Hyperparameter Tuning

Hyperparameter Optimization

- Also called metaparameter optimization and hyperparameter tuning
- Any system that chooses hyperparameters automatically
- What's the difference between the model parameters, and the hyper-parameters?
- A model parameter is a configuration variable that is internal to the model and whose value can be estimated from data (e.g., weight values in neural networks).
- A model hyper-parameter is a configuration that is external to the model and whose value cannot be estimated from data (e.g., learning rate, number of layers, number of nodes).

Hyperparameter Optimization

- Hyperparameter optimization is represented in equation form as:

$$x^* = \underset{x \in X}{\operatorname{argmin}} f(x)$$

- Here f(x) represents an objective score to minimize evaluated on the validation set;
  - x^* is the set of hyperparameters that yields the lowest value of the score
  - x can take on any value in the domain X.
  - We want to find the hyperparameters that yield the best score on the validation set metric.

- The problem with hyperparameter optimization is that evaluating the objective function to find the score is very expensive.
- Each time we try different hyperparameters, we have to train a model on the training data, make predictions on the validation data, and then calculate the validation metric.
- With a large number of hyperparameters and complex models, this process quickly becomes intractable to do by hand!

https://towardsdatascience.com/a-conceptual-explanation-of-bayesian-model-based-hyperparameter-optimization-for-machine-learning-b8172278050f

Basic Hyper-parameter Tuning Algorithms

- Tuning by hand
- Grid search
- Random search
- Bayesian optimization
- Evolutionary Optimization

Tuning By Hand

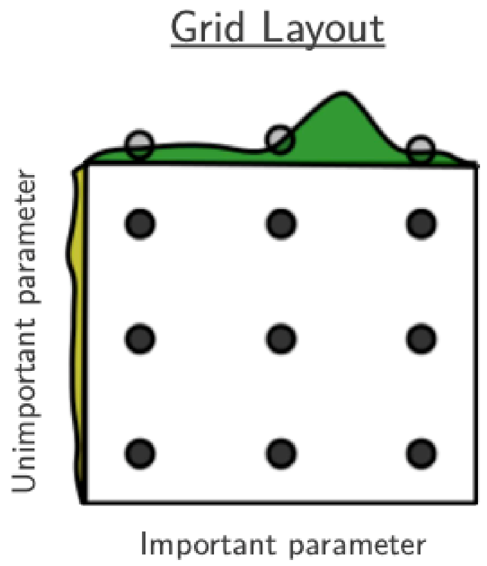
- Just fiddle with the parameters until you get the results you want
- Probably the most common type of hyperparameter optimization

- Upsides: the results are generally pretty good…
- To be honest, the results you get from just using the folklore settings are really not that bad for a lot of practical purposes.

- Downsides: lots of effort, and no theoretical guarantees
- Although there's nothing fundamental that prevents us from having theory here

Grid Search

$$\lambda^* = \operatorname{argmin}_{\lambda \in \Lambda} f_{\mathcal{A};\, D_{train}}(\lambda)$$



- Easy to implement
- Trivial to parallelize
- Curse of dimensionality

Grid Search

- Define some grid of parameters you want to try
- Try all the parameter values in the grid
- By running the whole system for each setting of parameters
- Then choose the setting with the best result
- Essentially a brute force method

Downsides of Grid Search

- As the number of parameters increases, the cost of grid search increases exponentially
- It often takes a lot of processing power and time to fit the model with every potential combination, which might not be available.
- Still need some way to choose the grid properly
- Something this can be as hard as the original hyperparameter optimization
- Can’t take advantage of any insight you have about the system

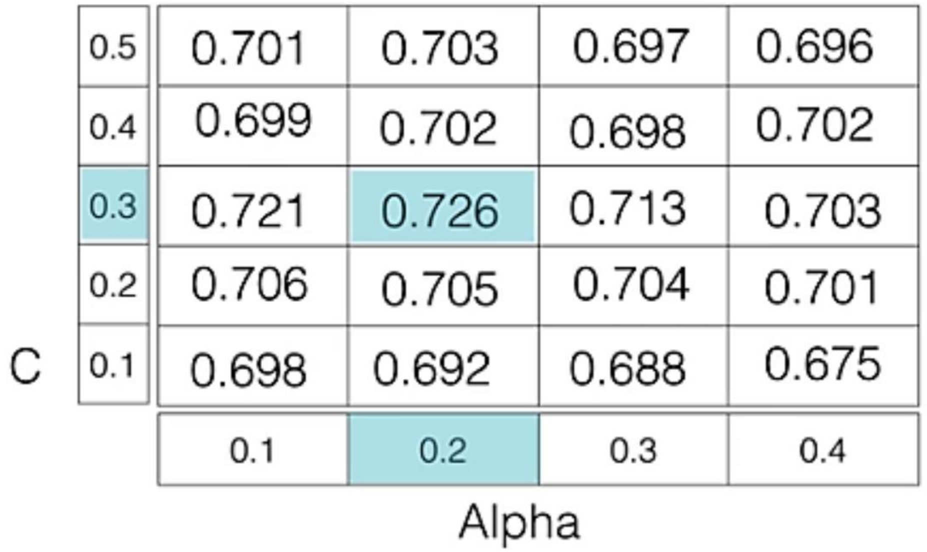
Grid Search

Making Grid Search Fast

- Early stopping to the rescue
- Can run all the grid points for one epoch, then discard the half that performed worse, then run for another epoch, discard half, and continue.
- Can take advantage of parallelism
- Run all the different parameter settings independently on different servers in a cluster.
- Very easy to parallelize this process
- Downside: doesn’t reduce the energy cost.

Grid Search

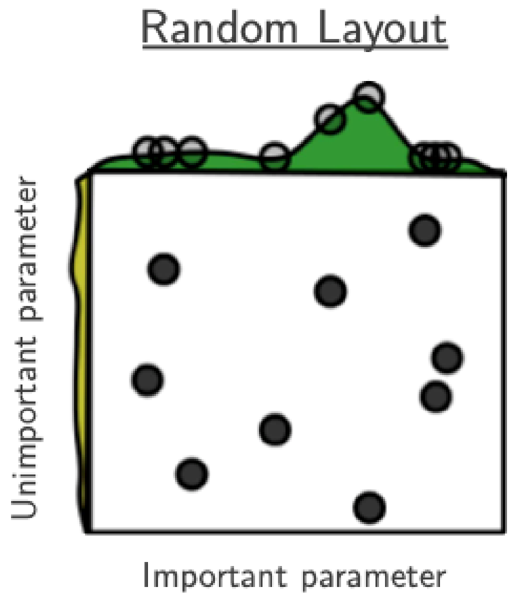
- For example: if we want to set two hyperparameters C (regularization constant) and Alpha (learning rate) of the Logistic Regression Classifier model, with different sets of values.
- The grid search technique will construct many versions of the model with all possible combinations of hyperparameters and will return the best one.
- As in the image, for C = [0.1, 0.2, 0.3, 0.4, 0.5] and Alpha = [0.1, 0.2, 0.3, 0.4].
- For a combination of *C=0.3 and Alpha=0.2*, the performance score comes out to be 0.726(Highest), therefore it is selected.



C \ Alpha	0.1	0.2	0.3	0.4
0.5	0.701	0.703	0.697	0.696
0.4	0.699	0.702	0.698	0.702
0.3	0.721	0.726	0.713	0.703
0.2	0.706	0.705	0.704	0.701
0.1	0.698	0.692	0.688	0.675

Random Search

$$\lambda^* = \operatorname{argmin}_{\lambda \in \Lambda} f_{\mathcal{A};\, D_{train}}(\lambda)$$



- Usually works better than naïve grid search
- High variance (rely on luck)

Random Search

- This is just grid search, but with randomly chosen points instead of points on a grid.
- This solves the curse of dimensionality
- Don’t increase the number of grid points exponentially as the number of dimensions increases.
- This method selects values at random as opposed to the grid search method’s use of a predetermined set of numbers.
- Every iteration, random search attempts a different set of hyperparameters
- It returns the combination that provided the best outcome after several iterations. This approach reduces unnecessary computation.
- **Advantages:**
    - In most cases, a random search will produce a comparable result faster than a grid search.
    - Very easy to parallelize this process
- **Drawback:** It’s possible that the outcome could not be the ideal hyperparameter combination

Grid search in sklearn

```
from sklearn.model_selection import GridSearchCV

# Define the hyperparameters and their possible values
param_grid = {
    'alpha': [0.01, 0.1, 1.0, 10.0],
    'beta': [0.01, 0.1, 1.0, 10.0]
}

# Create a model
model = SomeModel()

# Use grid search to find the optimal hyperparameters
grid_search = GridSearchCV(model, param_grid)
grid_search.fit(X, y)

# Print the optimal values for the hyperparameters
print(grid_search.best_params_)
```

Random search in sklearn

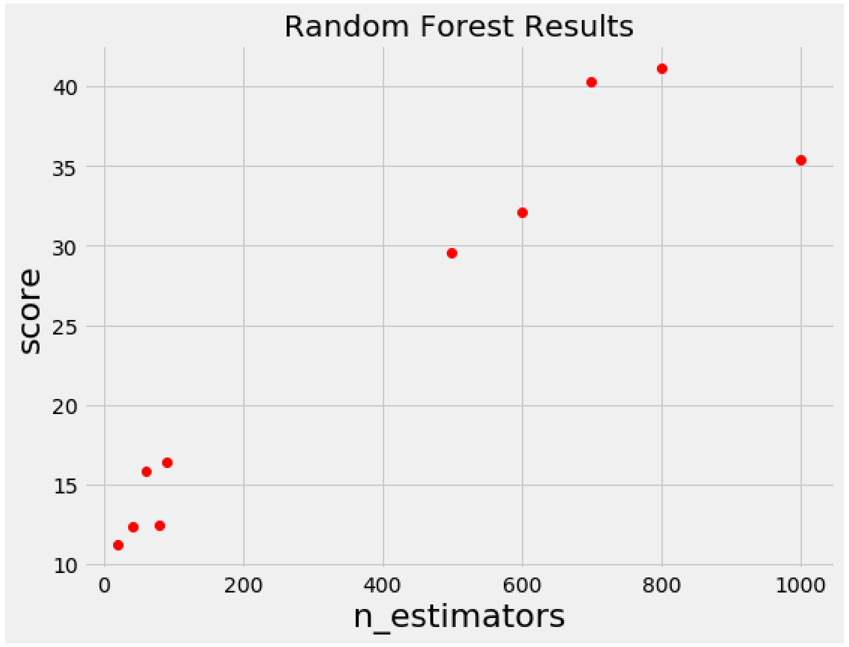
```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import uniform

# Define the hyperparameters and their distributions
param_distributions = {
    'alpha': uniform(0.01, 10.0),
    'beta': uniform(0.01, 10.0) }

# Create a model
model = SomeModel()

# Use randomized search to find the optimal hyperparameters
random_search = RandomizedSearchCV(model, param_distributions)
random_search.fit(X, y)

# Print the optimal values for the hyperparameters
print(random_search.best_params_)
```

Grid/Random Search

- Grid and random method are relatively inefficient because they do not choose the next hyperparameters to evaluate based on previous results.

- Grid and random search are completely uninformed by past evaluations, and as a result, often spend a significant amount of time evaluating “bad” hyperparameters.

- If we have the following graph with a lower score being better, where does it make sense to concentrate our search?
- Random and grid search pay no attention to past results at all and would keep searching across the entire range of the number of estimators even though it’s clear the optimal answer (probably) lies in a small region!



Bayesian Optimization

Bayesian Optimization

- Grid search and random search are often inefficient because they evaluate many unsuitable hyperparameter combinations without considering the previous results.
- Bayesian optimization treats the search for optimal hyperparameters as an optimization problem.
- It considers the previous evaluation results when selecting the next hyperparameter combination and applies a probabilistic function to choose the combination that will likely yield the best results.
- This method discovers a good hyperparameter combination in relatively few iterations.

Bayesian Optimization

- Hyperparameter tuning is a kind of black-box optimization: you want to minimize an error function $f(x)$, where x is hyperparameter.

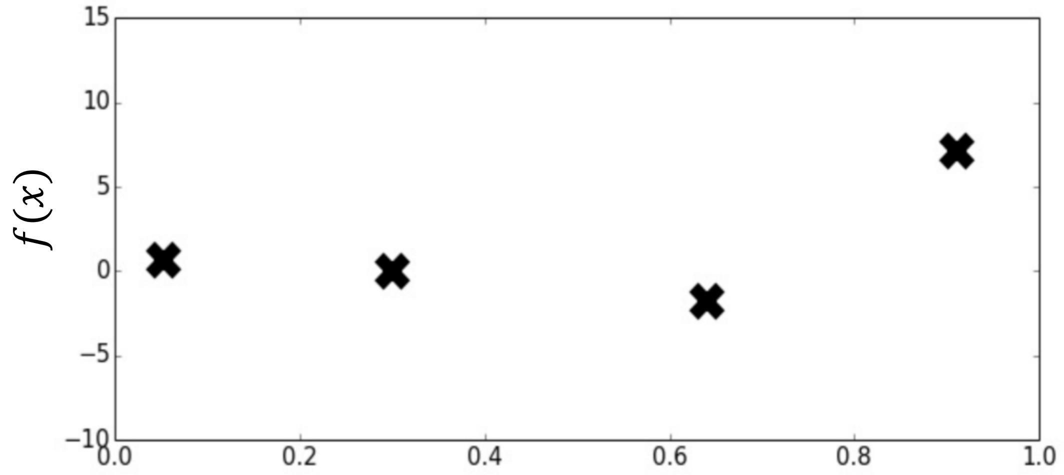
- In hyperparameter tuning, we can NOT use gradient method. We can not compute the gradient of $f(x)$ since x (hyperparameters) are given by users, not trained from data.

- You only get to function values given hyperparameters, not compute gradients
  - Input x : a configuration of hyperparameters
  - Function value $f(x)$: error on the validation set

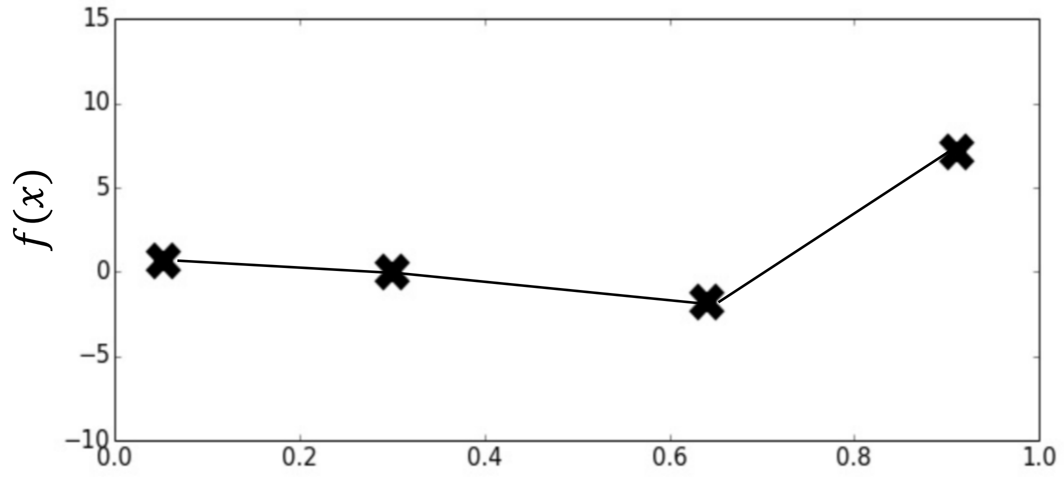
- Problems:
  - Target function is not known (black-box; non-parametric).
  - Each evaluation is expensive, so we want to use few evaluations.
- Goal: How do we find the optimal hyperparameter values that minimize the value of $f(x)$

Bayesian Optimization

- Suppose you have the following initial observations. Where would you try next?



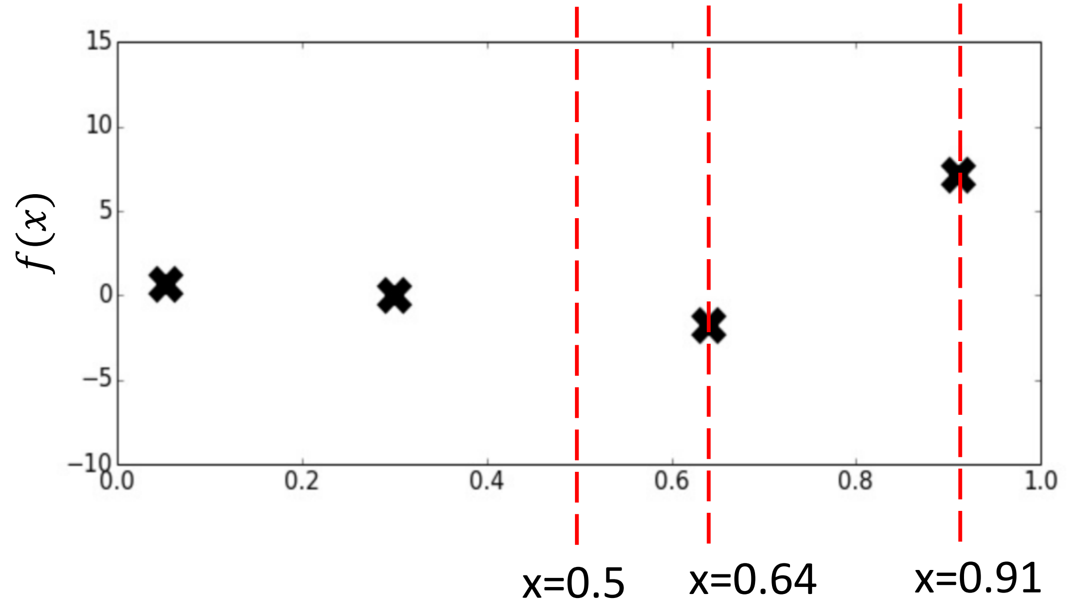
- One simple approach is to connect all the points with straight lines, and then use these lines to predict data points within the range of x=0.02 to x=0.91
  - we have no evidence to believe that the observations are related in this linear manner
  - we cannot predict data points outside of the region [0.02, 0.91]
  - we have no uncertainty measurements on predicted data points.



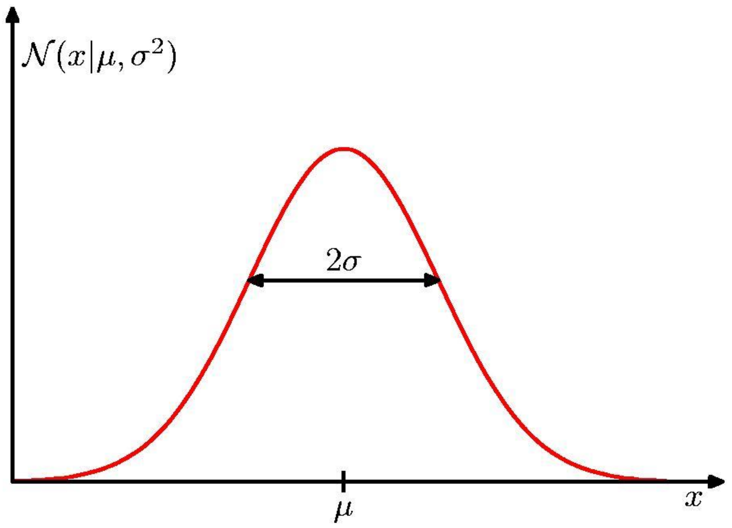
Bayesian Optimization

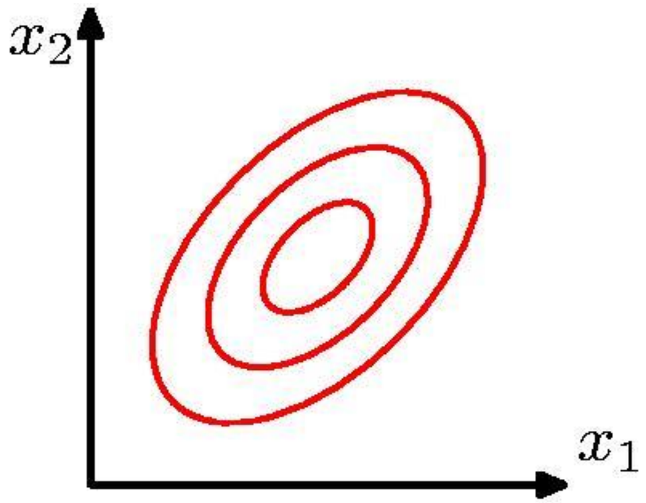
- Since our goal is to minimize $f(x)$, the points near x=0.91 don't look promising
- Current best point is x=0.64, and the points near x=0.64 look promising
- But, the points near x=0.5 may have better answer (high variance)

- Given these observation values, you want to query a point which:
  - you expect to be good
  - you are uncertain about
- How can we model our uncertainty about the function?



Gaussian Distributions

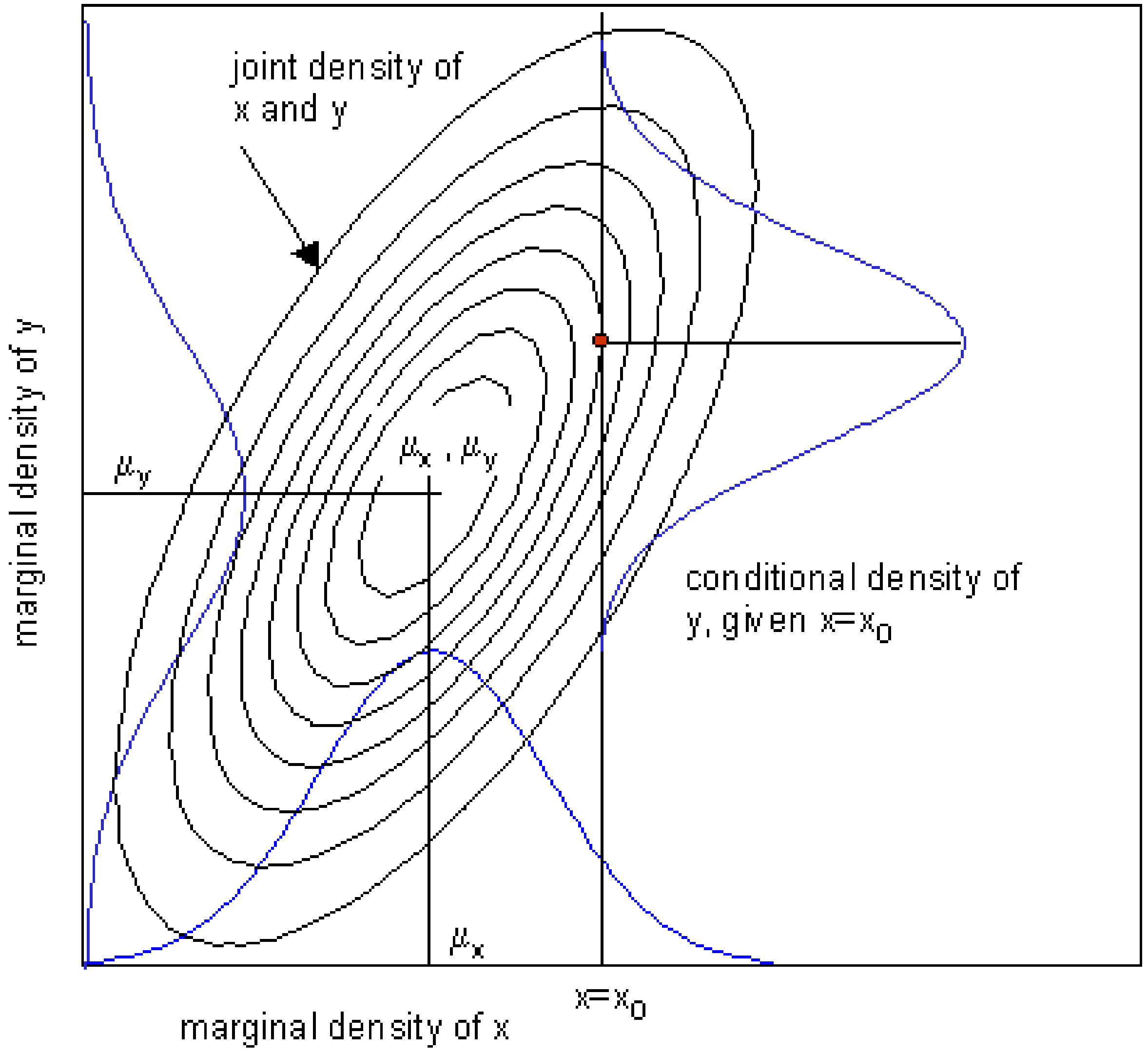




$$p(x;\mu,\sigma^2) = \frac{1}{\sigma\sqrt{2\pi}}\exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right)$$

$$p(x;\mu,\Sigma) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}}\exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right)$$

Multivariate Gaussian



https://statistical-engineering.com/probability-and-statistics/joint-marginal-conditional-distributions/

Gaussian Process

- From now on, we need to change our view of a function

- In Gaussian Process, a function is simply defined as “an array of numbers”
- We can represent $f(\cdot)$ compactly as an m-dimensional vector (array of m numbers)

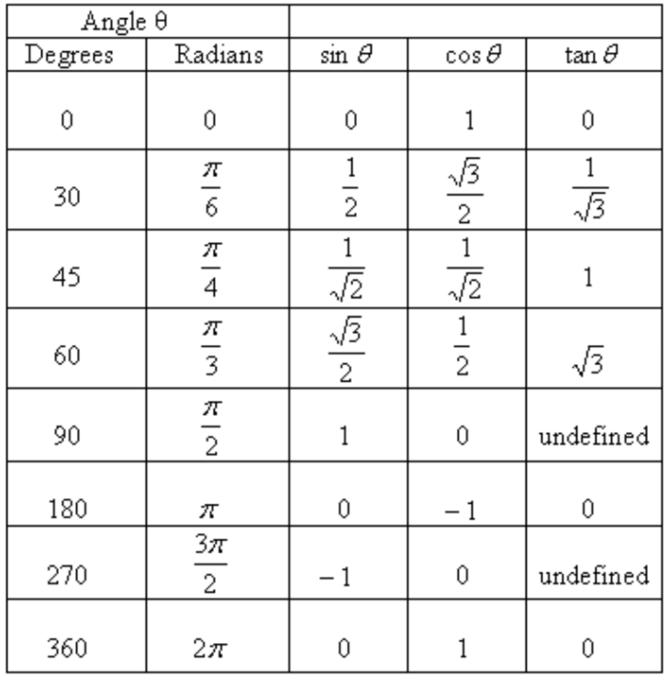
$$f = [f(x_1)\ f(x_2,)\ \ldots\ f(x_m)]^T$$

- For instance, one example of a function $f_0(\cdot) \in \mathcal{H}$ is defined as an array of numbers.

$$f_0(x_1) = 5.\ f_0(x_2) = 2.3, f_0(x_3) = -7, \ldots., f_0(x_m) = 8$$

- Therefore,

$$f_0 = [5, 2.3, -7, \ldots, 8]^T$$



Angle θ				
Degrees	Radians	sin θ	cos θ	tan θ
0	0	0	1	0
30	$\frac{\pi}{6}$	$\frac{1}{2}$	$\frac{\sqrt{3}}{2}$	$\frac{1}{\sqrt{3}}$
45	$\frac{\pi}{4}$	$\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}}$	1
60	$\frac{\pi}{3}$	$\frac{\sqrt{3}}{2}$	$\frac{1}{2}$	$\sqrt{3}$
90	$\frac{\pi}{2}$	1	0	undefined
180	π	0	-1	0
270	$\frac{3\pi}{2}$	-1	0	undefined
360	2π	0	1	0

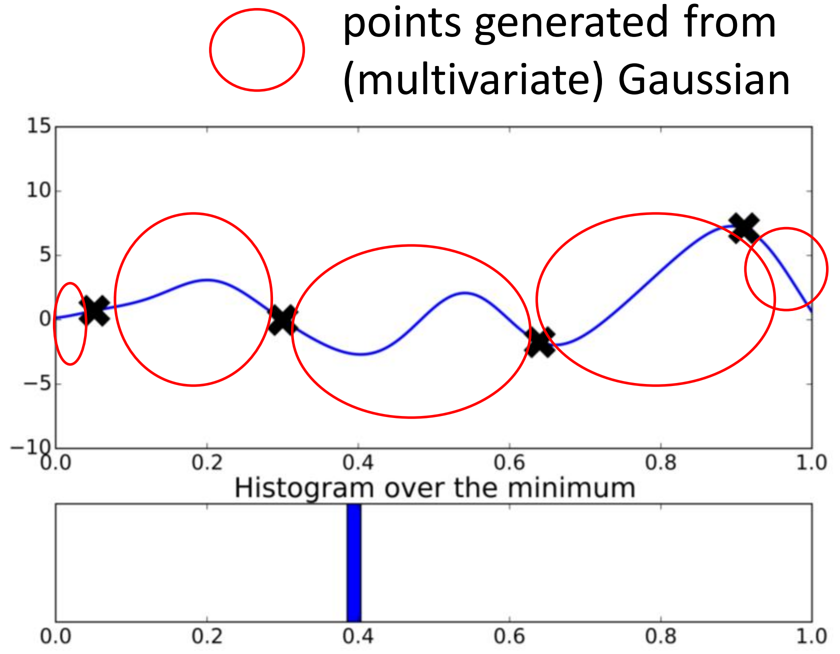
https://towardsdatascience.com/understanding-gaussian-process-the-socratic-way-ba02369d804

Bayesian Optimization

- We propose a different approach (called Gaussian Process Regression)

- In the empty regions, the correct function values only obtained by experiments, which is very expensive
- Instead, we will randomly generate observations by sampling points from a (multivariate) Gaussian distribution. (assume x is multivariate)
- In the picture, the points in red circles are generated from a multivariate Gaussian distribution

- Now we generated a function that is based on
  - observation points given by initial experiments and
  - estimated points from (multivariate) Gaussian distribution
- This function is called *surrogate* function
- Surrogate function is an approximation of the real target function
- The curve in the picture is an example of a function

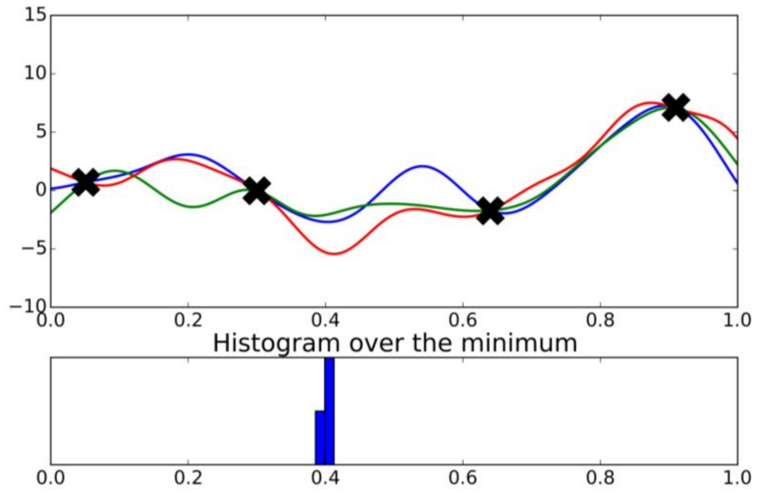


Bayesian Optimization

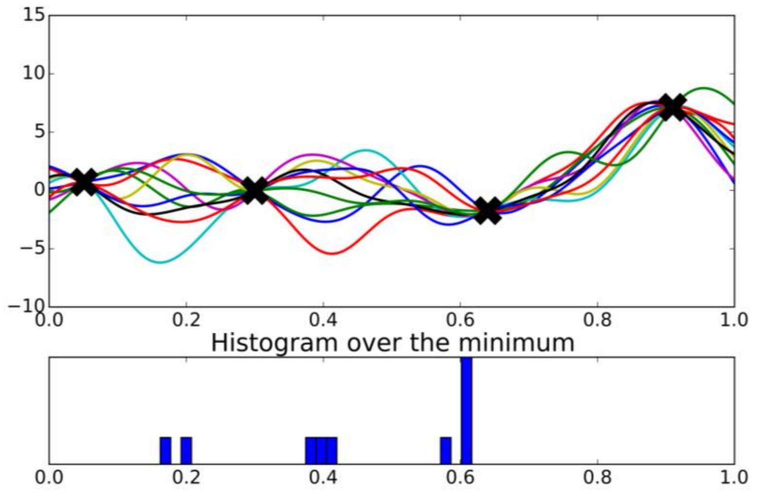
- We repeat this process and can generate many functions.
- These pictures show the results of generating 3,10, and many functions, respectively.

- We see that the functions generated are generally smooth.
- This is because a Gaussian process is able to create smooth lines by using a kernel function which states how data points are related to one another.
- If x1 and x2 are close, f(x1) and f(x2) are close too (smoothness of function)

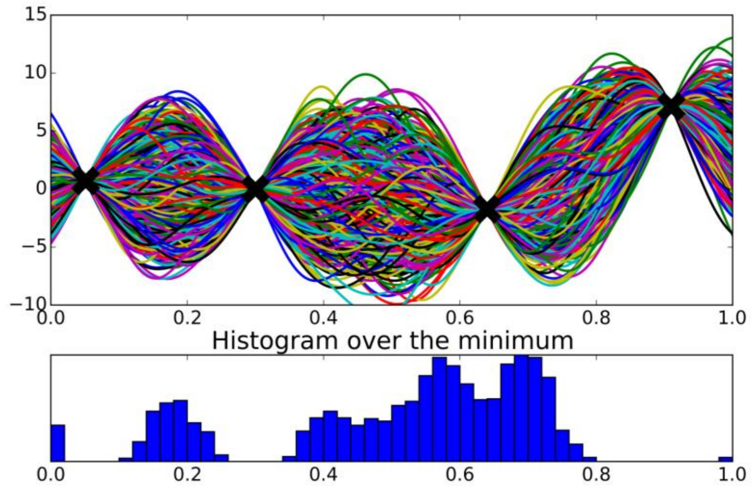
k=3



k=10



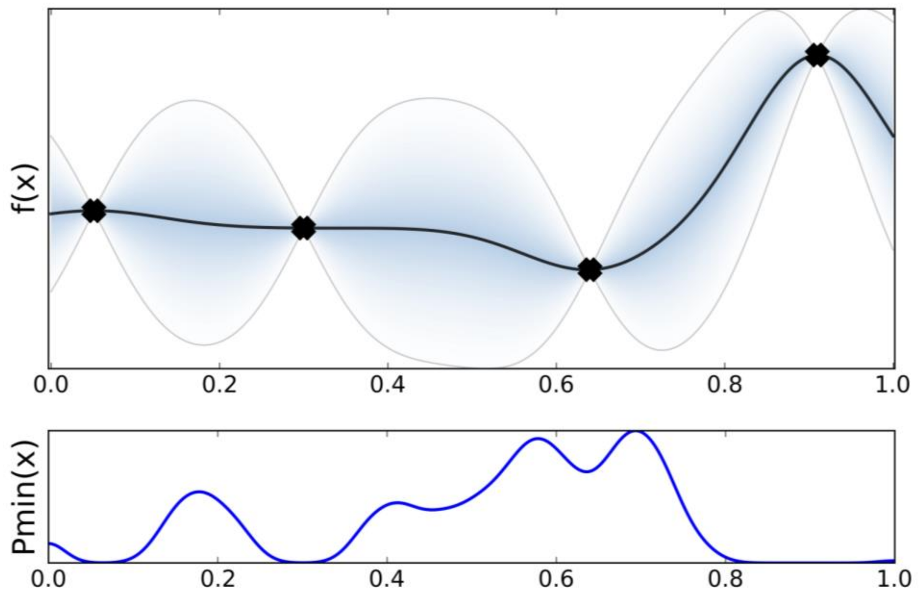
k=many



Bayesian Optimization

- After generating many (possibly *infinite*) functions, we have the following set of functions
  - The solid line means the average values of functions
  - The shaded areas represent the size of standard deviation (sd=1) of the function distribution

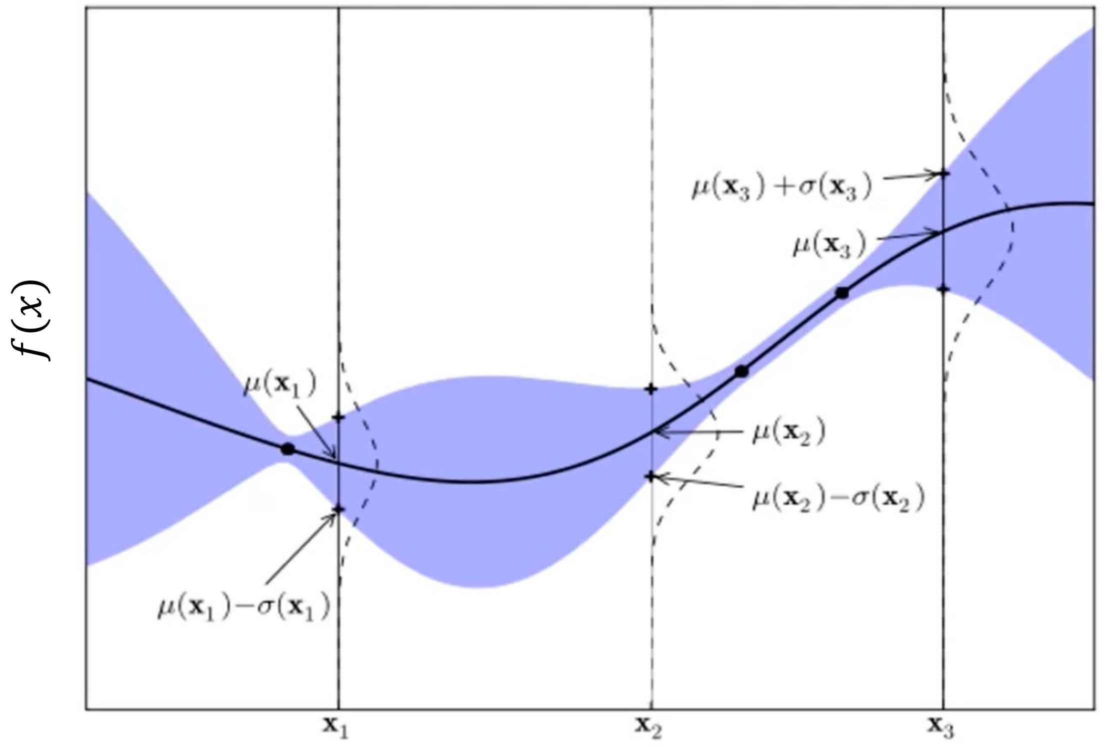
- We assumed that each function is generated from a multivariate Gaussian distribution.
- Since the addition of Gaussian is also Gaussian, it is known that for a certain x value, f(x) value also follows Gaussian distribution



Gaussian Process

- For a certain value of x (e.g.,x_*), we can see that the values of $f(x_*)|X$ form Gaussian distribution (X: observations)
- Solid line represents the mean of Gaussian and the shaded region shows the range of one standard deviation
- For example, for the value of x_*, $f(x_*)$ follows a Gaussian distribution with mean=μ^* and standard deviation=Σ^* where ($\vec{y}$ means $f(x)$ values)

$$\mu^* = k(x_*, x)K(x, x)^{-1}\vec{y}$$
$$\Sigma^* = k(x_*, x_*) - k(x_*, x)K(x, x)^{-1}k(x, x_*)$$



- Here, $K(x_*, x)$ is the kernel function of multivariate Gaussian, x is the current observation vector and $\vec{y}$ is the vector of its corresponding value. e.g.:

$$k(x_i, x_j) = \exp(-\frac{1}{2}\left\|x_i - x_j\right\|^2)$$

Gaussian Process Regression

- This lets us find the marginal distribution of $f(x_*)$ at a new test value x_* conditioned on the values of the function f we've already observed.
- Explicitly, if we define k as above, define the vector k_* as

$$f(x_*)|x_1, x_2, \ldots x_m \sim N(\mu^*, \Sigma^*)$$

where

$$\mu^* = k(x_*, x)K(x, x)^{-1}y$$
$$\Sigma^* = k(x_*, x_*) - k(x_*, x)K(x, x)^{-1}k(x, x_*)$$

- Kenel function:

$$k_*(x, x_*) = [k(x_1, x_*)\ k(x_2, x_*)\ \ \ldots k(x_m, x_*)]$$
$$k_*(x_*, x) = [k(x_*, x_1)\ k(x_*, x_2)\ \ \ldots k(x_*, x_m)]$$

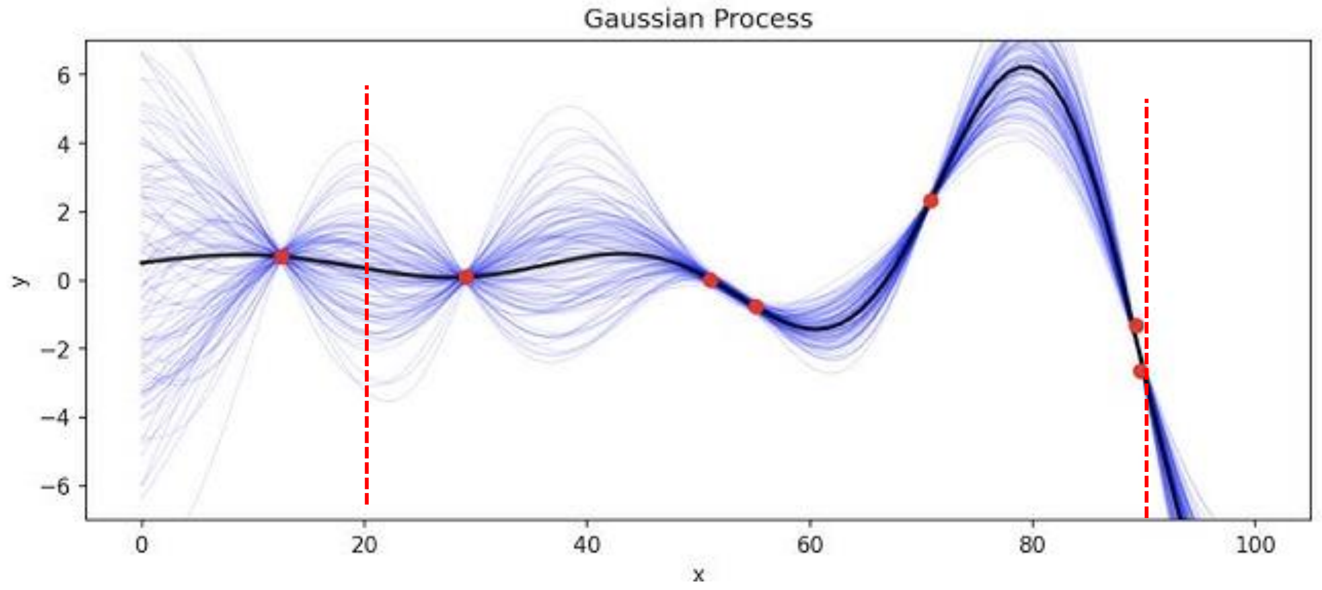
$$K(x, x) = \begin{bmatrix} k(x_1, x_1) & \cdots & k(x_1, x_m) \\ \vdots & \ddots & \vdots \\ k(x_m, x_1) & \cdots & k(x_m, x_m) \end{bmatrix}$$

- $\mu^* = k(x_*, x)K(x, x)^{-1}y$ is same as the value of kernel regression (will discuss in Kernel chapter)

Exploration vs Exploitation

- Given observation points, where should we evaluate next in order to improve the most?.

- In this example, let’s compare two possible points x=20 and x=90.
- We know that given x, the value of $f(x)$ follows Gaussian.
- Suppose $f(x=20) \sim N(0.4; 4.0)$ and $f(x=90) \sim N(-2; 0.1)$
- Which one do we choose?
- x=20 has higher mean with high variance, while x=90 has lower mean and lower variance



x	f(x)	
10	0.5 ± 0.1	
20	0.4 ± 4.0	exploration
…	.	
90	-2 ± 0.1	exploitation

Acquisition Functions

- To choose the next point to query, we must define an acquisition function which takes into account the following two points.
  - Exploration: choose a point which has high uncertainty to evaluate
  - Exploitation: choose a point which has high expected performance to evaluate

1) Expected Improvement(EI): perhaps the most used acquisitions

- Evaluates $f(x)$ at the point that, in expectation, improves upon $f(x_{best})$ the most

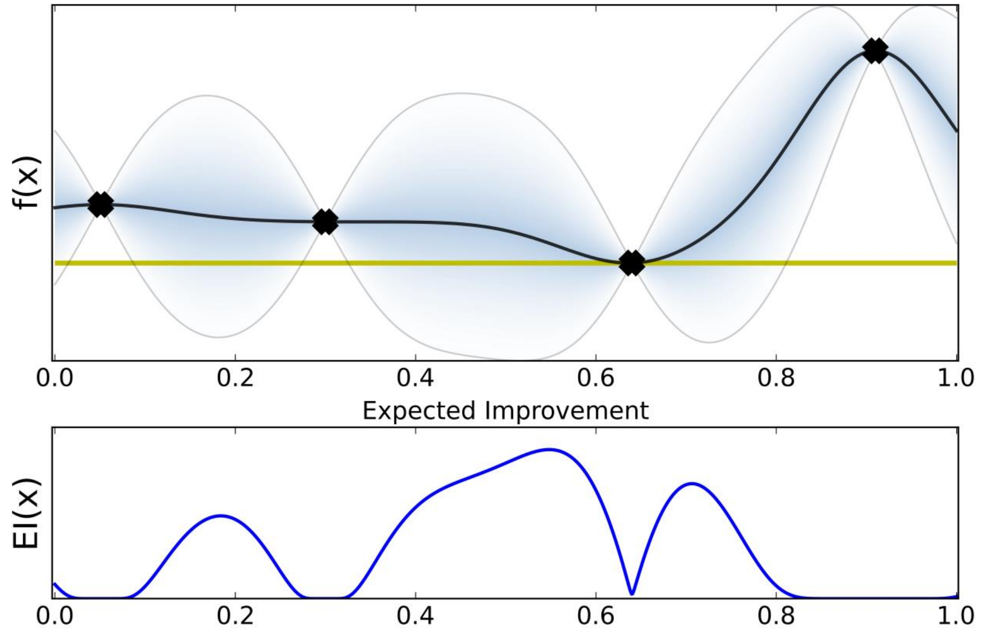
$$EI(x) = E[\max\{0, f(x_{best}) - f(x)\}]$$

- This corresponds to the following utility function

$$EI(x) = \sigma(x)\left(\gamma(x)\Phi\big(\gamma(x)\big) + N(\gamma(x)|0,1)\right)$$

where $\gamma(x) = \frac{f(x_{best} - \mu(x))}{\sigma(x)}$

- Φ is the cumulative distribution of the standard Gaussian distribution.

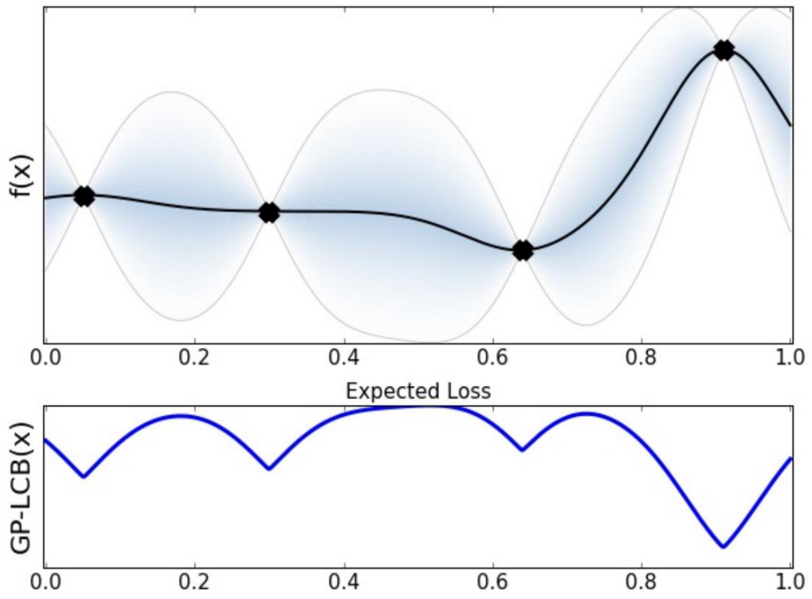


Acquisition Functions

2) Lower Confidence Bound

$$UCB(x) = \mu(x) - \kappa \cdot \sigma(x)$$

- In the above κ is a tunable parameter
- if κ is increased, will force the acquisition function to weigh areas of high variance more than areas of low variance.



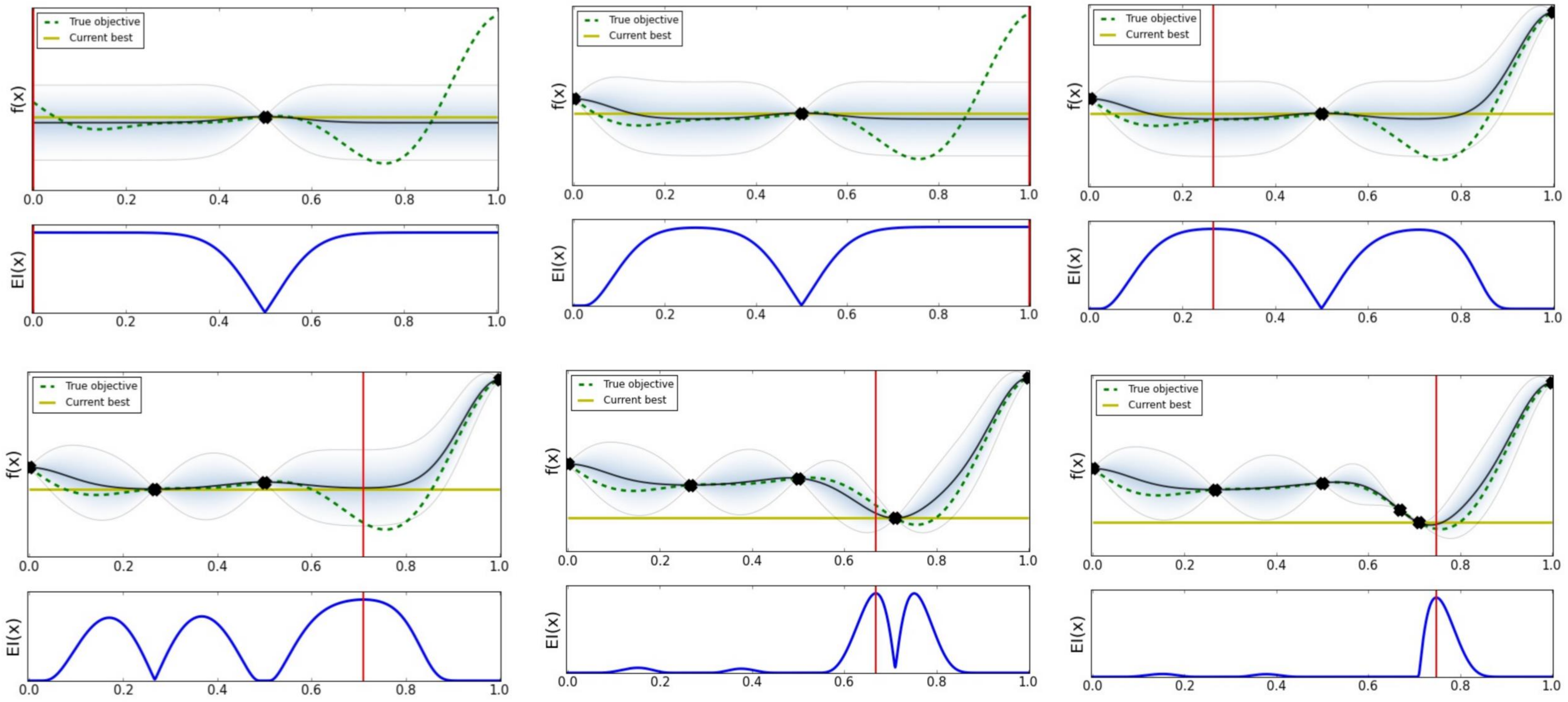
- Other acquisition functions include Probability of Improvement, less used in practice

Procedure of Bayesian Optimization

- By evaluating hyperparameters that appear more promising from past results, Bayesian methods can find better model settings than random search in fewer iterations

- Procedure Bayesian Optimization
  1) Build a surrogate probability model of the objective function
  2) Find the hyperparameters that perform best on the surrogate
  3) Apply these hyperparameters to the true objective function
  4) Update the surrogate model incorporating the new results
  5) Repeat steps 2–4 until max iterations or time is reached

Illustration of BO



Which Hyper Optimization Methods?

- Tree-based models such as Random Forest, Gradient Boosting, and XGBoost are often optimized using grid search, random search, and Bayesian optimization.
- These models have a relatively small number of hyperparameters and grid search and random search are often sufficient for tuning them.

- Support Vector Machines (SVMs) can be tuned using grid search, random search, and Bayesian optimization.

- Neural Networks, especially deep learning models, require a large number of hyperparameters and are often optimized using grid search, random search, Bayesian optimization, and evolutionary algorithms.

- K-Nearest Neighbors (KNN) is optimized using grid search, random search, and Bayesian optimization.

- Naive Bayes, Linear and Logistic Regression, and other linear models often have few hyperparameters and are usually optimized using grid search or random search.

Bayesian Optimization in Python - 1

```
from skopt import BayesSearchCV
from sklearn.datasets import load_digits
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split

X, y = load_digits(n_class=10, return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.75, test_size=.25, random_state=0)

opt = BayesSearchCV( SVC(),
    {  'C': (1e-6, 1e+6, 'log-uniform'),
        'gamma': (1e-6, 1e+1, 'log-uniform'),
        'degree': (1, 8),  # integer valued parameter
        'kernel': ['linear', 'poly', 'rbf'],  # categorical parameter
    },
    n_iter=32, cv=3 )

opt.fit(X_train, y_train)

print("val. score: %s" % opt.best_score_)
print("test score: %s" % opt.score(X_test, y_test))
```

Bayesian Optimization in Python - 2

```
import numpy as np
from pyGPGO.covfunc import squaredExponential
from pyGPGO.acquisition import Acquisition
from pyGPGO.surrogates.GaussianProcess import GaussianProcess
from pyGPGO.GPGO import GPGO

def f(x):
    return (np.sin(x))

sexp = squaredExponential()
gp = GaussianProcess(sexp)
acq = Acquisition(mode='ExpectedImprovement')
param = {'x': ('cont', [0, 2 * np.pi])}

np.random.seed(23)
gpgo = GPGO(gp, acq, f, param)
gpgo.run(max_iter=20)
```