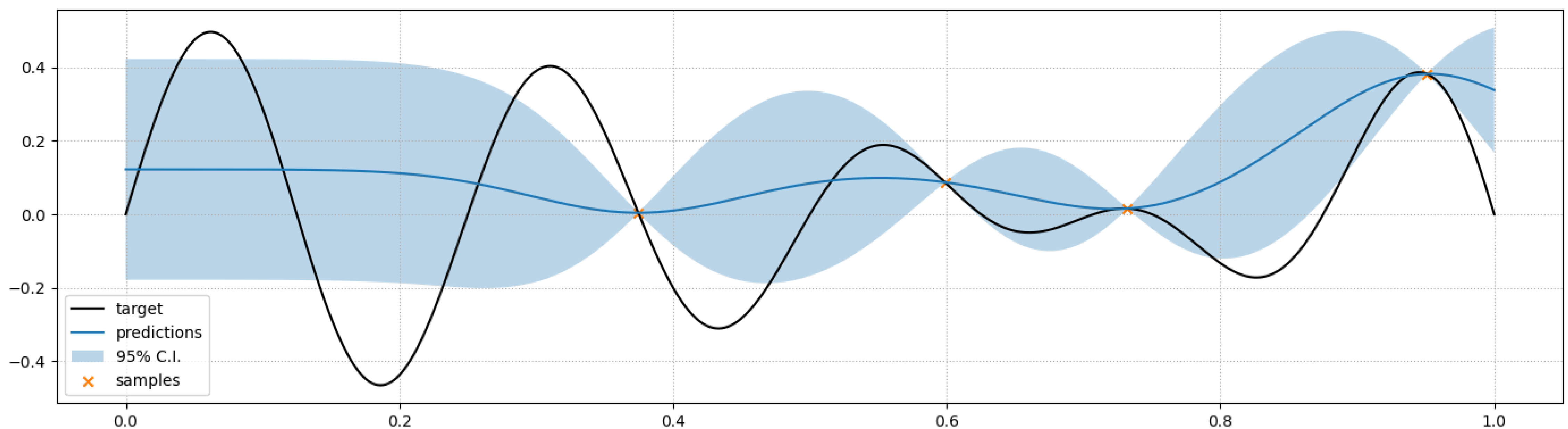
Out[25]:
```
RBF(length_scale=0.0792) + WhiteKernel(noise_level=0.000126)
```

- We use an RBF kernel to capture the distance-based correlation
- We also use a white noise kernel to avoid numerical instability
- ...But we keep it at a low value since the target function is deterministic

Gaussian Process Surrogate

Let's inspect our Gaussian Process Surrogate

```
In [26]: pmean, pstd = gpr.predict(xrange.reshape(-1, 1), return_std=True)
         util.plot_gp(target=target, figsize=figsize, pred=pd.Series(index=xrange, data=pmean),
                      std=pd.Series(index=xrange, data=pstd), samples=pd.Series(index=xtr, data=ytr))
```



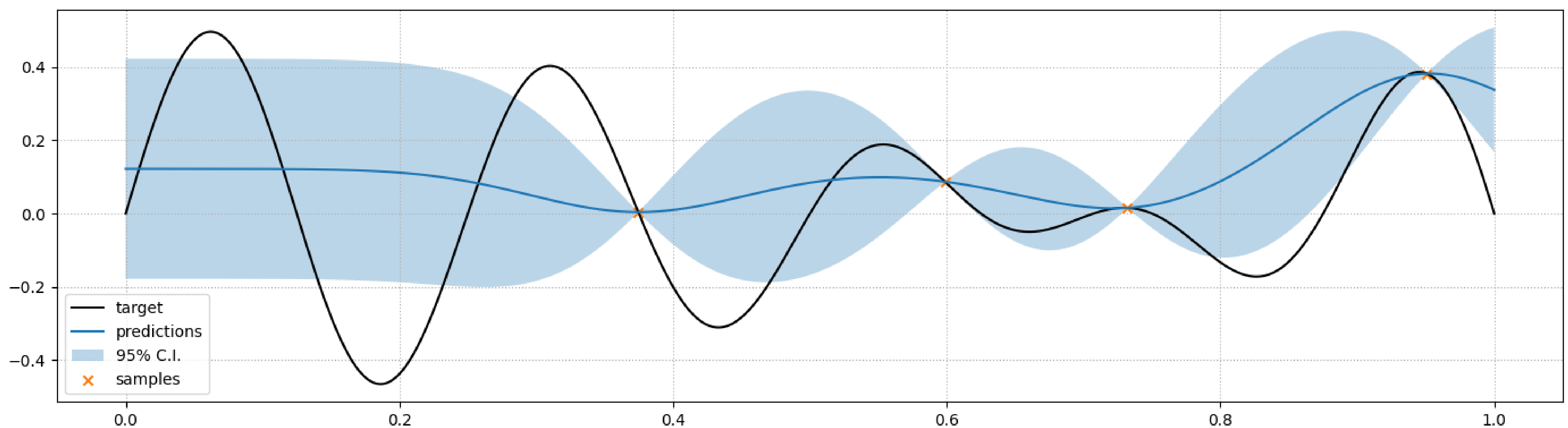
- All known points are interpolated (almost) exactly
- ...And the confidence intervals behave in an intuitive fashion

What to Optimize?

Now we need to search over the surrogate model

This is the same as choosing which function to optimize

In [27]:
```
pmean, pstd = gpr.predict(xrange.reshape(-1, 1), return_std=True)
util.plot_gp(target=target, figsize=figsize, pred=pd.Series(index=xrange, data=pmean),
             std=pd.Series(index=xrange, data=pstd), samples=pd.Series(index=xtr, data=ytr))
```



Which area does it make sense to explore, and why?

Acquisition Function

We need to account for both the predictions and their confidence

- Area with low predictions are promising
- ...But so are also areas with high confidence

This issue is solved in SBO by optimizig an acquisition function

...Which should balance exploration and exploitation.

- Examples include the Probability of improvement, the Expected Improvement
- ...And the Lower/Upper confidence bound

We will use the Lower Confidence Bound, which is given by:

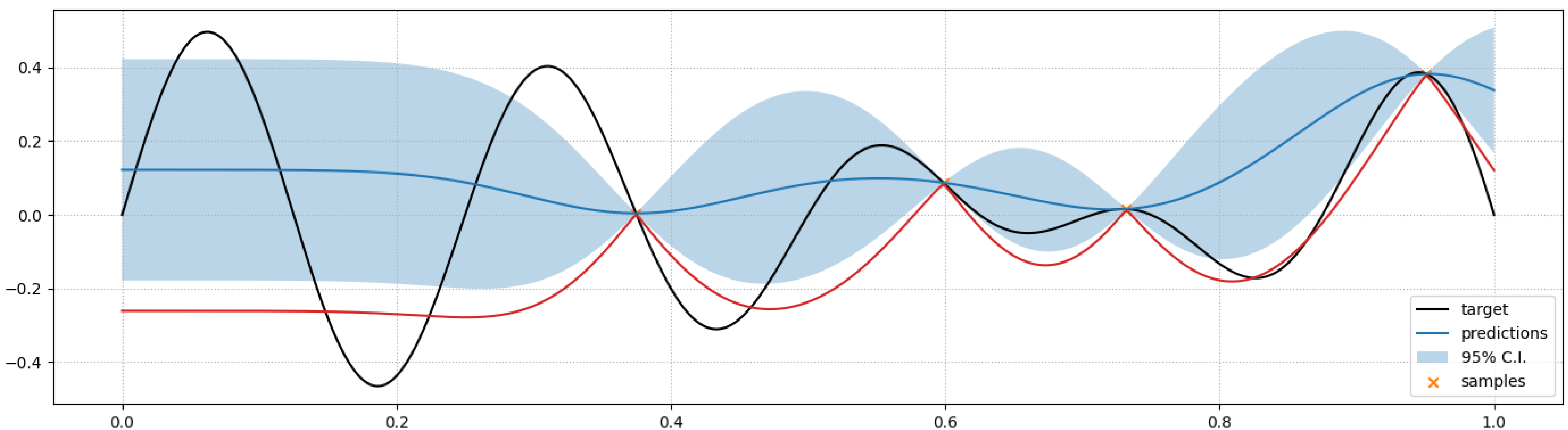
$$LCB(x) = \mu(x) - Z_\alpha \sigma(x)$$

- Where $\mu(x)$ is the predicted mean, $\sigma(x)$ is the predicted standard deviation
- ...And Z_α is multiplier for a $\alpha\%$ Normal confidence inteval

Lower Confidence Bound

Let's see an examle in our case with $Z_\alpha = 2.5$

In [28]:
```
lcb = pmean - 2.5 * pstd
util.plot_gp(target=target, figsize=figsize, pred=pd.Series(index=xrange, data=pmean), std=pd.Se
plt.plot(xrange, lcb, color='tab:red');
```

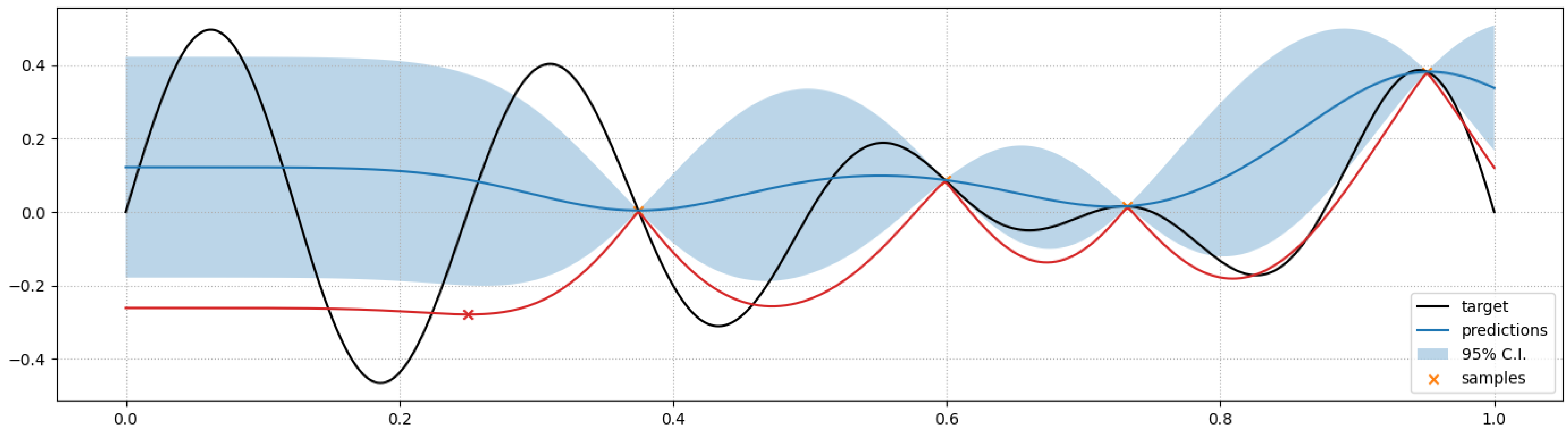


- We can then optimize via any method applicable to our surrogate
- E.g. Nelder-Mead, Mathmatical Programming, or even simple grid search

Lower Confidence Bound

Let's see which point we would choose in our case

In [29]:
```
best_idx = np.argmin(lcb)
util.plot_gp(target=target, figsize=figsize, pred=pd.Series(index=xrange, data=pmean), std=pd.Se
plt.plot(xrange, lcb, color='tab:red');
plt.scatter(xrange[best_idx], lcb[best_idx], marker='x', color='tab:red');
```



- The x value with the best acquisition function is highlighted with a red "x"

Updating the Surrogate

Now wen update our surrogate model

First, we evaluate f for the new point and grow our training set:

```
In [30]: xtr2 = np.hstack((xtr, [xrange[best_idx]]))
         ytr2 = np.hstack((ytr, [bbf(xrange[best_idx])]))
```

Then we can retrain our Gaussian Process:

```
In [31]: gpr2 = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=9, normalize_y=True)
         gpr2.fit(xtr2.reshape(-1, 1), ytr2);
         gpr2.kernel_

Out[31]: RBF(length_scale=0.0999) + WhiteKernel(noise_level=2.46e-06)
```

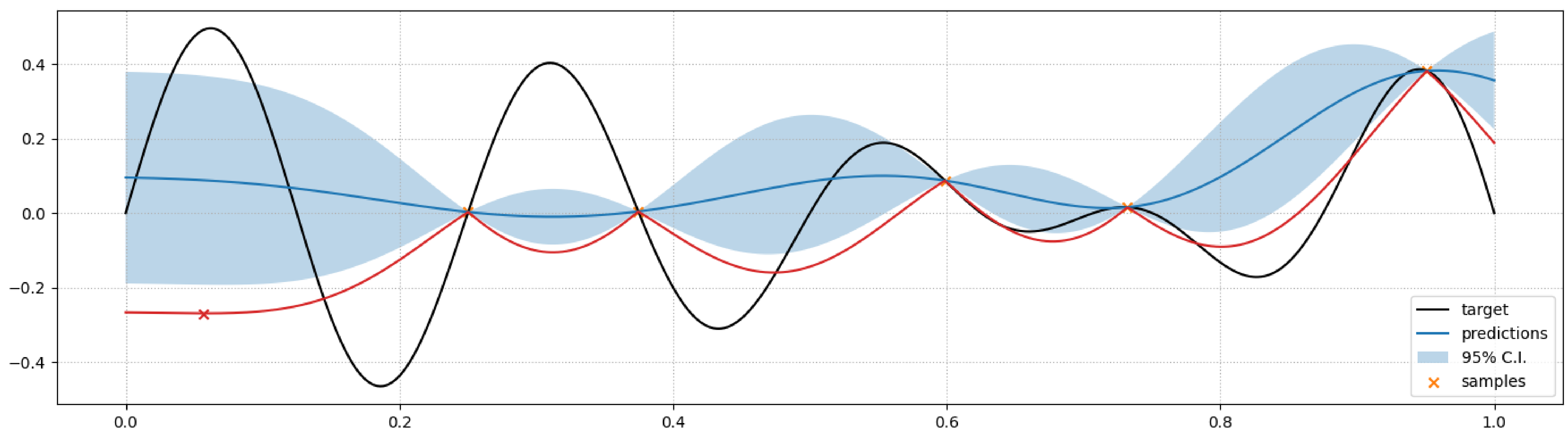
- Then we should optimize the acquisition function again
- ...But we will limit ourselves to showing the updated predictions

Updating the Surrogate

Here are the estimates for the udpate surrorate

...Together with the acquisition function and the next iterate

In [32]:
```
pmean2, pstd2 = gpr2.predict(xrange.reshape(-1, 1), return_std=True)
lcb2 = pmean2 - 2.5 * pstd2
best_idx2 = np.argmin(lcb2)
util.plot_gp(target=target, figsize=figsize, pred=pd.Series(index=xrange, data=pmean2), std=pd.S
plt.plot(xrange, lcb2, color='tab:red');
plt.scatter(xrange[best_idx2], lcb2[best_idx2], marker='x', color='tab:red');
```



Surrogate-Based Bayesian Optimization

Let's review the general method

- Given a collection $\{\hat{x}_i, \hat{y}_i\}_i$ of evaluated points
- ...We train a surrogate-model $\tilde{f}$ for f

Then we proceed as follows:

- We optimize an acquisition function $a_{\tilde{f}}(x)$ to find a value x'
- We evaluate $y' = f(x')$
- If y' is better than the current optimum $f(x^*)$:
  - Then we replace x^* with x'
- We expand our collection of measurements to include (x', y')
- We retrain $\tilde{f}$
- We repeat until a termination condition is reached

A Few Considerations

Different Bayesian optimization algorithms:

- Make use of different surrogate models
- Rely on different criteria for choosing x'
- Strike different trade-offs in terms of number of (expensive) evaluations of f
- ...And the quality of the obtained solutions

For more information, see (e.g.) this tutorial

In practice, you don't have to code from scratch

...Since multiple libraries are available, like:

- The `scikit-optimize` package (crude, but reasonably fast)
- The `bayesian-optimization` python module (more stable, but also slower)
- The RBFOpt solver (based on a powerful non-linear optimization solver)

SBO for Threshold Calibration

Back to Our Motivating Problem

We will use SBO to tackle our policy definition problem

$$\begin{aligned} &\mathop{\text{argmin}}_{\theta} \sum_{k \in K} cost(f(\hat{x}_k\ \omega^*), 1/2) \\ &\text{s.t.: } \omega^* = \mathop{\text{argmin}}_{\omega} L(f(\hat{x}_k, \lambda), \mathbb{1}_{y_k \geq \theta}) \end{aligned}$$

Here's our plan:

- We need to optimize over θ
- Our goal is minimizing the cost
- Computing the cost requires to re-define the classes
- ...And therefore to repeat training

Our implementation will be based on **scikit-optimize**

The Black Box Function

As a first step, we need to define our black box function

We will use a function class (in the `util` module) with this structure:

```
class ClassifierCost:
    def __init__(self, machines, X, y, cost_model, init_epochs=20, inc_epochs=3):
        ...

    def __call__(self, params):
        ...
```

- In the constructor, we provide parameters that are fixed during optimization
- In the `__call__` method, we retrain the model and evaluate the cost
- The `__call__` method is executed when we try to invoke an object of this class
- ...Meaning that we can treat an object of this class as a normal function

The Black Box Function

It is worth having a deeper look at the `__call__` method

```
def __call__(self, params):
    theta = params[0] # There is only one parameter to optimize
    lbl = (self.y >= theta) # Redefine classes
    # Determine the number of epochs and retrain
    epochs = self.init_epochs if not self.is_init else self.inc_epochs
    self.is_init = True
    train_nn_model(self.nn, self.X, lbl, loss='binary_crossentropy', epochs=epochs,
            verbose=0, patience=10, batch_size=32, validation_split=0.2)
    ...
```

- At each execution we redefine the classes
- We use warm starting to make the process faster
- Each training attempt after the first uses only a few epochs

The Black Box Function

It is worth having a deeper look at the `__call__` method

```
def __call__(self, params):
    ...
    self.stored_weights[theta] = self.nn.get_weights() # Store weights
    # Evaluate cost
    pred = np.round(self.nn.predict(self.X, verbose=0).ravel())
    cost, fails, slack = self.cost_model.cost(self.machines, pred, 0.5, return_margin=Tru
    return cost
```

- We store the weights in a dictionary for later retrieval
- We need this to rebuild the optimal network once optimization is over
- Finally, we evaluate the cost
- The actual code in `util` also prints some information

The Black Box Function

We can build an object in the usual way

```
In [47]: ccf = util.ClassifierCost(machines=tr['machine'], X=tr_s[dt_in], y=tr['rul'], cost_model=cmodel)
```

...But since it is a function, we can invoke it:

```
In [48]: ccf([20])

theta: 20.00, avg. cost: -87.46, avg. fails: 0.00, avg. slack: 28.96

Out[48]: -16268
```

- We pass an iterable type for compatibility with `scikit-optimize`
- ...Which is designed for multivariate optimization

Running the Solver

Now we can define our box constraints and run the optimization process

```
In [49]: box = [(1, 18)]
         res = skopt.gp_minimize(ccf, dimensions=box, acq_func='LCB', n_calls=10,
                   n_random_starts=3, noise=None, random_state=42, verbose=False)
```

```
theta: 15.00, avg. cost: -90.99, avg. fails: 0.00, avg. slack: 25.37
theta: 4.00, avg. cost: -84.63, avg. fails: 0.03, avg. slack: 10.68
theta: 14.00, avg. cost: -92.74, avg. fails: 0.00, avg. slack: 23.58
theta: 11.00, avg. cost: -95.79, avg. fails: 0.00, avg. slack: 20.46
theta: 9.00, avg. cost: -101.91, avg. fails: 0.00, avg. slack: 14.17
theta: 8.00, avg. cost: -100.84, avg. fails: 0.00, avg. slack: 15.36
theta: 18.00, avg. cost: -91.84, avg. fails: 0.00, avg. slack: 24.46
theta: 9.00, avg. cost: -100.59, avg. fails: 0.00, avg. slack: 15.47
theta: 7.00, avg. cost: -104.09, avg. fails: 0.00, avg. slack: 11.97
theta: 6.00, avg. cost: -92.20, avg. fails: 0.02, avg. slack: 10.24
```

- The `scikit-optimize` implementation is very close to what we have showed
- ...And in some cases it might revisit previously used θ values

Retrieve the Results

The result data structure contains a lot of detail

```
In [52]: print(list(res.keys()))
```

```
['x', 'fun', 'func_vals', 'x_iters', 'models', 'space', 'random_state', 'specs']
```

The best θ value is in the `x` field

```
In [53]: res.x
```

```
Out[53]: [7]
```

We will use it to retrive the weights of the best network:

```
In [55]: nn = keras.models.clone_model(ccf.nn)
         nn.set_weights(ccf.stored_weights[res.x[0]])
```

Evaluate the Classifier

AutoML

Many ML models have hyper parameters!

...And tuning them may sometimes improve the performance

- The problem is that tuning multiple parameters may be complicated
- ...And every training attempts is expensive

This makes hyper-parameter tuning a perfect application for SBO

...And other similar approaches. A few libraries you might have heard of:

- Hyperopt
- Optuna

In recent years the concept has been generalized to AutoML

...Where we can start chanking the architecture and model type, too!

- It's a big topic (and big techs have some available SW solutions)
- A good starting reference is this web site