synthetic

November 5, 2024

1 Imports

[1]:
```
# Notebook reload options
%load_ext autoreload
%autoreload 2
```

[2]:
```
# Global Imports
import numpy as np
import torch
import itertools
import pandas as pd
from torch.utils.data import DataLoader
import sys
from time import process_time as timer
import matplotlib.pyplot as plt
from tqdm import tqdm
import copy
from scipy.cluster.vq import kmeans2
import hamiltorch

# Local Imports
sys.path.append("..")
sys.path.append(".")
from bayesipy.utils.datasets import Synthetic_Dataset

from bayesipy.fmgp import FMGP
from bayesipy.laplace import Laplace, ELLA, VaLLA
from bayesipy.mfvi import MFVI
```

```
c:\Users\Ludvins\Documents\VariationalUncertaintyEstimation\.venv\Lib\site-
packages\tqdm\auto.py:21: TqdmWarning: IProgress not found. Please update
jupyter and ipywidgets. See
https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

2 Experimental settings

Set seed for reproductibility.

[3]:
```
from bayesipy.utils import assert_reproducibility

assert_reproducibility(1234)
```

Load Dataset and desired split.

[4]:
```
dataset = Synthetic_Dataset()
train_dataset, test_dataset = dataset.get_splits()
```

```
Number of samples:  400
Input dimension:  (1,)
Label dimension:  1
```

Create Data loaders for training and test partitions.

[5]:
```
batch_size = 100
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size)
```

3 Pretrained MAP solution

[6]:
```
f = torch.nn.Sequential(
    torch.nn.Linear(1, 50),
    torch.nn.Tanh(),
    torch.nn.Linear(50, 50),
    torch.nn.Tanh(),
    torch.nn.Linear(50, 1),
)

f = f.to(torch.float64)

# Define optimizer and compile model
opt = torch.optim.Adam(f.parameters(), lr=0.001)
criterion = torch.nn.MSELoss()

# Set the number of training samples to generate
# Train the model
start = timer()

iterator = iter(train_loader)
for _ in range(12000):
    try:
        X, y = next(iterator)
    except StopIteration:
```

```
        iterator = iter(train_loader)
        X, y = next(iterator)

    opt.zero_grad()
    y_pred = f(X)
    loss = criterion(y_pred, y)
    loss.backward()
    opt.step()


end = timer()
```

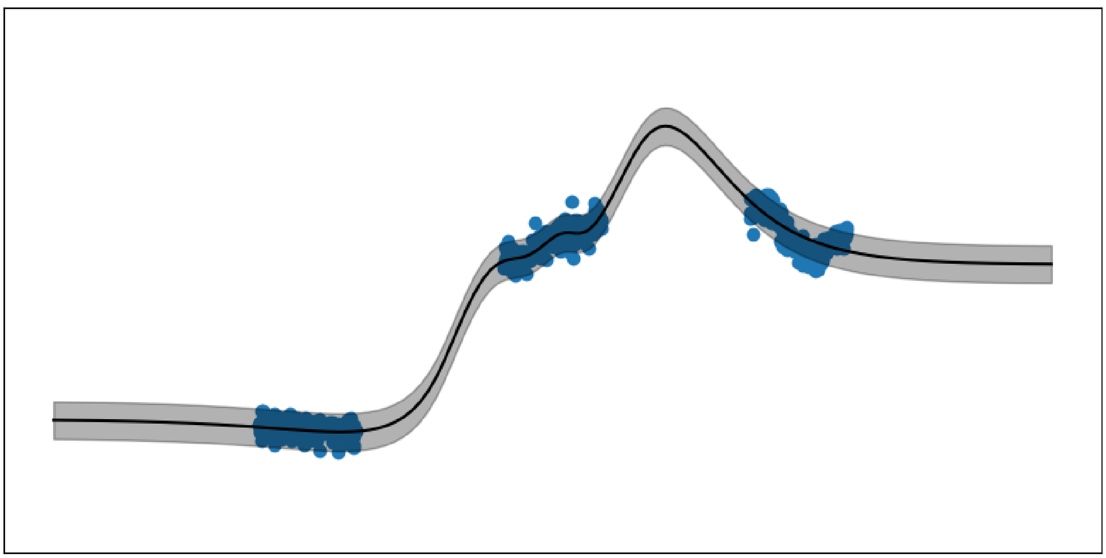
[7]:
```
params_init = copy.deepcopy(hamiltorch.util.flatten(f).detach())
```

[8]:
```
noises = np.linspace(0.001, 0.2, 100)

best_noise = 0
best_ll = -np.inf
mean = f(torch.tensor(train_dataset.inputs)).detach().cpu().numpy().flatten()
for noise in noises:
    # Compute Gaussian density of test_dataset
    ll = (
        -0.5 * np.log(2 * np.pi * noise)
        - 0.5 * np.square(train_dataset.targets.flatten() - mean) / noise
    ).sum()

    if ll > best_ll:
        best_ll = ll
        best_noise = noise

print(best_noise, best_ll)
```

```
0.00301010101010101 545.231212359985
```

[9]:
```
# Change figure size
plt.figure(figsize=(10, 5))
plt.scatter(train_dataset.inputs, train_dataset.targets, label="Training␣
 ↪points")
sort = np.argsort(test_dataset.inputs.flatten())

mean = f(torch.tensor(test_dataset.inputs)).detach().cpu().numpy().
 ↪flatten()[sort]

plt.plot(
    test_dataset.inputs.flatten()[sort],
    mean,
    label="Predictions",
```

```
    color="black",
)
plt.fill_between(
    test_dataset.inputs.flatten()[sort],
    mean - 2 * np.sqrt(best_noise),
    mean + 2 * np.sqrt(best_noise),
    alpha=0.3,
    color="black",
)
plt.ylim(-1.2, 2)

plt.xticks([])
plt.yticks([])
plt.savefig("synthetic_regression_map.pdf", format="pdf", bbox_inches="tight")
plt.show()
```



4 Gaussian Process

[10]:
```
# Define the RBF kernel
class RBFKernel(torch.nn.Module):
    def __init__(self, length_scale=1.0, variance=1.0):
        super(RBFKernel, self).__init__()
        # Initialize kernel hyperparameters
        self.length_scale = torch.nn.Parameter(
            torch.tensor(length_scale, dtype=torch.float64)
        )
```

```
        self.variance = torch.nn.Parameter(torch.tensor(variance, dtype=torch.
↪float64))

    def forward(self, X1, X2):
        # Compute squared Euclidean distance
        sqdist = ((X1.unsqueeze(1) - X2.unsqueeze(0)) ** 2).sum(2)
        # RBF kernel
        return self.variance * torch.exp(-0.5 * sqdist / self.length_scale**2)


# Define the Gaussian Process model
class GaussianProcess(torch.nn.Module):
    def __init__(self, kernel, noise=1e-1):
        super(GaussianProcess, self).__init__()
        self.kernel = kernel
        self.noise = torch.nn.Parameter(torch.tensor(noise, dtype=torch.
↪float64))

    def forward(self, X_train, y_train):
        K = self.kernel(X_train, X_train) + self.noise**2 * torch.eye(
            X_train.size(0), dtype=torch.float64
        )
        L = torch.linalg.cholesky(K)  # Cholesky decomposition
        # Solve for alpha
        alpha = torch.cholesky_solve(y_train, L)
        return K, L, alpha

    def marginal_likelihood(self, X_train, y_train):
        K, L, alpha = self.forward(X_train, y_train)
        # Compute the log marginal likelihood
        data_fit = -0.5 * y_train.T @ alpha
        complexity_penalty = -torch.sum(torch.log(torch.diagonal(L)))
        normalization = -0.5 * X_train.size(0) * np.log(2 * torch.pi)
        return (data_fit + complexity_penalty + normalization).squeeze()

    def predict(self, X_train, y_train, X_test):
        # Compute the kernel matrices needed for prediction
        K_train = self.kernel(X_train, X_train) + self.noise**2 * torch.eye(
            X_train.size(0), dtype=torch.float64
        )
        K_train_test = self.kernel(X_train, X_test)
        K_test_test = self.kernel(X_test, X_test)

        # Cholesky decomposition of the training kernel matrix
        L = torch.linalg.cholesky(K_train)

        # Compute alpha for the training points
```

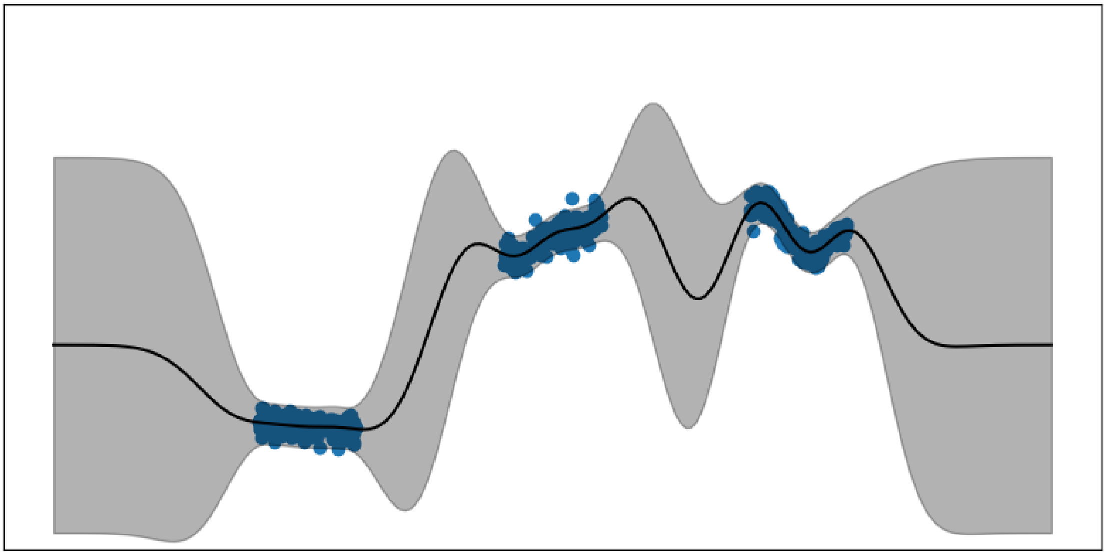
```python
        alpha = torch.cholesky_solve(y_train, L)

        # Predictive mean
        predictive_mean = K_train_test.T @ alpha

        # Compute the variance at the test points
        v = torch.cholesky_solve(K_train_test, L)
        predictive_variance = K_test_test - K_train_test.T @ v

        return predictive_mean.squeeze().detach(), torch.diag(
            predictive_variance
        ).sqrt().detach() + self.noise.item()


# Instantiate the GP model
kernel = RBFKernel(length_scale=1.0, variance=1.0)
gp = GaussianProcess(kernel=kernel, noise=1e-1)

# Define optimizer
from torch import optim

optimizer = optim.Adam(gp.parameters(), lr=0.01)

# Training loop to optimize the log marginal likelihood
num_epochs = 100
for epoch in range(num_epochs):
    optimizer.zero_grad()
    # Compute the negative log marginal likelihood
    nll = -gp.marginal_likelihood(
        torch.tensor(train_dataset.inputs), torch.tensor(train_dataset.targets)
    )
    # Backpropagate
    nll.backward()
    optimizer.step()

    if epoch % 10 == 0:
        print(f"Epoch {epoch}: Negative Log Marginal Likelihood = {nll.item()}")

# Print optimized hyperparameters
print(f"Optimized length_scale: {kernel.length_scale.item()}")
print(f"Optimized variance: {kernel.variance.item()}")
print(f"Optimized noise: {gp.noise.item()}")
```

```
Epoch 0: Negative Log Marginal Likelihood = -456.2971033954215
Epoch 10: Negative Log Marginal Likelihood = -574.74391134798
Epoch 20: Negative Log Marginal Likelihood = -575.7743086243393
Epoch 30: Negative Log Marginal Likelihood = -577.2815578425971
```

```
Epoch 40: Negative Log Marginal Likelihood = -577.3822041864476
Epoch 50: Negative Log Marginal Likelihood = -577.9768032203744
Epoch 60: Negative Log Marginal Likelihood = -578.758590406992
Epoch 70: Negative Log Marginal Likelihood = -579.1378203596666
Epoch 80: Negative Log Marginal Likelihood = -579.4150776165498
Epoch 90: Negative Log Marginal Likelihood = -579.4878538745731
Optimized length_scale: 1.080517380521987
Optimized variance: 0.25044569441512354
Optimized noise: 0.050881442835711142
```

[11]:

```
plt.figure(figsize=(10, 5))
plt.scatter(train_dataset.inputs, train_dataset.targets, label="Training␣
↪points")
sort = np.argsort(test_dataset.inputs.flatten())

mean, std = gp.predict(
    torch.tensor(train_dataset.inputs),
    torch.tensor(train_dataset.targets),
    torch.tensor(test_dataset.inputs),
)


plt.plot(
    test_dataset.inputs.flatten()[sort],
    mean,
    label="Predictions",
    color="black",
)
plt.fill_between(
    test_dataset.inputs.flatten()[sort],
    mean - 2 * std,
    mean + 2 * std,
    alpha=0.3,
    color="black",
)

plt.ylim(-1.2, 2)
# Remove ticks
plt.xticks([])
plt.yticks([])
plt.savefig("synthetic_regression_GP.pdf", format="pdf", bbox_inches="tight")
plt.show()
```



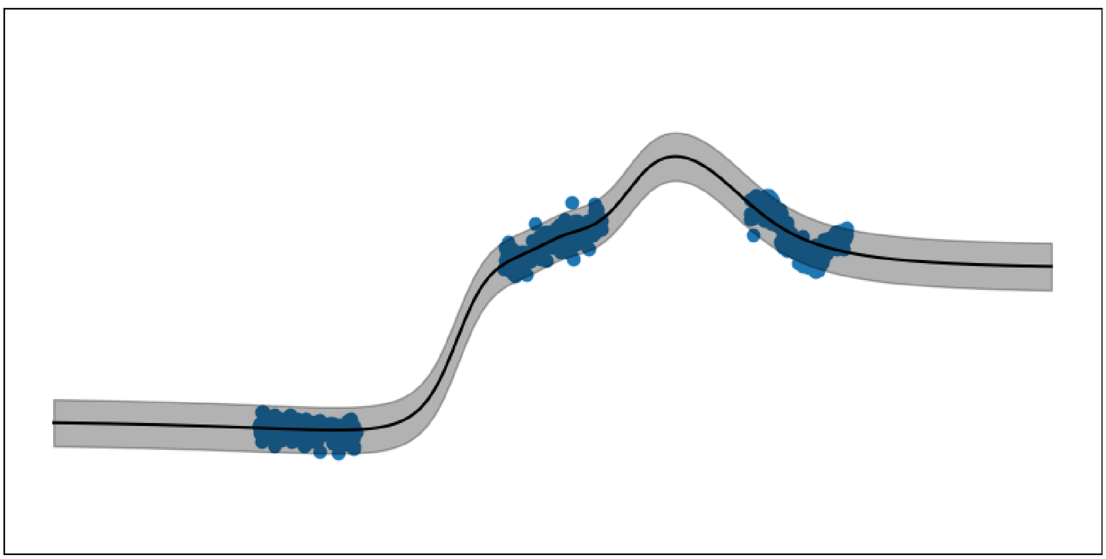
5 Mean Field VI

[12]:
```
mfvi = MFVI(
    copy.deepcopy(f),
    n_samples=200,
    likelihood="regression",
    noise_std=4,
    prior_precision=1.0,
    y_mean=0.0,
    y_std=1.0,
    seed=0,
)
losses = mfvi.fit(train_loader, 50000, verbose=True)


mfvi_preds = mfvi.sample(torch.tensor(test_dataset.inputs)).detach().cpu().
  ↪numpy()
```

```
Training : 100%|          | 50000/50000 [02:22<00:00, 351.95 iteration/s]
```

[13]:
```
plt.figure(figsize=(10, 5))
plt.scatter(train_dataset.inputs, train_dataset.targets, label="Training␣
  ↪points")
sort = np.argsort(test_dataset.inputs.flatten())

mean = mfvi_preds.mean(axis=0).flatten()
variance = mfvi.log_noise.exp().item() ** 2 + mfvi_preds.var(axis=0)
```

```
std = np.sqrt(variance).flatten()

plt.plot(
    test_dataset.inputs.flatten()[sort],
    mean,
    label="Predictions",
    color="black",
)
plt.fill_between(
    test_dataset.inputs.flatten()[sort],
    mean - 2 * std,
    mean + 2 * std,
    alpha=0.3,
    color="black",
)

plt.ylim(-1.2, 2)
# Remove ticks
plt.xticks([])
plt.yticks([])
plt.savefig("synthetic_regression_MFVI.pdf", format="pdf", bbox_inches="tight")
plt.show()
```



[14]:
```
f_new = torch.nn.Sequential(
    torch.nn.Linear(1, 50),
    torch.nn.Tanh(),
    torch.nn.Linear(50, 50),
    torch.nn.Tanh(),
```

```
    torch.nn.Linear(50, 1),
).to(torch.float64)
```

[15]:
```
from bayesipy.utils import gaussian_logdensity
```

[16]:
```
# Definir la función de probabilidad (negative log likelihood)
from hamiltorch import util

fmodel = util.make_functional(f_new)


def model_log_prob(params):
    # Separar los parámetros de la red, el ruido y del prior
    net_params = params[:-2]
    std_ruido = torch.exp(params[-2])
    std_prior = torch.exp(params[-1])

    # Actualizar los pesos de la red con los parámetros actuales
    params_unflattened = util.unflatten(f_new, net_params)

    # Calcular las predicciones
    y_pred = fmodel(
        torch.tensor(train_dataset.inputs), params=params_unflattened
    ).flatten()

    # Likelihood: Ruido Gaussiano con varianza v_ruido
    likelihood = gaussian_logdensity(
        y_pred, std_ruido**2, torch.tensor(train_dataset.targets).flatten()
    ).sum()

    # Prior en los pesos: N(0, v_prior)
    log_prior_w = gaussian_logdensity(
        net_params, std_prior**2, torch.zeros_like(net_params)
    ).sum()

    # Priors no normalizados: 1 / v_ruido y 1 / v_prior
    log_prior_v_ruido = -torch.log(std_ruido)
    log_prior_v_prior = -torch.log(std_prior)

    # Log conjunta
    return likelihood + log_prior_w + log_prior_v_prior + log_prior_v_ruido
```

[17]:
```
# Agregar parámetros de ruido y precisión al vector de parámetros iniciales
v_ruido_init = torch.tensor(
    [np.log(0.06)], dtype=torch.float64
)  # Varianza inicial del ruido
```

```
v_prior_init = torch.tensor([0], dtype=torch.float64)  # Varianza inicial del␣
↪prior

# Concatenar parámetros iniciales
params_init_a = torch.cat([params_init, v_ruido_init, v_prior_init])

# Set the Inverse of the Mass matrix
inv_mass = torch.ones(params_init_a.shape)


hamiltorch.set_random_seed(0)
# Realizar muestreo con HMC
params_hmc = hamiltorch.sample(
    log_prob_func=model_log_prob,
    params_init=params_init_a,
    num_samples=1000,
    burn=-1,
    inv_mass=inv_mass,
    step_size=0.0005,
    num_steps_per_sample=1000,
    sampler=hamiltorch.Sampler.HMC,
)
```

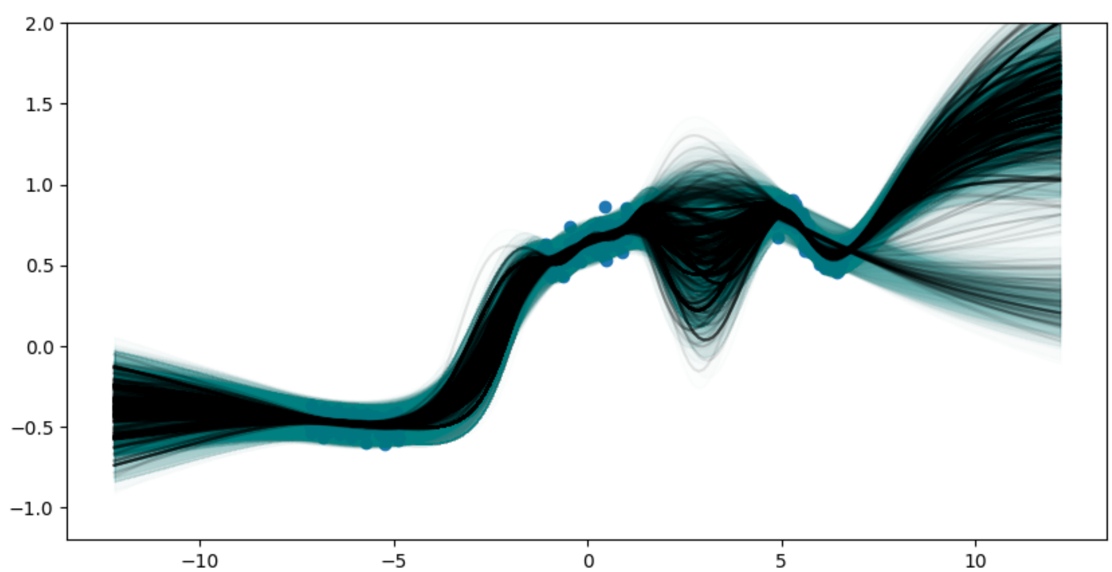
```
Sampling (Sampler.HMC; Integrator.IMPLICIT)
Time spent  | Time remain.| Progress             | Samples   | Samples/sec
0d:00:05:47 | 0d:00:00:00 | #################### | 1000/1000 | 2.88
Acceptance Rate 0.29
```

[28]:
```
y_preds = []
noise_stds = []
prior_precs = []

for params in params_hmc:
    net_params = params[:-2]
    std_ruido = torch.exp(params[-2])

    # Guardar valores de noise_std y prior_prec
    noise_stds.append(std_ruido.item())

    params_unflattened = util.unflatten(f_new, net_params)
    hamiltorch.util.unflatten(f_new, net_params)
    y_pred = fmodel(
        torch.tensor(test_dataset.inputs), params=params_unflattened
    ).flatten()
    y_preds.append(y_pred.detach().numpy())

noise_stds = np.array(noise_stds).squeeze()
```

```
y_preds = np.array(y_preds).squeeze()
mean_preds = y_preds.mean(axis=0)
```

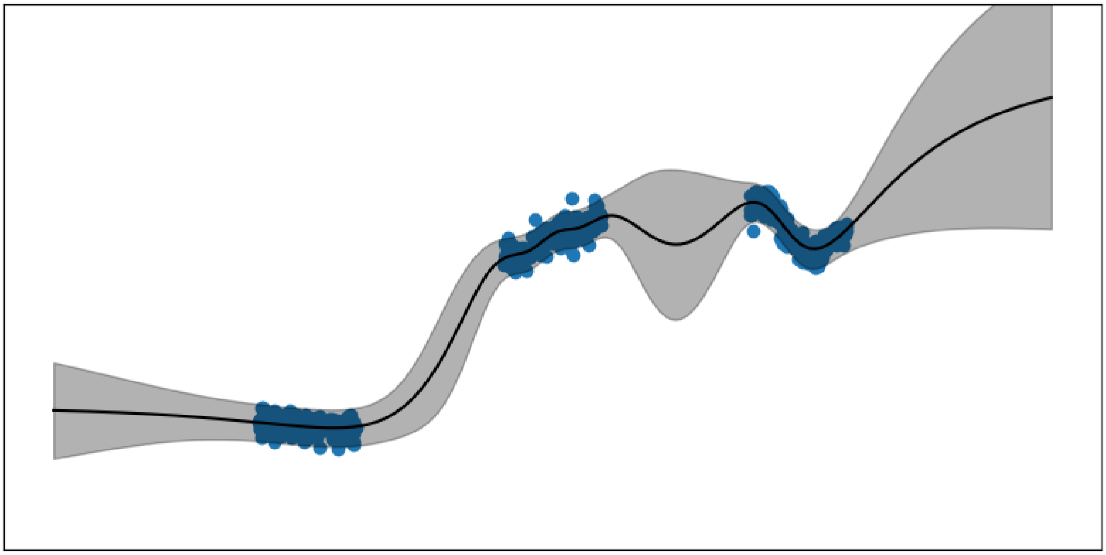
[29]:
```
plt.figure(figsize=(10, 5))
plt.scatter(train_dataset.inputs, train_dataset.targets, label="Training␣
 ↪points")


sort = np.argsort(test_dataset.inputs.flatten())

for mean, std in zip(y_preds, noise_stds):
    plt.plot(
        test_dataset.inputs.flatten()[sort],
        mean[sort],
        color="black",
        alpha=0.1,
    )
    plt.fill_between(
        test_dataset.inputs.flatten()[sort],
        mean[sort] - 2 * std,
        mean[sort] + 2 * std,
        alpha=0.01,
        color="teal",
    )

plt.ylim(-1.2, 2)
```

[29]: (-1.2, 2.0)



[30]:
```
mean = y_preds.mean(axis=0)[np.newaxis, :]

within_component_variance = noise_stds**2
between_component_variance = np.sum((y_preds - mean) ** 2, axis=0) / (y_preds.
 ↪shape[0])
var_mixture = between_component_variance + within_component_variance.mean()
std = np.sqrt(var_mixture).flatten()
# std = y_preds.std(axis=0)
```

[21]:
```
plt.figure(figsize=(10, 5))
plt.scatter(train_dataset.inputs, train_dataset.targets, label="Training␣
 ↪points")


sort = np.argsort(test_dataset.inputs.flatten())


plt.plot(
    test_dataset.inputs.flatten()[sort],
    mean.flatten(),
    label="Predictions",
    color="black",
)
plt.fill_between(
    test_dataset.inputs.flatten()[sort],
    mean.flatten() - 2 * std,
    mean.flatten() + 2 * std,
    alpha=0.3,
    color="black",
)

plt.ylim(-1.2, 2)
# Remove ticks
plt.xticks([])
plt.yticks([])
plt.savefig("synthetic_regression_HMC.pdf", format="pdf", bbox_inches="tight")
plt.show()
```

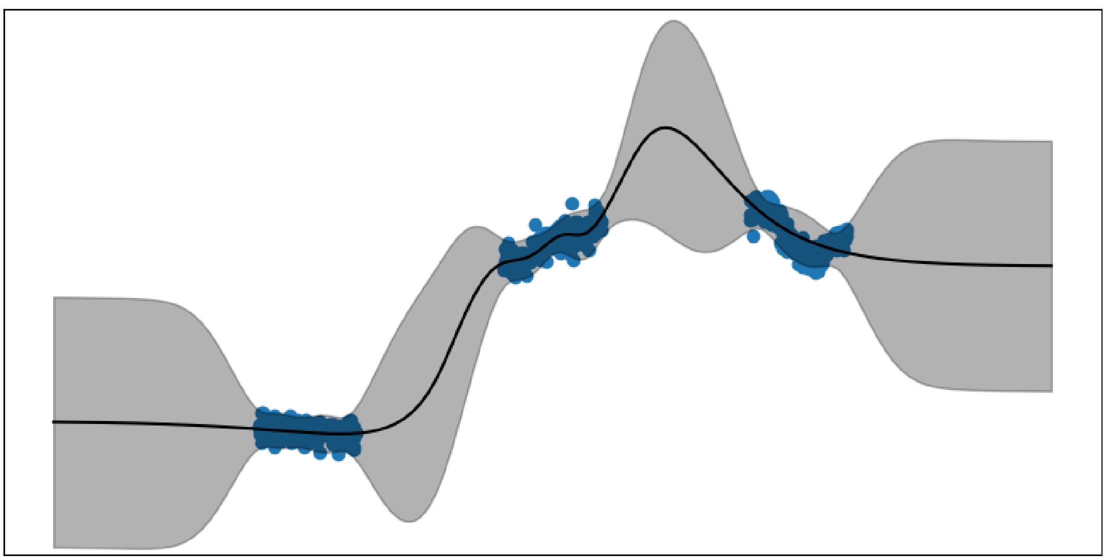


[22]:
```
ue = FMGP(
    model=copy.deepcopy(f),
    likelihood="regression",
    kernel="RBF",
    inducing_locations="kmeans",
    num_inducing=10,
    noise_variance=np.exp(-5),
    subrogate_regularizer=True,
    y_mean=0,
    y_std=1,
)

loss = ue.fit(iterations=70000, lr=0.001, train_loader=train_loader,␣
 ↪verbose=True)
```

```
Initializing inducing locations… done
Creating Kernel Function… done

Training : 100%|      | 70000/70000 [05:22<00:00, 216.93 iteration/s,
loss=-1126.35, lr=0.001]
```

[23]:
```
plt.figure(figsize=(10, 5))
plt.scatter(train_dataset.inputs, train_dataset.targets, label="Training␣
 ↪points")


sort = np.argsort(test_dataset.inputs.flatten())

f_mean, f_var = ue.predict(torch.tensor(test_dataset.inputs))
```

```python
plt.plot(
    test_dataset.inputs.flatten()[sort],
    f_mean.detach().cpu().numpy().flatten()[sort],
    label="Predictions",
    color="black",
)
plt.fill_between(
    test_dataset.inputs.flatten()[sort],
    f_mean.detach().cpu().numpy().flatten()[sort]
    - 2 * np.sqrt(f_var.detach().cpu().numpy().flatten()[sort]),
    f_mean.detach().cpu().numpy().flatten()[sort]
    + 2 * np.sqrt(f_var.detach().cpu().numpy().flatten()[sort]),
    alpha=0.3,
    color="black",
)
plt.ylim(-1.2, 2)
# Remove ticks
plt.xticks([])
plt.yticks([])
plt.savefig("synthetic_regression_fmgp.pdf", format="pdf", bbox_inches="tight")
plt.show()
```



[38]:

```python
lla = Laplace(
    model=copy.deepcopy(f),
    likelihood="regression",
    subset_of_weights="all",
    hessian_structure="full",
```

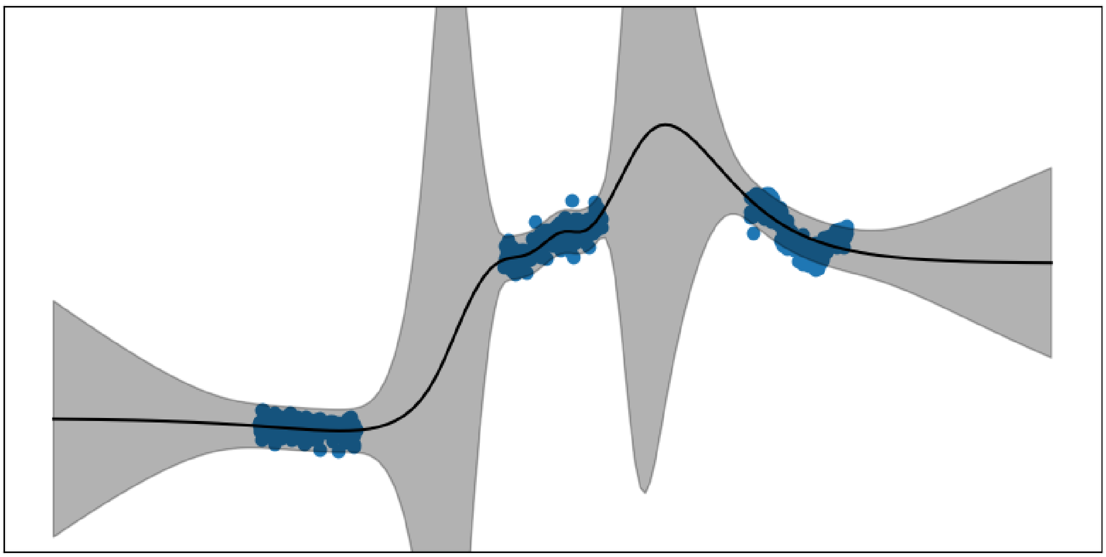
```
)

# Train the model
lla.fit(train_loader=train_loader)


log_sigma = torch.zeros(1, requires_grad=True)
log_prior = torch.zeros(1, requires_grad=True)

hyper_optimizer = torch.optim.Adam([log_prior, log_sigma], lr=1e-1)

for i in range(100):
    hyper_optimizer.zero_grad()
    neg_marglik = -lla.log_marginal_likelihood(log_prior.exp(), log_sigma.exp())
    neg_marglik.backward()
    hyper_optimizer.step()

prior_precision = log_prior.exp().item()
sigma_noise = log_sigma.exp().item()
print(prior_precision, sigma_noise)
```

```
0.1989615559577942 0.06208682060241699
```

[40]:
```
lla._compute_scale()
lla._posterior_scale = lla._posterior_scale.to(torch.float64)
```

[41]:
```
plt.figure(figsize=(10, 5))
plt.scatter(train_dataset.inputs, train_dataset.targets, label="Training␣
↪points")


sort = np.argsort(test_dataset.inputs.flatten())

f_mean, f_var = lla.predict(torch.tensor(test_dataset.inputs))

plt.plot(
    test_dataset.inputs.flatten()[sort],
    f_mean.detach().cpu().numpy().flatten()[sort],
    label="Predictions",
    color="black",
)
plt.fill_between(
    test_dataset.inputs.flatten()[sort],
    f_mean.detach().cpu().numpy().flatten()[sort]
    - 2 * np.sqrt(f_var.detach().cpu().numpy().flatten()[sort]),
    f_mean.detach().cpu().numpy().flatten()[sort]
    + 2 * np.sqrt(f_var.detach().cpu().numpy().flatten()[sort]),
```

```
    alpha=0.3,
    color="black",
)
plt.ylim(-1.2, 2)
# Remove ticks
plt.xticks([])
plt.yticks([])
plt.savefig("synthetic_regression_lla.pdf", format="pdf", bbox_inches="tight")

plt.show()
```



[33]:
```
ella = ELLA(
    model=copy.deepcopy(f),
    likelihood="regression",
    subsample_size=200,
    n_eigenvalues=20,
    seed=1234,
    y_mean=0,
    y_std=1,
)
```

[34]:
```
ella.fit(train_loader=train_loader, verbose=True)
```

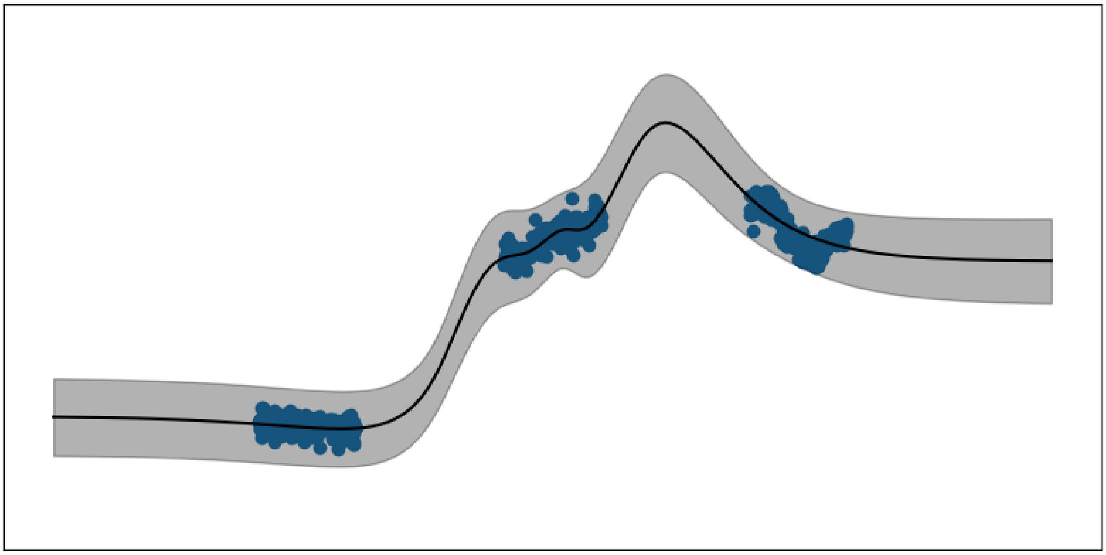
```
Computing Subset Kernel: 100%|      | 2/2 [00:00<00:00, 16.84it/s]
c:\Users\Ludvins\Documents\VariationalUncertaintyEstimation\demos\..\bayesipy\la
place\ella\utils.py:26: RuntimeWarning: K not p.d., added jitter of 10000.0 to
the diagonal
  warnings.warn(
```

```
Computing Dual Parameters: 100%|      | 2/2 [00:00<00:00, 16.38it/s]
Iterating Training Data: 100%|      | 4/4 [00:00<00:00,  5.56it/s]
```

[34]:
```
300
```

[35]:
```python
ella.prior_precision = 0.19904398918151855
ella.sigma_noise = 0.06210554763674736
```

[36]:
```python
plt.figure(figsize=(10, 5))
plt.scatter(train_dataset.inputs, train_dataset.targets, label="Training␣
↪points")


sort = np.argsort(test_dataset.inputs.flatten())

f_mean, f_var = ella.predict(torch.tensor(test_dataset.inputs))

plt.plot(
    test_dataset.inputs.flatten()[sort],
    f_mean.detach().cpu().numpy().flatten()[sort],
    label="Predictions",
    color="black",
)
plt.fill_between(
    test_dataset.inputs.flatten()[sort],
    f_mean.detach().cpu().numpy().flatten()[sort]
    - 2 * np.sqrt(f_var.detach().cpu().numpy().flatten()[sort]),
    f_mean.detach().cpu().numpy().flatten()[sort]
    + 2 * np.sqrt(f_var.detach().cpu().numpy().flatten()[sort]),
    alpha=0.3,
    color="black",
)
plt.ylim(-1.2, 2)
# Remove ticks
plt.xticks([])
plt.yticks([])
plt.savefig("synthetic_regression_ella.pdf", format="pdf", bbox_inches="tight")

plt.show()
```



[37]:
```
valla = VaLLA(
    model=copy.deepcopy(f),
    likelihood="regression",
    inducing_locations="kmeans",
    num_inducing=20,
    noise_variance=np.exp(-5),
    y_mean=0,
    y_std=1,
    seed=1234,
)

loss = valla.fit(
    iterations=40000,
    lr=0.001,
    train_loader=train_loader,
    verbose=True,
)
```

```
Initializing inducing locations… done
```

```
Training :   0%|          | 0/40000 [00:00<?, ? iteration/s]c:\Users\Ludvins\Doc
uments\VariationalUncertaintyEstimation\.venv\Lib\site-
packages\torch\autograd\graph.py:769: UserWarning: Using backward() with
create_graph=True will create a reference cycle between the parameter and its
gradient which can cause a memory leak. We recommend using autograd.grad when
creating the graph to avoid this. If you have to use this function, make sure to
reset the .grad fields of your parameters to None after use to break the cycle
and avoid the leak. (Triggered internally at C:\actions-runner\_work\pytorch\pyt
orch\builder\windows\pytorch\torch\csrc\autograd\engine.cpp:1208.)
```

```
  return Variable._execution_engine.run_backward(  # Calls into the C++ engine
to run the backward pass
Training :   7%|          | 2771/40000 [02:09<29:05, 21.33 iteration/s]
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[37], line 12
      1 valla = VaLLA(
      2     model=copy.deepcopy(f),
      3     likelihood="regression",
   (...)
      9     seed=1234,
     10 )
---> 12 loss = valla.fit(
     13     iterations=40000,
     14     lr=0.001,
     15     train_loader=train_loader,
     16     verbose=True,
     17 )

File c:\Users\Ludvins\Documents\VariationalUncertaintyEstimation\demos\..
 ↪\bayesipy\laplace\valla\src.py:505, in VaLLA.fit(self, iterations, lr,␣
 ↪train_loader, val_loader, val_steps, metrics_cls, verbose, override)
    503 inputs = inputs.to(self.device).to(self.dtype)
    504 targets = targets.to(self.device).to(self.dtype)
--> 505 loss = self.train_step(optimizer, inputs, targets)
    507 losses.append(loss.detach().cpu().numpy())
    509 if val_loader is not None:

File c:\Users\Ludvins\Documents\VariationalUncertaintyEstimation\demos\..
 ↪\bayesipy\laplace\valla\src.py:150, in VaLLA.train_step(self, optimizer, X, y
    147 X = X.to(self.device).to(self.dtype)
    148 y = y.to(self.device)
--> 150 loss = self.loss(X, y)
    152 optimizer.zero_grad()
    154 loss.backward()

File c:\Users\Ludvins\Documents\VariationalUncertaintyEstimation\demos\..
 ↪\bayesipy\laplace\valla\src.py:286, in VaLLA.loss(self, X, y)
    271 def loss(self, X, y):
    272     """Compute the loss of the model.
    273 
    274     Parameters
   (...)
    284         Contains the loss of the model.
    285     """
--> 286     F_mean, F_var = self(X)
    288     # Compute divergence term
```

```
    289     divergence = self.alpha_divergence(F_mean, F_var, y)

File c:\Users\Ludvins\Documents\VariationalUncertaintyEstimation\.
 ↪venv\Lib\site-packages\torch\nn\modules\module.py:1553, in Module.
 ↪_wrapped_call_impl(self, *args, **kwargs)
   1551     return self._compiled_call_impl(*args, **kwargs)  # type:␣
 ↪ignore[misc]
   1552 else:
-> 1553     return self._call_impl(*args, **kwargs)

File c:\Users\Ludvins\Documents\VariationalUncertaintyEstimation\.
 ↪venv\Lib\site-packages\torch\nn\modules\module.py:1562, in Module.
 ↪_call_impl(self, *args, **kwargs)
   1557 # If we don't have any hooks, we want to skip the rest of the logic in
   1558 # this function, and just call forward.
   1559 if not (self._backward_hooks or self._backward_pre_hooks or self.
 ↪_forward_hooks or self._forward_pre_hooks
   1560         or _global_backward_pre_hooks or _global_backward_hooks
   1561         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1562     return forward_call(*args, **kwargs)
   1564 try:
   1565     result = None

File c:\Users\Ludvins\Documents\VariationalUncertaintyEstimation\demos\..
 ↪\bayesipy\laplace\valla\src.py:203, in VaLLA.forward(self, X)
    200     F_mean = self.model[0](X)
    202 # Shape (batch_size)
--> 203 Jx = self.backend.jacobians(X, enable_back_prop=False)
    204 Jz = self.backend.jacobians_on_outputs(
    205     self.inducing_locations,
    206     self.inducing_classes.unsqueeze(-1),
    207     enable_back_prop=self.training,
    208 ).squeeze(1)
    209 var = 1 / torch.exp(self.log_prior_precision)

File c:\Users\Ludvins\Documents\VariationalUncertaintyEstimation\demos\..
 ↪\bayesipy\laplace\valla\backpack_interface.py:36, in BackPackInterface.
 ↪jacobians(self, x, enable_back_prop)
     33 # Enable grads in this section of code
     34 with torch.set_grad_enabled(True):
     35     # Extend model using BackPack converter
---> 36     model = extend(self.model, use_converter=True)
     37     # Set model in evaluation mode to ignore Dropout, BatchNorm..
     38     model.eval()

File c:\Users\Ludvins\Documents\VariationalUncertaintyEstimation\.
 ↪venv\Lib\site-packages\backpack\__init__.py:248, in extend(module, debug,␣
 ↪use_converter)
    245     print("[DEBUG] Extending", module)
```

```
    247 if use_converter:
--> 248     module: GraphModule = convert_module_to_backpack(module, debug)
    249     return extend(module)
    251 for child in module.children():

File c:\Users\Ludvins\Documents\VariationalUncertaintyEstimation\.venv\Lib\site-packages\backpack\custom_module\graph_utils.py:57, in convert_module_to_backpack(module, debug)
     55 module_new = _transform_get_item_to_module(module_new, debug)
     56 module_new = _transform_permute_to_module(module_new, debug)
---> 57 module_new = _transform_transpose_to_module(module_new, debug)
     58 module_new = _transform_lstm_rnn(module_new, debug)
     59 _transform_inplace_to_normal(module_new, debug)

File c:\Users\Ludvins\Documents\VariationalUncertaintyEstimation\.venv\Lib\site-packages\backpack\custom_module\graph_utils.py:316, in _transform_transpose_to_module(module, debug)
    314 if debug:
    315     print(f"\tBegin transformation: {target_method} -> Permute")
--> 316 graph: Graph = BackpackTracer().trace(module)
    318 nodes = [
    319     n
    320     for n in graph.nodes
    321     if (n.op == "call_function" and target_function in str(n.target))
    322     or (n.op == "call_method" and target_method == str(n.target))
    323 ]
    325 for node in nodes:

File c:\Users\Ludvins\Documents\VariationalUncertaintyEstimation\.venv\Lib\site-packages\torch\fx\_symbolic_trace.py:802, in Tracer.trace(self, root, concrete_args)
    795     _autowrap_check(
    796         patcher,
    797         getattr(getattr(mod, "forward", mod), "__globals__", {}),
    798         self._autowrap_function_ids,
    799     )
    800     return self.call_module(mod, forward, args, kwargs)
--> 802 with _Patcher() as patcher:
    803     # allow duplicate patches to support the case of nested calls
    804     patcher.patch_method(
    805         torch.nn.Module,
    806         "__getattr__",
    807         module_getattr_wrapper,
    808         deduplicate=False,
    809     )
    810     patcher.patch_method(
    811         torch.nn.Module, "__call__", module_call_wrapper, deduplicate=False
```

```
    812         )

File c:\Users\Ludvins\Documents\VariationalUncertaintyEstimation\.
 ↪venv\Lib\site-packages\torch\fx\_symbolic_trace.py:1068, in _Patcher.
 ↪__exit__(self, exc_type, exc_val, exc_tb)
   1064 def __exit__(self, exc_type, exc_val, exc_tb):
   1065     """
   1066     Undo all the changes made via self.patch() and self.patch_method()
   1067     """
-> 1068     while self.patches_made:
   1069         # unpatch in reverse order to handle duplicates correctly
   1070         self.patches_made.pop().revert()
   1071     self.visited.clear()

KeyboardInterrupt:
```

[31]:

```python
plt.figure(figsize=(10, 5))
plt.scatter(train_dataset.inputs, train_dataset.targets, label="Training␣
 ↪points")


sort = np.argsort(test_dataset.inputs.flatten())

f_mean, f_var = valla.predict(torch.tensor(test_dataset.inputs))

plt.plot(
    test_dataset.inputs.flatten()[sort],
    f_mean.detach().cpu().numpy().flatten()[sort],
    label="Predictions",
    color="black",
)
plt.fill_between(
    test_dataset.inputs.flatten()[sort],
    f_mean.detach().cpu().numpy().flatten()[sort]
    - 2 * np.sqrt(f_var.detach().cpu().numpy().flatten()[sort]),
    f_mean.detach().cpu().numpy().flatten()[sort]
    + 2 * np.sqrt(f_var.detach().cpu().numpy().flatten()[sort]),
    alpha=0.3,
    color="black",
)
plt.ylim(-1.2, 2)
# Remove ticks
plt.xticks([])
plt.yticks([])
plt.savefig("synthetic_regression_valla.pdf", format="pdf", bbox_inches="tight")

plt.show()
```