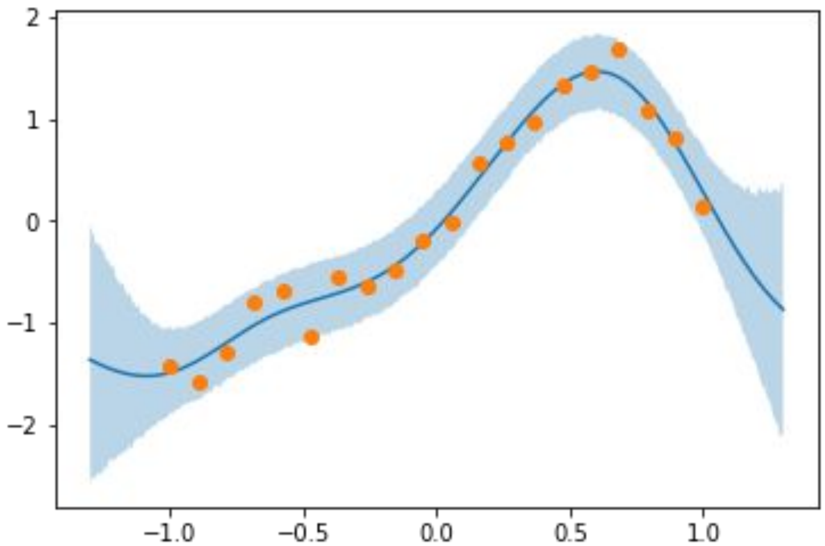
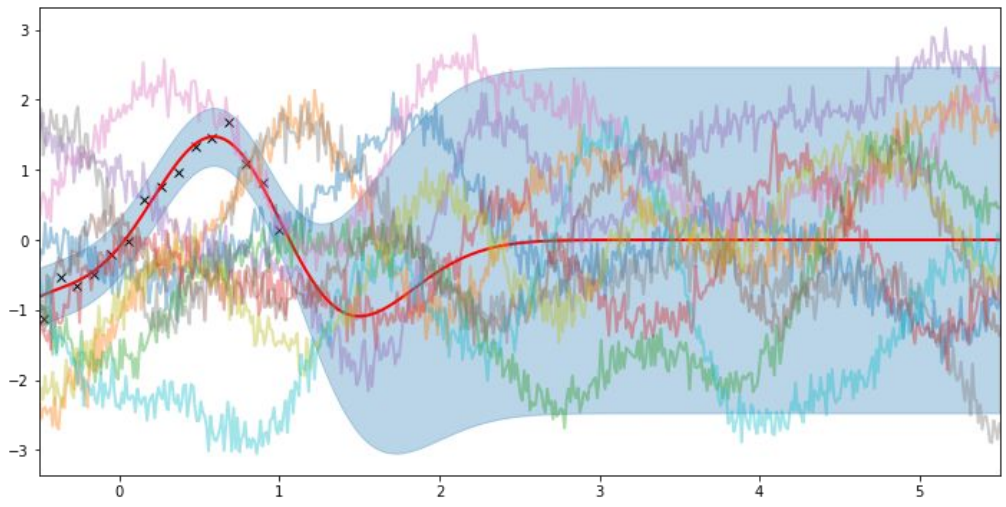
Gaussian processes and Bayesian Optimization

Prof. Dr. Nico Serra - University of Zurich

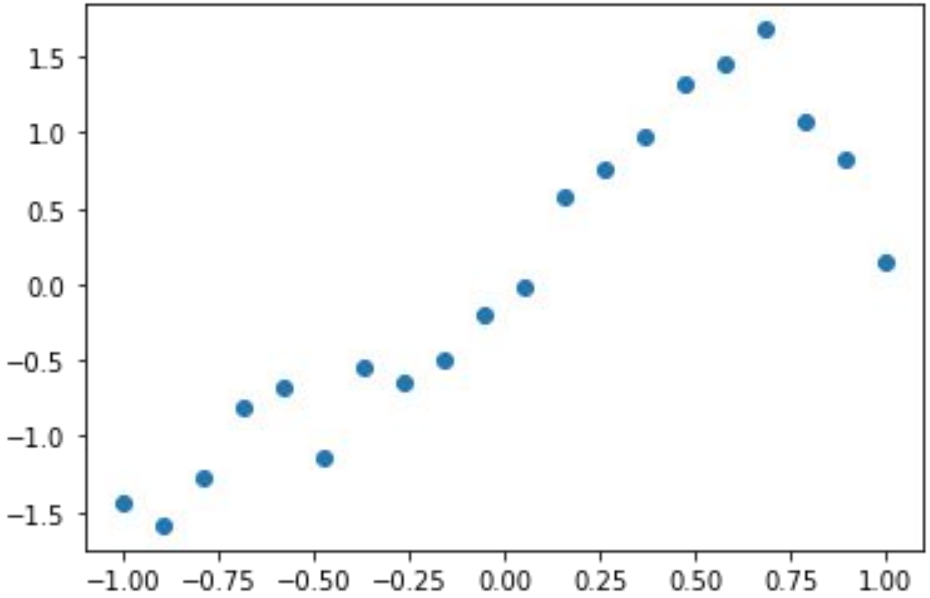
https://github.com/nserra78/misis_bayesian_ML.git





Numpyro implementation of GP

- Let's implement a GP with numpyro



Numpyro implementation of GP

```
import os
```

```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np

import jax
from jax import vmap
import jax.numpy as jnp
import jax.random as random

import numpyro
import numpyro.distributions as dist
```

```
from numpyro.infer import (
    MCMC,
    NUTS,
    init_to_feasible,
    init_to_median,
    init_to_sample,
    init_to_uniform,
    init_to_value,
)
```

Numpyro implementation of GP

```
# squared exponential kernel with diagonal noise term
def kernel(X, Z, var, length, noise, jitter=1.0e-6, include_noise=True):
    deltaXsq = jnp.power((X[:, None] - Z) / length, 2.0)
    k = var * jnp.exp(-0.5 * deltaXsq)
    if include_noise:
        k += (noise + jitter) * jnp.eye(X.shape[0])
    return k
```

```
def model(X, Y):
    # set uninformative log-normal priors on our three kernel hyperparameters
    var = numpyro.sample("kernel_var", dist.LogNormal(0.0, 10.0))
    noise = numpyro.sample("kernel_noise", dist.LogNormal(0.0, 10.0))
    length = numpyro.sample("kernel_length", dist.LogNormal(0.0, 10.0))

    # compute kernel
    k = kernel(X, X, var, length, noise)

    # sample Y according to the standard gaussian process formula
    numpyro.sample(
        "Y",
        dist.MultivariateNormal(loc=jnp.zeros(X.shape[0]), covariance_matrix=k),
        obs=Y,
    )
```

Numpyro implementation of GP

```
# do GP prediction for a given set of hyperparameters. this makes use of the well-known
# formula for gaussian process predictions
def predict(rng_key, X, Y, X_test, var, length, noise):
    # compute kernels between train and test data, etc.
    k_pp = kernel(X_test, X_test, var, length, noise, include_noise=True)
    k_pX = kernel(X_test, X, var, length, noise, include_noise=False)
    k_XX = kernel(X, X, var, length, noise, include_noise=True)
    K_xx_inv = jnp.linalg.inv(k_XX)
    K = k_pp - jnp.matmul(k_pX, jnp.matmul(K_xx_inv, jnp.transpose(k_pX)))
    sigma_noise = jnp.sqrt(jnp.clip(jnp.diag(K), a_min=0.0)) * jax.random.normal(
        rng_key, X_test.shape[:1]
    )
    mean = jnp.matmul(k_pX, jnp.matmul(K_xx_inv, Y))
    # we return both the mean function and a sample from the posterior predictive for the
    # given set of hyperparameters
    return mean, mean + sigma_noise
```

Numpyro implementation of GP

```
# helper function for doing hmc inference
def run_inference(model, rng_key, X, Y, init_strategy="value", num_warmup=100,
                  num_samples=1000, num_chains=2, thinning=1):
    # demonstrate how to use different HMC initialization strategies
    if init_strategy == "value":
        init_strategy = init_to_value(
            values={"kernel_var": 1.0, "kernel_noise": 0.05, "kernel_length": 0.5}
        )
    elif init_strategy == "median":
        init_strategy = init_to_median(num_samples=10)
    elif init_strategy == "feasible":
        init_strategy = init_to_feasible()
    elif init_strategy == "sample":
        init_strategy = init_to_sample()
    elif init_strategy == "uniform":
        init_strategy = init_to_uniform(radius=1)
    kernel = NUTS(model, init_strategy=init_strategy)
    mcmc = MCMC(
        kernel,
        num_warmup,
        num_samples,
        num_chains=num_chains,
        thinning=thinning,
        progress_bar=False if "NUMPYRO_SPHINXBUILD" in os.environ else True,
    )
    mcmc.run(rng_key, X, Y)
    mcmc.print_summary()
    #print("\nMCMC elapsed time:", time.time() - start)
    return mcmc.get_samples()
```

Numpyro implementation of GP

```
rng_key, rng_key_predict = random.split(random.PRNGKey(0))
```

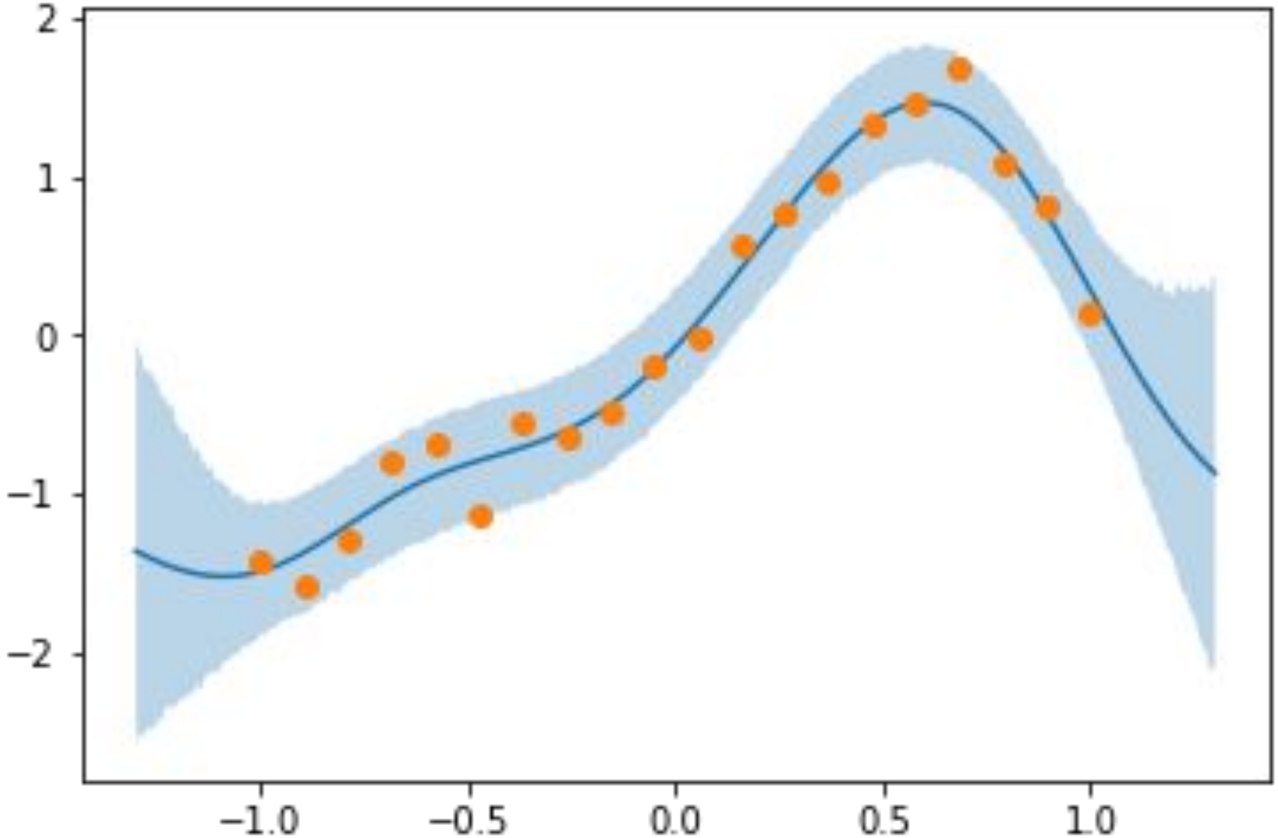
```
sample = run_inference(model, rng_key, X, Y)
```

```
/disk/users/nserra/numpyro/numpyro/infer/mcmc.py:257: UserWarning: There are not enough devic
es to run parallel chains: expected 2 but got 1. Chains will be drawn sequentially. If you ar
e running MCMC in CPU, consider using `numpyro.set_host_device_count(2)` at the beginning of
your program. You can double-check how many devices are available in your system using `jax.l
ocal_device_count()`.
  warnings.warn('There are not enough devices to run parallel chains: expected {} but got
{}.'
sample: 100%|██████████| 1100/1100 [00:05<00:00, 197.08it/s, 7 steps of size 3.04e-01. acc. p
rob=0.92]
sample: 100%|██████████| 1100/1100 [00:01<00:00, 778.20it/s, 3 steps of size 4.15e-01. acc. p
rob=0.95]
```

	mean	std	median	5.0%	95.0%	n_eff	r_hat
kernel_length	0.49	0.14	0.48	0.26	0.70	936.11	1.00
kernel_noise	0.04	0.02	0.03	0.01	0.06	803.78	1.00
kernel_var	3.38	6.79	1.60	0.27	7.03	745.36	1.00

```
Number of divergences: 0
```

Numpyro implementation of GP



Using pyro GP API

In [1]:
```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np

import jax
from jax import vmap
import jax.numpy as jnp
import jax.random as random

import numpyro
import numpyro.distributions as dist
```

In [2]:
```
import torch

import pyro
import pyro.contrib.gp as gp
import pyro.distributions as dist
```

In [3]:
```
from pyro.infer import Predictive
```

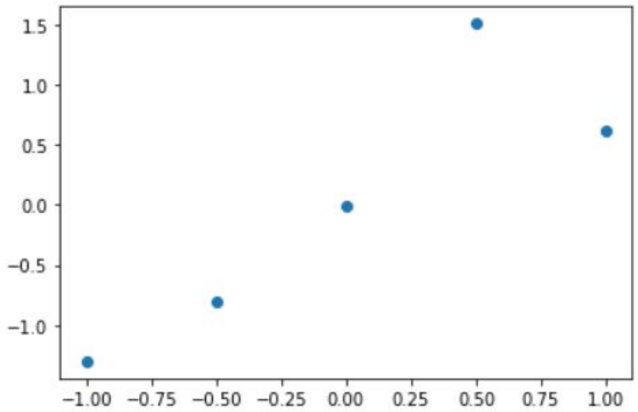
Using pyro GP API

```
In [6]: X, Y, Xtest = get_data(N=5, x_min=-1.5, x_max=1.5)
```

```
In [7]: X = torch.from_numpy(X.astype(np.float32))
        Y = torch.from_numpy(Y.astype(np.float32))
        Xtest = torch.from_numpy(Xtest.astype(np.float32))
```

```
In [8]: plt.plot(X, Y, "o")
```

```
Out[8]: [<matplotlib.lines.Line2D at 0x7ff469a37ca0>]
```

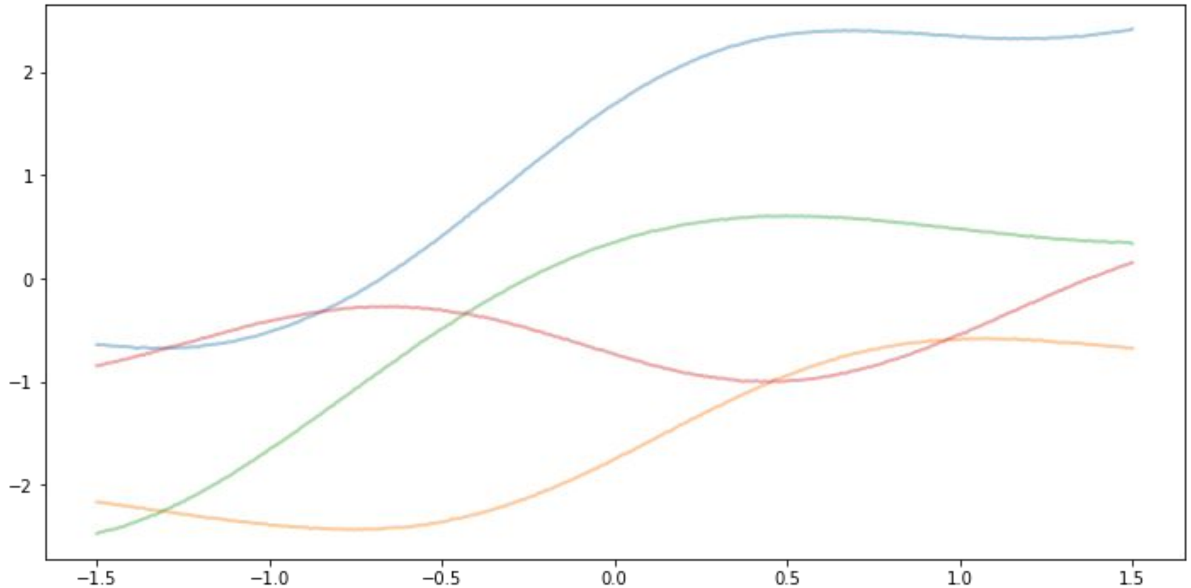


Using pyro GP API

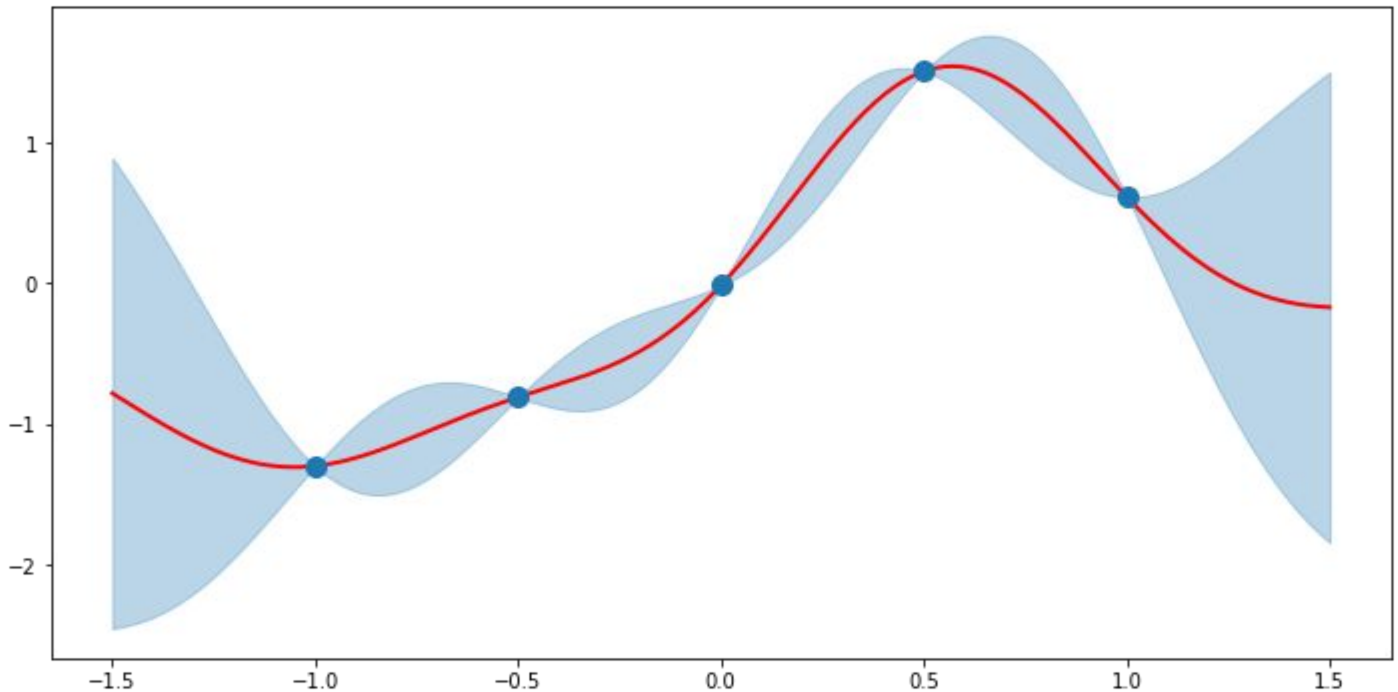
```
kernel = gp.kernels.RBF(input_dim=1, variance=torch.tensor(1.),
                        lengthscale=torch.tensor(1.))
```

```
gpr = gp.models.GPRegression(X, Y, kernel, noise=torch.tensor(1e-5))
#gpr = gp.models.GPRegression(X, Y, kernel, noise=torch.tensor(1e-2))
```

```
plot(model=gpr, kernel=kernel, n_prior_samples=4)
```



Using pyro GP API

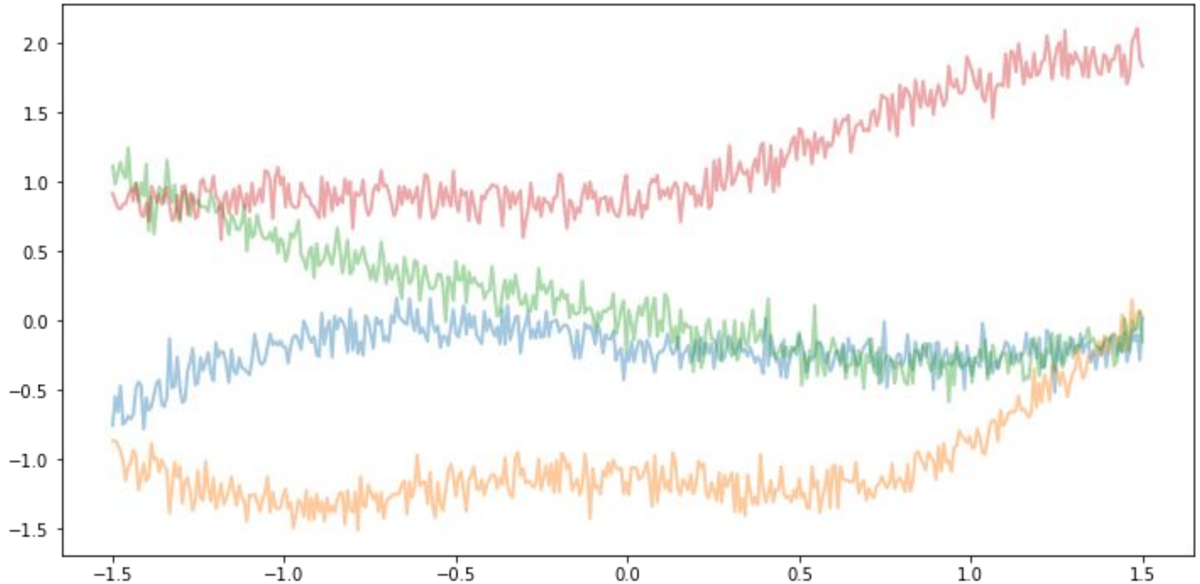


Using pyro GP API

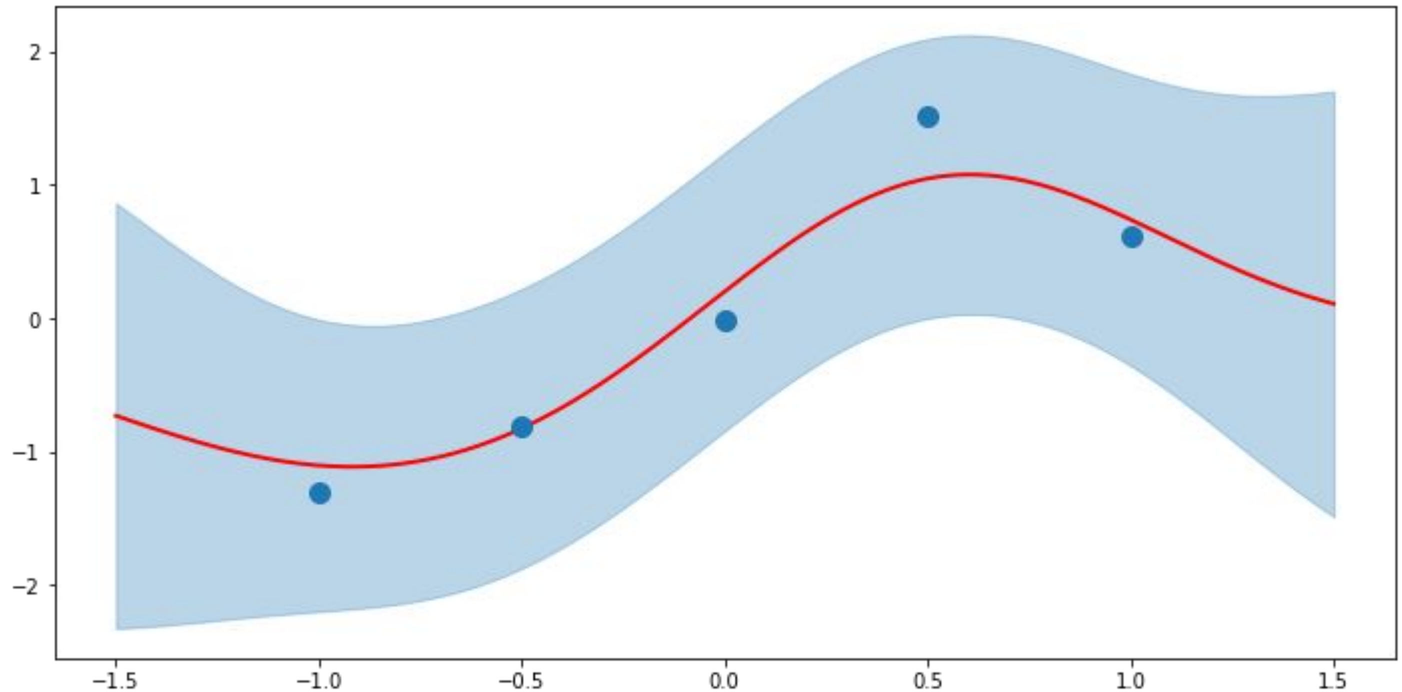
```
kernel = gp.kernels.RBF(input_dim=1, variance=torch.tensor(1.),
                        lengthscale=torch.tensor(1.))
```

```
#gpr = gp.models.GPRegression(X, Y, kernel, noise=torch.tensor(1e-5))
gpr = gp.models.GPRegression(X, Y, kernel, noise=torch.tensor(1e-2))
```

```
plot(model=gpr, kernel=kernel, n_prior_samples=4)
```



Using pyro GP API

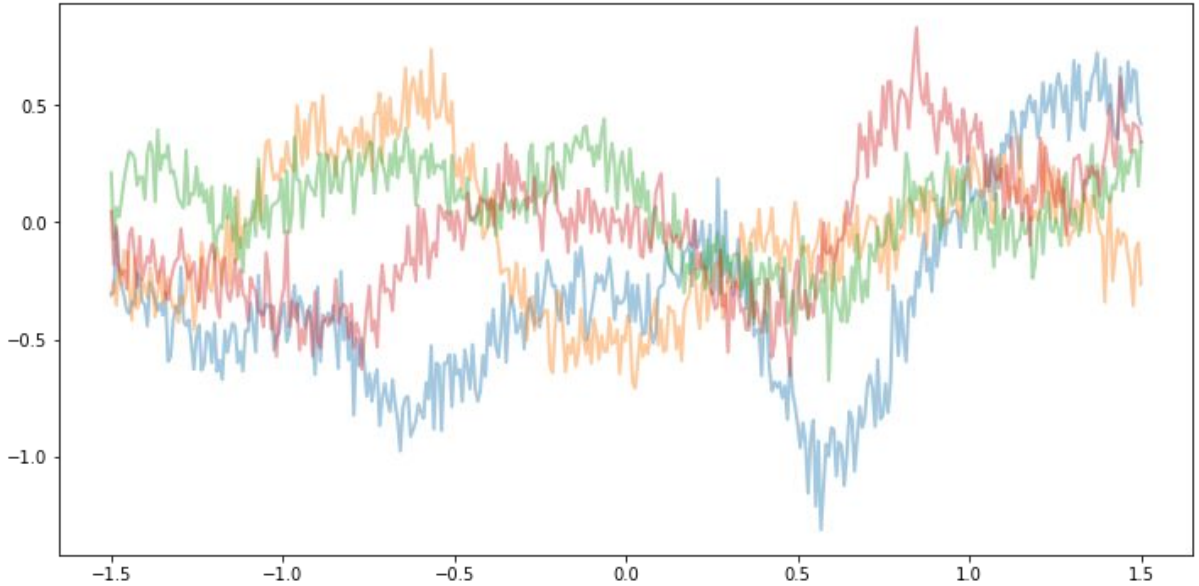


Using pyro GP API

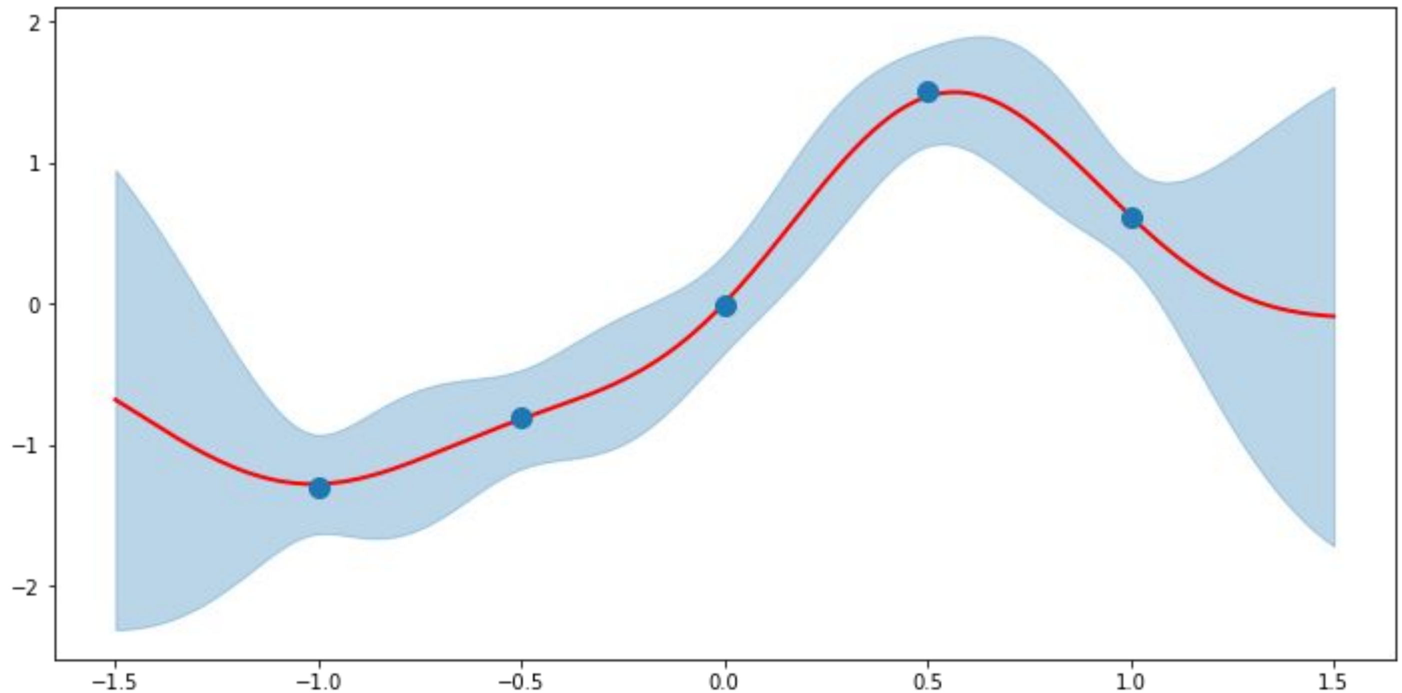
```
kernel = gp.kernels.RBF(input_dim=1, variance=torch.tensor(.1),
                        lengthscale=torch.tensor(0.2))
```

```
#gpr = gp.models.GPRegression(X, Y, kernel, noise=torch.tensor(1e-5))
gpr = gp.models.GPRegression(X, Y, kernel, noise=torch.tensor(1e-2))
```

```
plot(model=gpr, kernel=kernel, n_prior_samples=4)
```

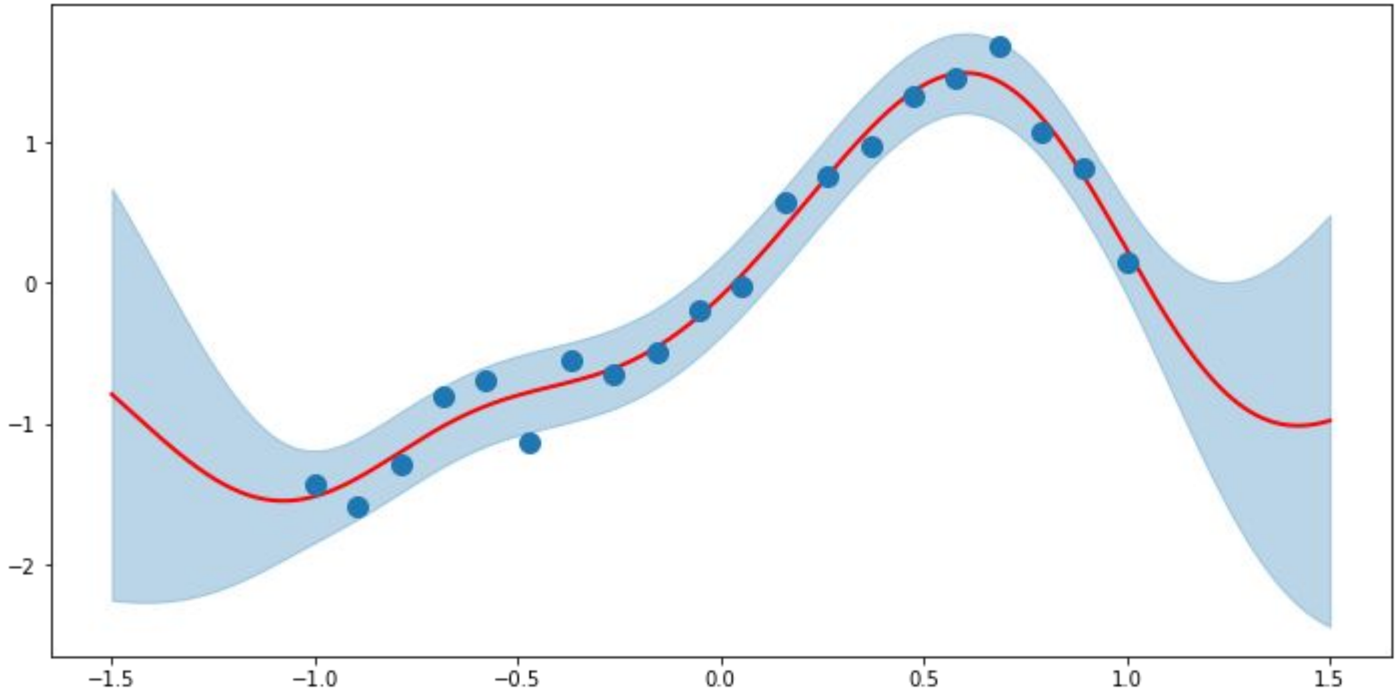


Using pyro GP API



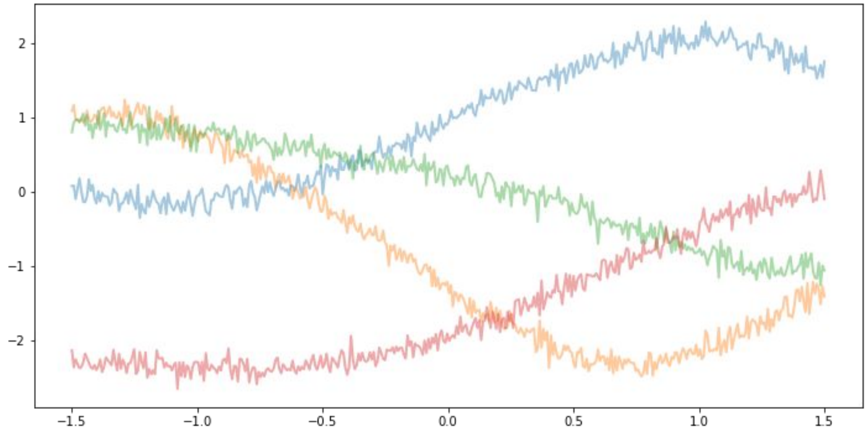
Using pyro GP API

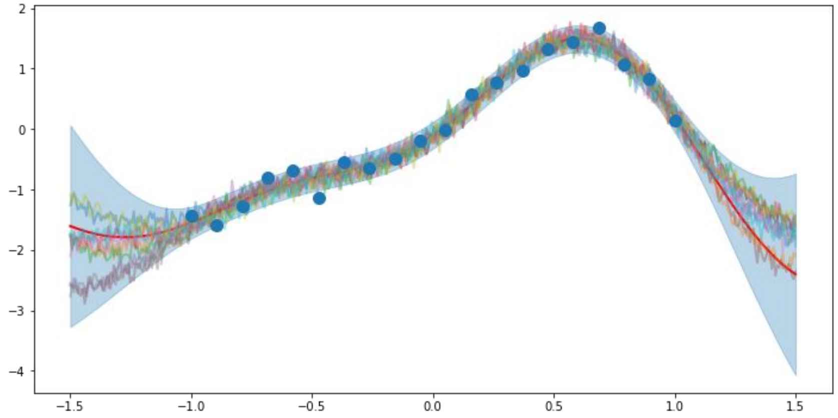
- When you decide on the kernel it depends what you want to achieve, if you are interested in interpolation, or extrapolation or you have some prior information about the function



Determining Length and Variance

```
# note that our priors have support on the positive reals
gpr.kernel.lengthscale = pyro.nn.PyroSample(dist.LogNormal(0.0, 1.0))
gpr.kernel.variance = pyro.nn.PyroSample(dist.LogNormal(0.0, 1.0))
```



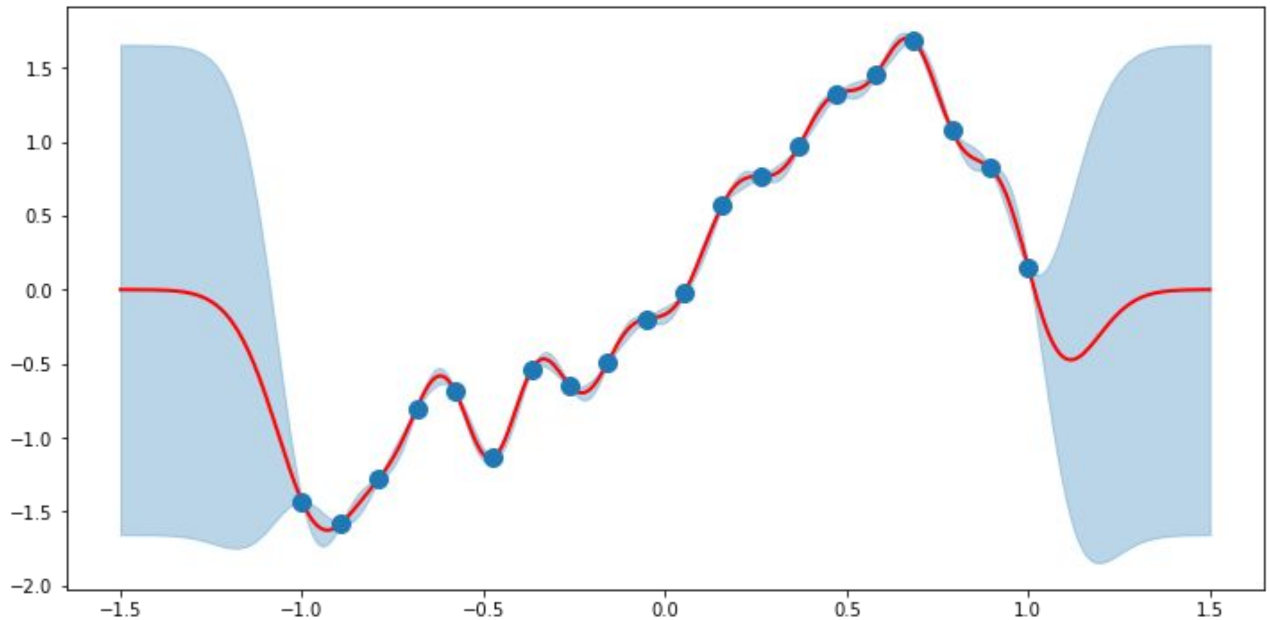


MISIS
National University of Science and Technology

University of Zurich[UZH]

Using pyro GP API

```
# note that our priors have support on the positive reals
gpr.kernel.lengthscale = pyro.nn.PyroSample(dist.LogNormal(0.0, 1.0))
gpr.kernel.variance = pyro.nn.PyroSample(dist.LogNormal(0.0, 1.0))
gpr.kernel.noise = pyro.nn.PyroSample(dist.LogNormal(0.0, 1e-1))
```



- Be careful that if you optimize all parameters you will describe the points very well, but you might have zero prediction power… choosing the kernel can be treated as a prior