GPyTorch Documentation

Release 0.0.1 (alpha)

Cornellius GP

Oct 20, 2018

Tutorials:

CHAPTER 1

GPyTorch Regression Tutorial

1.1 Introduction

In this notebook, we demonstrate many of the design features of GPyTorch using the simplest example, training an RBF kernel Gaussian process on a simple function. We'll be modeling the function

$$y = \sin(2\pi x) + \epsilon$$
$$\epsilon \sim \mathcal{N}(0, 0.2)$$

with 11 training examples, and testing on 51 test examples.

Note: this notebook is not necessarily intended to teach the mathematical background of Gaussian processes, but rather how to train a simple one and make predictions in GPyTorch. For a mathematical treatment, Chapter 2 of Gaussian Processes for Machine Learning provides a very thorough introduction to GP regression (this entire text is highly recommended): http://www.gaussianprocess.org/gpml/chapters/RW2.pdf

```
In [1]: import math
        import torch
        import gpytorch
        from matplotlib import pyplot as plt

        %matplotlib inline
        %load_ext autoreload
        %autoreload 2
```

1.1.1 Set up training data

In the next cell, we set up the training data for this example. We'll be using 11 regularly spaced points on [0,1] which we evaluate the function on and add Gaussian noise to get the training labels.

```
In [2]: # Training data is 11 points in [0,1] inclusive regularly spaced
        train_x = torch.linspace(0, 1, 100)
        # True function is sin(2*pi*x) with Gaussian noise
        train_y = torch.sin(train_x * (2 * math.pi)) + torch.randn(train_x.size()) * 0.2
```

1.2 Setting up the model

The next cell demonstrates the most critical features of a user-defined Gaussian process model in GPyTorch. Building a GP model in GPyTorch is different in a number of ways.

First in contrast to many existing GP packages, we do not provide full GP models for the user. Rather, we provide *the tools necessary to quickly construct one*. This is because we believe, analogous to building a neural network in standard PyTorch, it is important to have the flexibility to include whatever components are necessary. As can be seen in more complicated examples, like the *CIFAR10 Deep Kernel Learning* example which combines deep learning and Gaussian processes, this allows the user great flexibility in designing custom models.

For most GP regression models, you will need to construct the following GPyTorch objects:

1. A **GP Model** (`gpytorch.models.ExactGP`) - This handles most of the inference.
2. A **Likelihood** (`gpytorch.likelihoods.GaussianLikelihood`) - This is the most common likelihood used for GP regression.
3. A **Mean** - This defines the prior mean of the GP.

- If you don't know which mean to use, a `gpytorch.means.ConstantMean()` is a good place to start.

1. A **Kernel** - This defines the prior covariance of the GP.

- If you don't know which kernel to use, a `gpytorch.kernels.ScaleKernel(gpytorch.kernels.RBFKernel())` is a good place to start.

1. A **MultivariateNormal** Distribution (`gpytorch.distributions.MultivariateNormal`) - This is the object used to represent multivariate normal distributions.

The components of a user built (Exact, i.e. non-variational) GP model in GPyTorch are, broadly speaking:

1. An `__init__` method that takes the training data and a likelihood, and constructs whatever objects are necessary for the model's `forward` method. This will most commonly include things like a mean module and a kernel module.
2. A `forward` method that takes in some $n \times d$ data `x` and returns a `MultivariateNormal` with the *prior* mean and covariance evaluated at `x`. In other words, we return the vector $\mu(x)$ and the $n \times n$ matrix K_{xx} representing the prior mean and covariance matrix of the GP.

This specification leaves a large amount of flexibility when defining a model. For example, to compose two kernels via addition, you can either add the kernel modules directly:

```
self.covar_module = ScaleKernel(RBFKernel() + WhiteNoiseKernel())
```

Or you can add the outputs of the kernel in the forward method:

```
covar_x = self.rbf_kernel_module(x) + self.white_noise_module(x)
```

```
In [3]: # We will use the simplest form of GP model, exact inference
        class ExactGPModel(gpytorch.models.ExactGP):
            def __init__(self, train_x, train_y, likelihood):
                super(ExactGPModel, self).__init__(train_x, train_y, likelihood)
                self.mean_module = gpytorch.means.ConstantMean()
                self.covar_module = gpytorch.kernels.ScaleKernel(gpytorch.kernels.RBFKernel())

            def forward(self, x):
                mean_x = self.mean_module(x)
                covar_x = self.covar_module(x)
                return gpytorch.distributions.MultivariateNormal(mean_x, covar_x)
```

```
        # initialize likelihood and model
        likelihood = gpytorch.likelihoods.GaussianLikelihood()
        model = ExactGPModel(train_x, train_y, likelihood)
```

1.2.1 Model modes

Like most PyTorch modules, the `ExactGP` has a `.train()` and `.eval()` mode. - `.train()` mode is for optimizing model hyperameters. - `.eval()` mode is for computing predictions through the model posterior.

1.3 Training the model

In the next cell, we handle using Type-II MLE to train the hyperparameters of the Gaussian process.

The most obvious difference here compared to many other GP implementations is that, as in standard PyTorch, the core training loop is written by the user. In GPyTorch, we make use of the standard PyTorch optimizers as from `torch.optim`, and all trainable parameters of the model should be of type `torch.nn.Parameter`. Because GP models directly extend `torch.nn.Module`, calls to methods like `model.parameters()` or `model.named_parameters()` function as you might expect coming from PyTorch.

In most cases, the boilerplate code below will work well. It has the same basic components as the standard PyTorch training loop:

1. Zero all parameter gradients
2. Call the model and compute the loss
3. Call backward on the loss to fill in gradients
4. Take a step on the optimizer

However, defining custom training loops allows for greater flexibility. For example, it is easy to save the parameters at each step of training, or use different learning rates for different parameters (which may be useful in deep kernel learning for example).

```
In [4]: # Find optimal model hyperparameters
        model.train()
        likelihood.train()

        # Use the adam optimizer
        optimizer = torch.optim.Adam([
            {'params': model.parameters()},  # Includes GaussianLikelihood parameters
        ], lr=0.1)

        # "Loss" for GPs - the marginal log likelihood
        mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)

        training_iter = 50
        for i in range(training_iter):
            # Zero gradients from previous iteration
            optimizer.zero_grad()
            # Output from model
            output = model(train_x)
            # Calc loss and backprop gradients
            loss = -mll(output, train_y)
            loss.backward()
            print('Iter %d/%d - Loss: %.3f   log_lengthscale: %.3f   log_noise: %.3f' % (
                i + 1, training_iter, loss.item(),
                model.covar_module.base_kernel.log_lengthscale.item(),
```

```
            model.likelihood.log_noise.item()
        ))
        optimizer.step()
```

```
Iter 1/50 - Loss: 1.084   log_lengthscale: 0.000   log_noise: 0.000
Iter 2/50 - Loss: 1.043   log_lengthscale: -0.100   log_noise: -0.100
Iter 3/50 - Loss: 1.004   log_lengthscale: -0.196   log_noise: -0.200
Iter 4/50 - Loss: 0.964   log_lengthscale: -0.293   log_noise: -0.300
Iter 5/50 - Loss: 0.922   log_lengthscale: -0.387   log_noise: -0.399
Iter 6/50 - Loss: 0.877   log_lengthscale: -0.479   log_noise: -0.499
Iter 7/50 - Loss: 0.825   log_lengthscale: -0.572   log_noise: -0.598
Iter 8/50 - Loss: 0.767   log_lengthscale: -0.667   log_noise: -0.698
Iter 9/50 - Loss: 0.705   log_lengthscale: -0.762   log_noise: -0.799
Iter 10/50 - Loss: 0.644   log_lengthscale: -0.860   log_noise: -0.899
Iter 11/50 - Loss: 0.590   log_lengthscale: -0.960   log_noise: -1.001
Iter 12/50 - Loss: 0.543   log_lengthscale: -1.058   log_noise: -1.102
Iter 13/50 - Loss: 0.502   log_lengthscale: -1.150   log_noise: -1.204
Iter 14/50 - Loss: 0.462   log_lengthscale: -1.234   log_noise: -1.306
Iter 15/50 - Loss: 0.426   log_lengthscale: -1.303   log_noise: -1.408
Iter 16/50 - Loss: 0.389   log_lengthscale: -1.360   log_noise: -1.509
Iter 17/50 - Loss: 0.360   log_lengthscale: -1.404   log_noise: -1.611
Iter 18/50 - Loss: 0.321   log_lengthscale: -1.432   log_noise: -1.712
Iter 19/50 - Loss: 0.280   log_lengthscale: -1.454   log_noise: -1.812
Iter 20/50 - Loss: 0.250   log_lengthscale: -1.465   log_noise: -1.911
Iter 21/50 - Loss: 0.227   log_lengthscale: -1.469   log_noise: -2.010
Iter 22/50 - Loss: 0.188   log_lengthscale: -1.461   log_noise: -2.108
Iter 23/50 - Loss: 0.158   log_lengthscale: -1.442   log_noise: -2.204
Iter 24/50 - Loss: 0.125   log_lengthscale: -1.411   log_noise: -2.300
Iter 25/50 - Loss: 0.095   log_lengthscale: -1.377   log_noise: -2.393
Iter 26/50 - Loss: 0.070   log_lengthscale: -1.340   log_noise: -2.485
Iter 27/50 - Loss: 0.050   log_lengthscale: -1.298   log_noise: -2.574
Iter 28/50 - Loss: 0.032   log_lengthscale: -1.256   log_noise: -2.662
Iter 29/50 - Loss: 0.014   log_lengthscale: -1.218   log_noise: -2.746
Iter 30/50 - Loss: 0.003   log_lengthscale: -1.182   log_noise: -2.828
Iter 31/50 - Loss: -0.001   log_lengthscale: -1.148   log_noise: -2.906
Iter 32/50 - Loss: -0.008   log_lengthscale: -1.121   log_noise: -2.980
Iter 33/50 - Loss: -0.012   log_lengthscale: -1.102   log_noise: -3.049
Iter 34/50 - Loss: -0.011   log_lengthscale: -1.103   log_noise: -3.114
Iter 35/50 - Loss: -0.014   log_lengthscale: -1.114   log_noise: -3.174
Iter 36/50 - Loss: -0.014   log_lengthscale: -1.138   log_noise: -3.228
Iter 37/50 - Loss: -0.010   log_lengthscale: -1.169   log_noise: -3.275
Iter 38/50 - Loss: -0.011   log_lengthscale: -1.204   log_noise: -3.317
Iter 39/50 - Loss: -0.008   log_lengthscale: -1.239   log_noise: -3.352
Iter 40/50 - Loss: -0.001   log_lengthscale: -1.270   log_noise: -3.380
Iter 41/50 - Loss: -0.005   log_lengthscale: -1.296   log_noise: -3.401
Iter 42/50 - Loss: 0.008   log_lengthscale: -1.317   log_noise: -3.415
Iter 43/50 - Loss: 0.001   log_lengthscale: -1.331   log_noise: -3.422
Iter 44/50 - Loss: 0.009   log_lengthscale: -1.343   log_noise: -3.423
Iter 45/50 - Loss: 0.001   log_lengthscale: -1.350   log_noise: -3.419
Iter 46/50 - Loss: -0.001   log_lengthscale: -1.360   log_noise: -3.410
Iter 47/50 - Loss: 0.007   log_lengthscale: -1.374   log_noise: -3.397
Iter 48/50 - Loss: 0.000   log_lengthscale: -1.388   log_noise: -3.380
Iter 49/50 - Loss: -0.010   log_lengthscale: -1.396   log_noise: -3.359
Iter 50/50 - Loss: -0.008   log_lengthscale: -1.404   log_noise: -3.337
```

1.4 Make predictions with the model

In the next cell, we make predictions with the model. To do this, we simply put the model and likelihood in eval mode, and call both modules on the test data.

Just as a user defined GP model returns a `MultivariateNormal` containing the prior mean and covariance from forward, a trained GP model in eval mode returns a `MultivariateNormal` containing the posterior mean and covariance. Thus, getting the predictive mean and variance, and then sampling functions from the GP at the given test points could be accomplished with calls like:

```
f_preds = model(test_x)
y_preds = likelihood(model(test_x))

f_mean = f_preds.mean
f_var = f_preds.variance
f_covar = f_preds.covariance_matrix
f_samples = f_preds.sample(sample_shape=torch.Size(1000,))
```

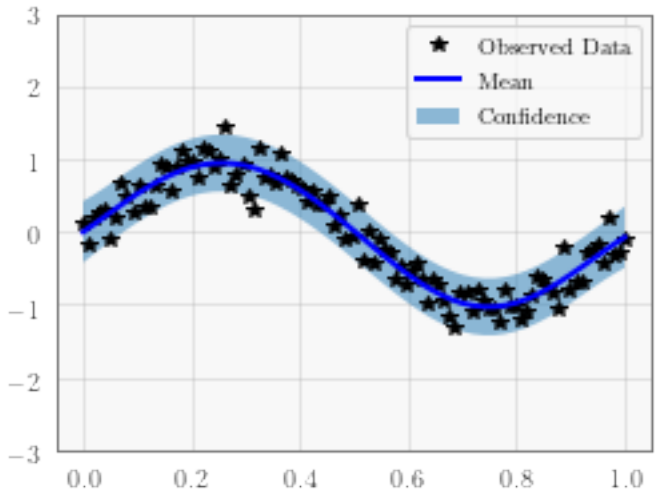
The `gpytorch.fast_pred_var` context is not needed, but here we are giving a preview of using one of our cool features, getting faster predictive distributions using LOVE.

```
In [5]: # Get into evaluation (predictive posterior) mode
        model.eval()
        likelihood.eval()

        # Test points are regularly spaced along [0,1]
        # Make predictions by feeding model through likelihood
        with torch.no_grad(), gpytorch.fast_pred_var():
            test_x = torch.linspace(0, 1, 51)
            observed_pred = likelihood(model(test_x))
```

1.5 Plot the model fit

In the next cell, we plot the mean and confidence region of the Gaussian process model. The `confidence_region` method is a helper method that returns 2 standard deviations above and below the mean.

```
In [6]: with torch.no_grad():
            # Initialize plot
            f, ax = plt.subplots(1, 1, figsize=(4, 3))

            # Get upper and lower confidence bounds
            lower, upper = observed_pred.confidence_region()
            # Plot training data as black stars
            ax.plot(train_x.numpy(), train_y.numpy(), 'k*')
            # Plot predictive means as blue line
            ax.plot(test_x.numpy(), observed_pred.mean.numpy(), 'b')
            # Shade between the lower and upper confidence bounds
            ax.fill_between(test_x.numpy(), lower.numpy(), upper.numpy(), alpha=0.5)
            ax.set_ylim([-3, 3])
            ax.legend(['Observed Data', 'Mean', 'Confidence'])
```



In [ ]:

CHAPTER 2

GPyTorch Classification Tutorial

2.1 Introduction

This example is the simplest form of using an RBF kernel in an `VariationalGP` module for classification. This basic model is usable when there is not much training data and no advanced techniques are required.

In this example, we're modeling a unit wave with period 1/2 centered with positive values @ x=0. We are going to classify the points as either +1 or -1.

Variational inference uses the assumption that the posterior distribution factors multiplicatively over the input variables. This makes approximating the distribution via the KL divergence possible to obtain a fast approximation to the posterior. For a good explanation of variational techniques, sections 4-6 of the following may be useful: https://www.cs.princeton.edu/courses/archive/fall11/cos597C/lectures/variational-inference-i.pdf

```
In [1]: import math
        import torch
        import gpytorch
        from matplotlib import pyplot as plt

        %matplotlib inline
```

2.1.1 Set up training data

In the next cell, we set up the training data for this example. We'll be using 15 regularly spaced points on [0,1] which we evaluate the function on and add Gaussian noise to get the training labels. Labels are unit wave with period 1/2 centered with positive values @ x=0.

```
In [2]: train_x = torch.linspace(0, 1, 10)
        train_y = torch.sign(torch.cos(train_x * (4 * math.pi)))
```

2.2 Setting up the classification model

The next cell demonstrates the simplist way to define a classification Gaussian process model in GPyTorch. If you have already done the *GP regression tutorial*, you have already seen how GPyTorch model construction differs from other GP packages. In particular, the GP model expects a user to write out a `forward` method in a way analogous to PyTorch models. This gives the user the most possible flexibility.

Since exact inference is intractable for GP classification, GPyTorch approximates the classification posterior using **variational inference.** We believe that variational inference is ideal for a number of reasons. Firstly, variational inference commonly relies on gradient descent techniques, which take full advantage of PyTorch's autograd. This reduces the amount of code needed to develop complex variational models. Additionally, variational inference can be performed with stochastic gradient decent, which can be extremely scalable for large datasets.

If you are unfamiliar with variational inference, we recommend the following resources: - Variational Inference: A Review for Statisticians by David M. Blei, Alp Kucukelbir, Jon D. McAuliffe. - Scalable Variational Gaussian Process Classification by James Hensman, Alex Matthews, Zoubin Ghahramani.

2.2.1 The necessary casses

For most GP regression models, you will need to construct the following GPyTorch objects:

1. A **GP Model** (`gpytorch.models.VariationalGP`) - This handles basic variational inference.
2. A **Likelihood** (`gpytorch.likelihoods.BernoulliLikelihood`) - This is a good likelihood for binary classification
3. A **Mean** - This defines the prior mean of the GP.

- If you don't know which mean to use, a `gpytorch.means.ConstantMean()` is a good place to start.

1. A **Kernel** - This defines the prior covariance of the GP.

- If you don't know which kernel to use, a `gpytorch.kernels.ScaleKernel(gpytorch.kernels.RBFKernel())` is a good place to start.

1. A **MultivariateNormal** Distribution (`gpytorch.distributions.MultivariateNormal`) - This is the object used to represent multivariate normal distributions.

The GP Model

The `VariationalGP` model is GPyTorch's simplist approximate inference model. It approximates the true posterior with a multivariate normal distribution. The model defines all the variational parameters that are needed, and keeps all of this information under the hood.

The components of a user built `VariationalGP` model in GPyTorch are:

1. An `__init__` method that takes the training data as an input. The `__init__` function will also construct a mean module, a kernel module, and whatever other modules might be necessary.
2. A `forward` method that takes in some $n \times d$ data `x` and returns a MultivariateNormal with the *prior* mean and covariance evaluated at `x`. In other words, we return the vector $\mu(x)$ and the $n \times n$ matrix K_{xx} representing the prior mean and covariance matrix of the GP.

(For those who are unfamiliar with GP classification: even though we are performing classification, the GP model still returns a `MultivariateNormal`. The likelihood transforms this latent Gaussian variable into a Bernoulli variable)

Here we present a simple classification model, but it is posslbe to construct more complex models. See some of the scalable classification examples or deep kernel learning examples for some other examples.

```
In [3]: class GPClassificationModel(gpytorch.models.VariationalGP):
            def __init__(self, train_x):
                super(GPClassificationModel, self).__init__(train_x)
                self.mean_module = gpytorch.means.ConstantMean()
                self.covar_module = gpytorch.kernels.ScaleKernel(gpytorch.kernels.RBFKernel())

            def forward(self, x):
                mean_x = self.mean_module(x)
                covar_x = self.covar_module(x)
                latent_pred = gpytorch.distributions.MultivariateNormal(mean_x, covar_x)
                return latent_pred

        # Initialize model and likelihood
        model = GPClassificationModel(train_x)
        likelihood = gpytorch.likelihoods.BernoulliLikelihood()
```

2.2.2 Model modes

Like most PyTorch modules, the `ExactGP` has a `.train()` and `.eval()` mode. - `.train()` mode is for optimizing variational parameters model hyperameters. - `.eval()` mode is for computing predictions through the model posterior.

2.3 Learn the variational parameters (and other hyperparameters)

In the next cell, we optimize the variational parameters of our Gaussian process. In addition, this optimization loop also performs Type-II MLE to train the hyperparameters of the Gaussian process.

The most obvious difference here compared to many other GP implementations is that, as in standard PyTorch, the core training loop is written by the user. In GPyTorch, we make use of the standard PyTorch optimizers as from `torch.optim`, and all trainable parameters of the model should be of type `torch.nn.Parameter`. The variational parameters are predefined as part of the `VariationalGP` model.

In most cases, the boilerplate code below will work well. It has the same basic components as the standard PyTorch training loop:

1. Zero all parameter gradients
2. Call the model and compute the loss
3. Call backward on the loss to fill in gradients
4. Take a step on the optimizer

However, defining custom training loops allows for greater flexibility. For example, it is possible to learn the variational parameters and kernel hyperparameters with different learning rates.

```
In [4]: # Find optimal model hyperparameters
        model.train()
        likelihood.train()

        # Use the adam optimizer
        optimizer = torch.optim.Adam([
            {'params': model.parameters()},
            # BernoulliLikelihood has no parameters
        ], lr=0.1)

        # "Loss" for GPs - the marginal log likelihood
        # num_data refers to the amount of training data
```

```
        mll = gpytorch.mlls.VariationalMarginalLogLikelihood(likelihood, model, num_data=len(train_y))

        training_iter = 50
        for i in range(training_iter):
            # Zero backpropped gradients from previous iteration
            optimizer.zero_grad()
            # Get predictive output
            output = model(train_x)
            # Calc loss and backprop gradients
            loss = -mll(output, train_y)
            loss.backward()
            print('Iter %d/%d - Loss: %.3f' % (
                i + 1, training_iter, loss.item(),
            ))
            optimizer.step()
```

```
Iter 1/50 - Loss: 326.033
Iter 2/50 - Loss: 229.205
Iter 3/50 - Loss: 147.791
Iter 4/50 - Loss: 94.322
Iter 5/50 - Loss: 58.356
Iter 6/50 - Loss: 34.048
Iter 7/50 - Loss: 20.180
Iter 8/50 - Loss: 14.053
Iter 9/50 - Loss: 12.395
Iter 10/50 - Loss: 11.749
Iter 11/50 - Loss: 10.924
Iter 12/50 - Loss: 10.087
Iter 13/50 - Loss: 9.328
Iter 14/50 - Loss: 8.994
Iter 15/50 - Loss: 7.611
Iter 16/50 - Loss: 6.660
Iter 17/50 - Loss: 6.201
Iter 18/50 - Loss: 5.975
Iter 19/50 - Loss: 5.992
Iter 20/50 - Loss: 5.758
Iter 21/50 - Loss: 5.404
Iter 22/50 - Loss: 5.043
Iter 23/50 - Loss: 4.632
Iter 24/50 - Loss: 4.990
Iter 25/50 - Loss: 5.021
Iter 26/50 - Loss: 5.250
Iter 27/50 - Loss: 4.832
Iter 28/50 - Loss: 4.951
Iter 29/50 - Loss: 5.426
Iter 30/50 - Loss: 4.512
Iter 31/50 - Loss: 5.000
Iter 32/50 - Loss: 4.938
Iter 33/50 - Loss: 4.912
Iter 34/50 - Loss: 4.636
Iter 35/50 - Loss: 4.414
Iter 36/50 - Loss: 4.946
Iter 37/50 - Loss: 4.769
Iter 38/50 - Loss: 4.891
Iter 39/50 - Loss: 4.702
Iter 40/50 - Loss: 4.603
Iter 41/50 - Loss: 4.739
Iter 42/50 - Loss: 4.913
Iter 43/50 - Loss: 4.942
```

```
Iter 44/50 - Loss: 4.547
Iter 45/50 - Loss: 4.944
Iter 46/50 - Loss: 4.412
Iter 47/50 - Loss: 4.744
Iter 48/50 - Loss: 4.679
Iter 49/50 - Loss: 4.549
Iter 50/50 - Loss: 4.697
```

2.4 Make predictions with the model

In the next cell, we make predictions with the model. To do this, we simply put the model and likelihood in eval mode, and call both modules on the test data.

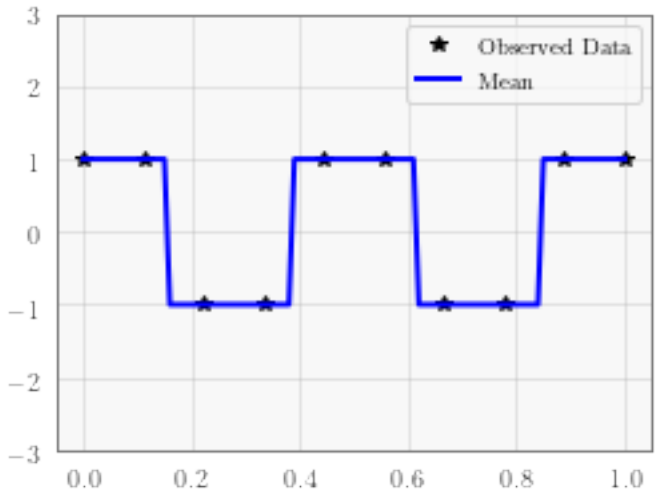
In `.eval()` mode, when we call `model()` - we get GP's latent posterior predictions. These will be MultivariateNormal distributions. But since we are performing binary classification, we want to transform these outputs to classification probabilities using our likelihood.

When we call `likelihood(model())`, we get a `torch.distributions.Bernoulli` distribution, which represents our posterior probability that the data points belong to the positive class.

```
f_preds = model(test_x)
y_preds = likelihood(model(test_x))

f_mean = f_preds.mean
f_samples = f_preds.sample(sample_shape=torch.Size((1000,))
```

The `gpytorch.fast_pred_var` context is not needed, but here we are giving a preview of using one of our cool features, getting faster predictive distributions using LOVE.

```
In [5]: # Go into eval mode
        model.eval()
        likelihood.eval()

        with torch.no_grad(), gpytorch.fast_pred_var():
            # Test x are regularly spaced by 0.01 0,1 inclusive
            test_x = torch.linspace(0, 1, 101)
            # Get classification predictions
            observed_pred = likelihood(model(test_x))

            # Initialize fig and axes for plot
            f, ax = plt.subplots(1, 1, figsize=(4, 3))
            ax.plot(train_x.numpy(), train_y.numpy(), 'k*')
            # Get the predicted labels (probabilites of belonging to the positive class)
            # Transform these probabilities to be 0/1 labels
            pred_labels = observed_pred.mean.ge(0.5).float().mul(2).sub(1)
            ax.plot(test_x.numpy(), pred_labels.numpy(), 'b')
            ax.set_ylim([-3, 3])
            ax.legend(['Observed Data', 'Mean', 'Confidence'])
```



In [ ]:

CHAPTER 3

Overview of Examples

This `examples` directory provides numerous ipython notebooks that demonstrate the use of GPyTorch.

1. *Getting started*
2. *Specialty Models/Tasks*
3. *Scalable GP Regression Models*
4. *Scalable GP Classification Models*
5. *Deep Kernel Learning*

3.1 Getting started

These are no-frills GP models, which will work in most small data applications. If you are looking to get familiar with GPyTorch, start here.

- **Regression** - check out the *simple regression example*
- **Classification** - check out the *simple classification example*

Some advanced techniques that you can apply to soup up these simple models:

- **GPU Acceleration** - see how to use CUDA with GPyTorch
- **Fast Predictive Variances w/ LOVE** - see *how to get really fast predictions with LOVE*

3.2 Specialty Models and Tasks

- **Multitask GP Regression** - check out the examples in the multitask GP folder
- **Bayesian Optimization** - example coming soon!

3.3 Scalable GP Regression Models

If you have more than ~1,000 training data points, the simple GP models might start acting a bit slow. There are multiple methods to scale up GP regression, and the correct choice depends on your application. GPyTorch supports the following inducing point methods:

- **KISS-GP Regression** - more info
  - A *simple KISS-GP example* for one-dimensional data
  - *An example* for low-dimensional data
  - An *example that combines KISS-GP with Deep Kernel Learning*
  - And more!
- **SGPR** - more info
  - Example coming soon!

While there are lots of different choices, switching between methods requires a quick one-line change to your model. In addition, it is fairly straightforward to create your own custom scalable GP method. (Tutorial coming soon!) This is especially useful if your data is structured (e.g. if your data lies on a regularly-spaced grid).

Additionally, it is possible to use stochastic variational inference for regression problems. This is useful if you have an extremely large dataset. Some examples:

- A 1D example combining KISS-GP and stochastic variational inference
- An example combining KISS-GP, Deep Kernel Learning, and stochastic variational inference

3.4 Scalable GP Classification Models

There are multiple methods for scalable GP classification, and the correct choice depends on your application. Some examples:

- **KISS-GP Classification**
  - A *simple KISSGP example* for one-dimensional data
  - An example for low-dimensional data
  - And more!

3.5 Deep Kernel Learning

GPyTorch seemlessly integrates with PyTorch, making it extremely easy to combine GPs with neural networks. The following examples use **Deep Kernel Learning**:

- A *large-scale regression problem* with Deep Kernel Learning
- Training *a GP for CIFAR image classification* with Deep Kernel Learning
- And more!

CHAPTER 4

Simple GP Regression

Here are examples for simple GP regression models. These examples will work for small to medium sized datasets (~2,000 data points). All examples here use exact GP inference (and therefore assume a Gaussian noise observation model).

New to GPyTorch? Check out the *GP Regression Tutorial*!

- *GP Regression Tutorial*
  - This is the simplist of the notebooks - a GP regression model with an RBF kernel. Start here if you are new to GPyTorch, or new to GPs in general.
- *Spectral Mixture GP Regression*
  - This notebook expands on previous eample with a more complex kernel. The spectral mixture kernel is a great choice if you have a complex extrapolation problem.
- *GP Regression (CUDA) with Fast Variances (LOVE)*
  - This notebook demonstrates LOVE, a technique to rapidly speed up predictive variance computaitons. Check out this notebook to see how to use LOVE in GPyTorch, and how it compares to standard variance computations.

4.1 GP Regression with a Spectral Mixture Kernel

4.1.1 Introduction

This example shows how to use a `SpectralMixtureKernel` module on an `ExactGP` model. This module is designed for - When you want to use exact inference (e.g. for regression) - When you want to use a more sophisticated kernel than RBF

Function to be modeled is $sin(2\pi x)$

The Spectral Mixture (SM) kernel was invented and discussed in this paper: https://arxiv.org/pdf/1302.4245.pdf

```
In [1]: import math
        import torch
        import gpytorch
        from matplotlib import pyplot as plt

        %matplotlib inline
        %load_ext autoreload
        %autoreload 2
```

Set up training data

In the next cell, we set up the training data for this example. We'll be using 15 regularly spaced points on [0,1] which we evaluate the function on and add Gaussian noise to get the training labels.

```
In [2]: train_x = torch.linspace(0, 1, 15)
        train_y = torch.sin(train_x * (2 * math.pi))
```

4.1.2 Set up the model

The model should be very similar to the `ExactGP` model in the *simple regression example*.

The only difference is here, we're using a more complex kernel (the `SpectralMixtureKernel`). This kernel requires careful initialization to work properly. To that end, in the model `__init__` function, we call

```
self.covar_module = gpytorch.kernels.SpectralMixtureKernel(n_mixtures=4)
self.covar_module.initialize_from_data(train_x, train_y)
```

This ensures that, when we perform optimization to learn kernel hyperparameters, we will be starting from a reasonable initialization.

```
In [3]: class SpectralMixtureGPModel(gpytorch.models.ExactGP):
            def __init__(self, train_x, train_y, likelihood):
                super(SpectralMixtureGPModel, self).__init__(train_x, train_y, likelihood)
                self.mean_module = gpytorch.means.ConstantMean()
                self.covar_module = gpytorch.kernels.SpectralMixtureKernel(num_mixtures=4)
                self.covar_module.initialize_from_data(train_x, train_y)

            def forward(self,x):
                mean_x = self.mean_module(x)
                covar_x = self.covar_module(x)
                return gpytorch.distributions.MultivariateNormal(mean_x, covar_x)


        likelihood = gpytorch.likelihoods.GaussianLikelihood()
        model = SpectralMixtureGPModel(train_x, train_y, likelihood)
```

4.1.3 Training the model

In the next cell, we handle using Type-II MLE to train the hyperparameters of the Gaussian process. The spectral mixture kernel's hyperparameters start from what was specified in `initialize_from_data`.

See the *simple regression example* for more info on this step.

```
In [4]: # Find optimal model hyperparameters
        model.train()
        likelihood.train()
```

```
        # Use the adam optimizer
        optimizer = torch.optim.Adam(model.parameters(), lr=0.1)

        # "Loss" for GPs - the marginal log likelihood
        mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)

        training_iter = 100
        for i in range(training_iter):
            optimizer.zero_grad()
            output = model(train_x)
            loss = -mll(output, train_y)
            loss.backward()
            print('Iter %d/%d - Loss: %.3f' % (i + 1, training_iter, loss.item()))
            optimizer.step()
```

```
Iter 1/100 - Loss: 1.339
Iter 2/100 - Loss: 1.303
Iter 3/100 - Loss: 1.248
Iter 4/100 - Loss: 1.219
Iter 5/100 - Loss: 1.170
Iter 6/100 - Loss: 1.109
Iter 7/100 - Loss: 1.088
Iter 8/100 - Loss: 1.059
Iter 9/100 - Loss: 1.008
Iter 10/100 - Loss: 0.953
Iter 11/100 - Loss: 0.906
Iter 12/100 - Loss: 0.887
Iter 13/100 - Loss: 0.811
Iter 14/100 - Loss: 0.762
Iter 15/100 - Loss: 0.715
Iter 16/100 - Loss: 0.683
Iter 17/100 - Loss: 0.639
Iter 18/100 - Loss: 0.585
Iter 19/100 - Loss: 0.542
Iter 20/100 - Loss: 0.498
Iter 21/100 - Loss: 0.467
Iter 22/100 - Loss: 0.402
Iter 23/100 - Loss: 0.349
Iter 24/100 - Loss: 0.313
Iter 25/100 - Loss: 0.251
Iter 26/100 - Loss: 0.214
Iter 27/100 - Loss: 0.172
Iter 28/100 - Loss: 0.148
Iter 29/100 - Loss: 0.122
Iter 30/100 - Loss: 0.036
Iter 31/100 - Loss: -0.020
Iter 32/100 - Loss: -0.073
Iter 33/100 - Loss: -0.102
Iter 34/100 - Loss: -0.163
Iter 35/100 - Loss: -0.146
Iter 36/100 - Loss: -0.174
Iter 37/100 - Loss: -0.216
Iter 38/100 - Loss: -0.289
Iter 39/100 - Loss: -0.393
Iter 40/100 - Loss: -0.430
Iter 41/100 - Loss: -0.331
Iter 42/100 - Loss: -0.388
Iter 43/100 - Loss: -0.504
Iter 44/100 - Loss: -0.629
```

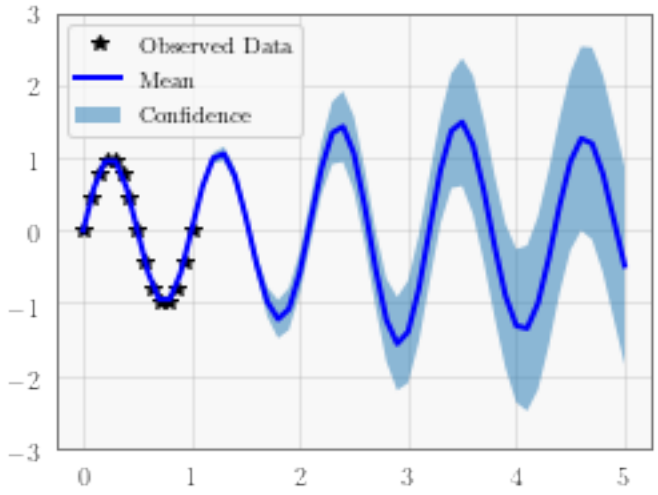
```
Iter 45/100 - Loss: -0.570
Iter 46/100 - Loss: -0.578
Iter 47/100 - Loss: -0.728
Iter 48/100 - Loss: -0.787
Iter 49/100 - Loss: -0.186
Iter 50/100 - Loss: -0.532
Iter 51/100 - Loss: -0.850
Iter 52/100 - Loss: -0.914
Iter 53/100 - Loss: -0.879
Iter 54/100 - Loss: -0.815
Iter 55/100 - Loss: -0.804
Iter 56/100 - Loss: -0.808
Iter 57/100 - Loss: -0.850
Iter 58/100 - Loss: -0.939
Iter 59/100 - Loss: -1.047
Iter 60/100 - Loss: -1.128
Iter 61/100 - Loss: -1.181
Iter 62/100 - Loss: -1.210
Iter 63/100 - Loss: -1.181
Iter 64/100 - Loss: -1.044
Iter 65/100 - Loss: -0.988
Iter 66/100 - Loss: -1.113
Iter 67/100 - Loss: -1.085
Iter 68/100 - Loss: -1.284
Iter 69/100 - Loss: -1.252
Iter 70/100 - Loss: -1.305
Iter 71/100 - Loss: -1.318
Iter 72/100 - Loss: -1.300
Iter 73/100 - Loss: -1.312
Iter 74/100 - Loss: -1.365
Iter 75/100 - Loss: -1.418
Iter 76/100 - Loss: -1.484
Iter 77/100 - Loss: -1.564
Iter 78/100 - Loss: -1.599
Iter 79/100 - Loss: -1.678
Iter 80/100 - Loss: -1.731
Iter 81/100 - Loss: -1.793
Iter 82/100 - Loss: -1.790
Iter 83/100 - Loss: -1.801
Iter 84/100 - Loss: -1.832
Iter 85/100 - Loss: -1.916
Iter 86/100 - Loss: -1.974
Iter 87/100 - Loss: -2.030
Iter 88/100 - Loss: -2.108
Iter 89/100 - Loss: -2.166
Iter 90/100 - Loss: -2.152
Iter 91/100 - Loss: -2.119
Iter 92/100 - Loss: -2.088
Iter 93/100 - Loss: -2.101
Iter 94/100 - Loss: -2.174
Iter 95/100 - Loss: -2.247
Iter 96/100 - Loss: -2.223
Iter 97/100 - Loss: -1.789
Iter 98/100 - Loss: -2.284
Iter 99/100 - Loss: -2.083
Iter 100/100 - Loss: -2.164
```

4.1.4 Make predictions, and plot the model fit

Now that we've learned good hyperparameters, it's time to use our model to make predictions. The spectral mixture kernel is especially good at extrapolation. To that end, we'll see how well the model extrapolates past the interval `[0, 1]`.

In the next cell, we plot the mean and confidence region of the Gaussian process model. The `confidence_region` method is a helper method that returns 2 standard deviations above and below the mean.

```
In [5]: # Test points every 0.1 between 0 and 5
        test_x = torch.linspace(0, 5, 51)

        # Get into evaluation (predictive posterior) mode
        model.eval()
        likelihood.eval()

        # The gpytorch.fast_pred_var flag activates LOVE (for fast variances)
        # See https://arxiv.org/abs/1803.06058
        with torch.no_grad(), gpytorch.fast_pred_var():
            # Make predictions
            observed_pred = likelihood(model(test_x))

            # Initialize plot
            f, ax = plt.subplots(1, 1, figsize=(4, 3))

            # Get upper and lower confidence bounds
            lower, upper = observed_pred.confidence_region()
            # Plot training data as black stars
            ax.plot(train_x.numpy(), train_y.numpy(), 'k*')
            # Plot predictive means as blue line
            ax.plot(test_x.numpy(), observed_pred.mean.numpy(), 'b')
            # Shade between the lower and upper confidence bounds
            ax.fill_between(test_x.numpy(), lower.numpy(), upper.numpy(), alpha=0.5)
            ax.set_ylim([-3, 3])
            ax.legend(['Observed Data', 'Mean', 'Confidence'])
```




```
In [ ]:
```

4.2 GP Regression (CUDA) with Fast Predictive Distributions (LOVE)

4.2.1 Overview

In this notebook, we demonstrate that LOVE (the method for fast variances and sampling introduced in this paper https://arxiv.org/abs/1803.06058 and the `fast_variances_ski_LOVE.ipynb`) can significantly reduce the cost of computing predictive distributions with exact GPs too. This can be especially useful in settings like small-scale Bayesian optimization, where predictions need to be made at enormous numbers of candidate points, but there aren't enough training examples to necessarily warrant the use of sparse GP methods.

In this notebook, we will train an exact GP on the `skillcraft`UCI dataset, and then compare the time required to make predictions with each model.

NOTE: The timing results reported in the paper compare the time required to compute (co)variances **only**. Because excluding the mean computations from the timing results requires hacking the internals of GPyTorch, the timing results presented in this notebook include the time required to compute predictive means, which are not accelerated by LOVE. Nevertheless, as we will see, LOVE achieves impressive speed-ups.

```
In [1]: import math
        import torch
        import gpytorch
        from matplotlib import pyplot as plt

        # Make plots inline
        %matplotlib inline
```

Loading Data

For this example notebook, we'll be using the `skillcraft` UCI dataset used in the paper. Running the next cell downloads a copy of the dataset that has already been scaled and normalized appropriately. For this notebook, we'll simply be splitting the data using the first 40% of the data as training and the last 60% as testing.

Note: Running the next cell will attempt to download a small dataset file to the current directory.

```
In [2]: import urllib.request
        import os.path
        from scipy.io import loadmat
        from math import floor

        if not os.path.isfile('skillcraft.mat'):
            print('Downloading \'skillcraft\' UCI dataset...')
            urllib.request.urlretrieve('https://drive.google.com/uc?export=download&id=1xQ1vgx_bOsLD(

        data = torch.Tensor(loadmat('skillcraft.mat')['data'])
        X = data[:, :-1]
        X = X - X.min(0)[0]
        X = 2 * (X / X.max(0)[0]) - 1
        y = data[:, -1]

        # Use the first 80% of the data for training, and the last 20% for testing.
        train_n = int(floor(0.4*len(X)))

        train_x = X[:train_n, :].contiguous().cuda()
        train_y = y[:train_n].contiguous().cuda()

        test_x = X[train_n:, :].contiguous().cuda()
        test_y = y[train_n:].contiguous().cuda()
```

```
Downloading 'skillcraft' UCI dataset...
```

4.2.2 Defining the GP Model

We now define the GP model. The model below is essentially the same as the model in the `simple_gp_regression` example, with an added log outputscale.

```
In [3]: class GPRegressionModel(gpytorch.models.ExactGP):
            def __init__(self, train_x, train_y, likelihood):
                super(GPRegressionModel, self).__init__(train_x, train_y, likelihood)
                self.mean_module = gpytorch.means.ConstantMean()
                self.covar_module = gpytorch.kernels.ScaleKernel(
                    gpytorch.kernels.RBFKernel()
                )

            def forward(self, x):
                mean_x = self.mean_module(x)
                covar_x = self.covar_module(x)
                return gpytorch.distributions.MultivariateNormal(mean_x, covar_x)

In [4]: likelihood = gpytorch.likelihoods.GaussianLikelihood().cuda()
        model = GPRegressionModel(train_x, train_y, likelihood).cuda()
```

4.2.3 Training the model

The cell below trains the GP model, finding optimal hyperparameters using Type-II MLE. We run 20 iterations of training using the `Adam` optimizer built in to PyTorch. With a decent GPU, this should only take a few seconds.

```
In [ ]: # Find optimal model hyperparameters
        model.train()
        likelihood.train()

        # Use the adam optimizer
        optimizer = torch.optim.Adam([
            {'params': model.parameters()},  # Includes GaussianLikelihood parameters
        ], lr=0.1)

        # "Loss" for GPs - the marginal log likelihood
        mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)

        training_iterations = 20
        def train():
            for i in range(training_iterations):
                optimizer.zero_grad()
                output = model(train_x)
                loss = -mll(output, train_y)
                loss.backward()
                print('Iter %d/%d - Loss: %.3f' % (i + 1,
                                                   training_iterations,
                                                   loss.item()))
                optimizer.step()

        %time train()
```

4.2.4 Make Predictions using Standard SKI Code

The next cell gets the predictive covariance for the test set (and also technically gets the predictive mean, stored in `preds.mean`) using the standard SKI testing code, with no acceleration or precomputation.

Note: Full predictive covariance matrices (and the computations needed to get them) can be quite memory intensive. Depending on the memory available on your GPU, you may need to reduce the size of the test set for the code below to run. If you run out of memory, try replacing `test_x` below with something like `test_x[:1000]` to use the first 1000 test points only, and then restart the notebook.

```
In [6]: import time

        # Set into eval mode
        model.eval()
        likelihood.eval()

        with torch.no_grad():
            start_time = time.time()
            preds = model(test_x)
            exact_covar = preds.covariance_matrix
            exact_covar_time = time.time() - start_time
```

```
In [7]: print('Time to compute exact mean + covariances: {:.2f}s'.format(exact_covar_time))
```

```
Time to compute exact mean + covariances: 0.13s
```

4.2.5 Clear Memory and any Precomputed Values

The next cell clears as much as possible to avoid influencing the timing results of the fast predictive variances code. Strictly speaking, the timing results above and the timing results to follow should be run in entirely separate notebooks. However, this will suffice for this simple example.

```
In [8]: # Clear as much 'stuff' as possible
        import gc
        gc.collect()
        torch.cuda.empty_cache()
        model.train()
        likelihood.train()
```

```
Out[8]: GaussianLikelihood()
```

4.2.6 Compute Predictions with LOVE, but Before Precomputation

Next we compute predictive covariances (and the predictive means) for LOVE, but starting from scratch. That is, we don't yet have access to the precomputed cache discussed in the paper. This should still be faster than the full covariance computation code above.

To use LOVE, use the context manager `with gpytorch.fast_pred_var():`

You can also set some of the LOVE settings with context managers as well. For example, `gpytorch.settings.max_root_decomposition_size(35)` affects the accuracy of the LOVE solves (larger is more accurate, but slower).

In this simple example, we allow a rank 10 root decomposition, although increasing this to rank 20-40 should not affect the timing results substantially.

```
In [9]: # Set into eval mode
        model.eval()
        likelihood.eval()
```

```
        # The
        with torch.no_grad(), gpytorch.beta_features.fast_pred_var(), gpytorch.settings.max_root_deco
            start_time = time.time()
            preds = model(test_x)
            fast_time_no_cache = time.time() - start_time
```

4.2.7 Compute Predictions with LOVE After Precomputation

The above cell additionally computed the caches required to get fast predictions. From this point onwards, unless we put the model back in training mode, predictions should be extremely fast. The cell below re-runs the above code, but takes full advantage of both the mean cache and the LOVE cache for variances.

```
In [10]: with torch.no_grad(), gpytorch.beta_features.fast_pred_var(), gpytorch.settings.max_root_dec
             start_time = time.time()
             preds = model(test_x)
             fast_covar = preds.covariance_matrix
             fast_time_with_cache = time.time() - start_time
```

```
In [11]: print('Time to compute mean + covariances (no cache) {:.2f}s'.format(fast_time_no_cache))
         print('Time to compute mean + variances (cache): {:.2f}s'.format(fast_time_with_cache))
```

```
Time to compute mean + covariances (no cache) 0.06s
Time to compute mean + variances (cache): 0.02s
```

4.2.8 Compute Error between Exact and Fast Variances

Finally, we compute the mean absolute error between the fast variances computed by LOVE (stored in fast_covar), and the exact variances computed previously.

Note that these tests were run with a root decomposition of rank 10, which is about the minimum you would realistically ever run with. Despite this, the fast variance estimates are quite good. If more accuracy was needed, increasing `max_root_decomposition_size` to 30 or 40 would provide even better estimates.

```
In [12]: print('MAE between exact covar matrix and fast covar matrix: {}'.format((exact_covar - fast_
```

```
MAE between exact covar matrix and fast covar matrix: 3.323644705233164e-05
```

CHAPTER 5

Simple GP Classification

CHAPTER 6

Multitask GP Regression

6.1 Multitask GP Regression

6.1.1 Introduction

This notebook demonstrates how to perform standard (Kronecker) multitask regression.

This differs from the *hadamard multitask example* in one key way: - Here, we assume that we want to learn **all tasks per input**. (The kernel that we learn is expressed as a Kronecker product of an input kernel and a task kernel). - In the other notebook, we assume that we want to learn one tasks per input. For each input, we specify the task of the input that we care about. (The kernel in that notebook is the Hadamard product of an input kernel and a task kernel).

Multitask regression, first introduced in this paper learns similarities in the outputs simultaneously. It's useful when you are performing regression on multiple functions that share the same inputs, especially if they have similarities (such as being sinusodial).

Given inputs x and x', and tasks i and j, the covariance between two datapoints and two tasks is given by

$$k([x,i],[x',j]) = k_{\text{inputs}}(x,x') * k_{\text{tasks}}(i,j)$$

where k_{inputs} is a standard kernel (e.g. RBF) that operates on the inputs. k

```
In [5]: import math
        import torch
        import gpytorch
        from matplotlib import pyplot as plt

        %matplotlib inline
        %load_ext autoreload
        %autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

Set up training data

In the next cell, we set up the training data for this example. We'll be using 100 regularly spaced points on [0,1] which we evaluate the function on and add Gaussian noise to get the training labels.

We'll have two functions - a sine function (y1) and a cosine function (y2).

For MTGPs, our `train_targets` will actually have two dimensions: with the second dimension corresponding to the different tasks.

```
In [6]: train_x = torch.linspace(0, 1, 100)

        train_y = torch.stack([
            torch.sin(train_x * (2 * math.pi)) + torch.randn(train_x.size()) * 0.2,
            torch.cos(train_x * (2 * math.pi)) + torch.randn(train_x.size()) * 0.2,
        ], -1)
```

6.1.2 Set up the model

The model should be somewhat similar to the `ExactGP` model in the simple regression example.

The differences:

1. We're going to wrap ConstantMean with a `MultitaskMean`. This makes sure we have a mean function for each task.
2. Rather than just using a RBFKernel, we're using that in conjunction with a `MultitaskKernel`. This gives us the covariance function described in the introduction.
3. We're using a `MultitaskMultivariateNormal` and `MultitaskGaussianLikelihood`. This allows us to deal with the predictions/outputs in a nice way. For example, when we call MultitaskMultivariateNormal.mean, we get a `n x num_tasks` matrix back.

You may also notice that we don't use a ScaleKernel, since the IndexKernel will do some scaling for us. (This way we're not overparameterizing the kernel.)

```
In [9]: class MultitaskGPModel(gpytorch.models.ExactGP):
            def __init__(self, train_x, train_y, likelihood):
                super(MultitaskGPModel, self).__init__(train_x, train_y, likelihood)
                self.mean_module = gpytorch.means.MultitaskMean(
                    gpytorch.means.ConstantMean(), num_tasks=2
                )
                self.covar_module = gpytorch.kernels.MultitaskKernel(
                    gpytorch.kernels.RBFKernel(), num_tasks=2, rank=1
                )

            def forward(self, x):
                mean_x = self.mean_module(x)
                covar_x = self.covar_module(x)
                return gpytorch.distributions.MultitaskMultivariateNormal(mean_x, covar_x)


        likelihood = gpytorch.likelihoods.MultitaskGaussianLikelihood(num_tasks=2)
        model = MultitaskGPModel(train_x, train_y, likelihood)
```

6.1.3 Train the model hyperparameters

```
In [10]: # Find optimal model hyperparameters
         model.train()
```

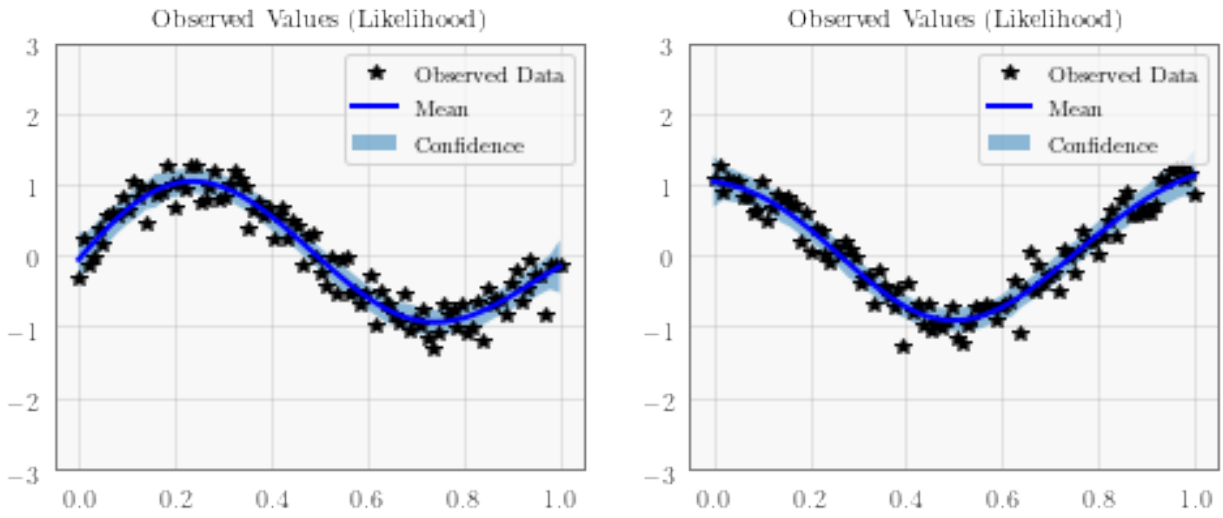
```
        likelihood.train()

        # Use the adam optimizer
        optimizer = torch.optim.Adam([
            {'params': model.parameters()},  # Includes GaussianLikelihood parameters
        ], lr=0.1)

        # "Loss" for GPs - the marginal log likelihood
        mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)

        n_iter = 50
        for i in range(n_iter):
            optimizer.zero_grad()
            output = model(train_x)
            loss = -mll(output, train_y)
            loss.backward()
            print('Iter %d/%d - Loss: %.3f' % (i + 1, n_iter, loss.item()))
            optimizer.step()
```

```
Iter 1/50 - Loss: 47.568
Iter 2/50 - Loss: 42.590
Iter 3/50 - Loss: 37.327
Iter 4/50 - Loss: 32.383
Iter 5/50 - Loss: 27.693
Iter 6/50 - Loss: 22.967
Iter 7/50 - Loss: 18.709
Iter 8/50 - Loss: 13.625
Iter 9/50 - Loss: 9.454
Iter 10/50 - Loss: 3.937
Iter 11/50 - Loss: -0.266
Iter 12/50 - Loss: -5.492
Iter 13/50 - Loss: -9.174
Iter 14/50 - Loss: -14.201
Iter 15/50 - Loss: -17.646
Iter 16/50 - Loss: -23.065
Iter 17/50 - Loss: -27.227
Iter 18/50 - Loss: -31.771
Iter 19/50 - Loss: -35.461
Iter 20/50 - Loss: -40.396
Iter 21/50 - Loss: -43.209
Iter 22/50 - Loss: -48.011
Iter 23/50 - Loss: -52.596
Iter 24/50 - Loss: -55.427
Iter 25/50 - Loss: -58.277
Iter 26/50 - Loss: -62.170
Iter 27/50 - Loss: -66.251
Iter 28/50 - Loss: -68.859
Iter 29/50 - Loss: -71.799
Iter 30/50 - Loss: -74.687
Iter 31/50 - Loss: -77.924
Iter 32/50 - Loss: -80.209
Iter 33/50 - Loss: -82.885
Iter 34/50 - Loss: -85.627
Iter 35/50 - Loss: -87.761
Iter 36/50 - Loss: -88.781
Iter 37/50 - Loss: -88.784
Iter 38/50 - Loss: -90.362
Iter 39/50 - Loss: -92.546
Iter 40/50 - Loss: -92.249
```

```
Iter 41/50 - Loss: -93.311
Iter 42/50 - Loss: -92.987
Iter 43/50 - Loss: -93.307
Iter 44/50 - Loss: -93.322
Iter 45/50 - Loss: -92.269
Iter 46/50 - Loss: -91.461
Iter 47/50 - Loss: -90.908
Iter 48/50 - Loss: -92.142
Iter 49/50 - Loss: -93.466
Iter 50/50 - Loss: -90.492
```

6.1.4 Make predictions with the model

```
In [13]: # Set into eval mode
         model.eval()
         likelihood.eval()

         # Initialize plots
         f, (y1_ax, y2_ax) = plt.subplots(1, 2, figsize=(8, 3))

         # Make predictions
         with torch.no_grad(), gpytorch.fast_pred_var():
             test_x = torch.linspace(0, 1, 51)
             predictions = likelihood(model(test_x))
             mean = predictions.mean
             lower, upper = predictions.confidence_region()

         # This contains predictions for both tasks, flattened out
         # The first half of the predictions is for the first task
         # The second half is for the second task

         # Plot training data as black stars
         y1_ax.plot(train_x.detach().numpy(), train_y[:, 0].detach().numpy(), 'k*')
         # Predictive mean as blue line
         y1_ax.plot(test_x.numpy(), mean[:, 0].numpy(), 'b')
         # Shade in confidence
         y1_ax.fill_between(test_x.numpy(), lower[:, 0].numpy(), upper[:, 0].numpy(), alpha=0.5)
         y1_ax.set_ylim([-3, 3])
         y1_ax.legend(['Observed Data', 'Mean', 'Confidence'])
         y1_ax.set_title('Observed Values (Likelihood)')

         # Plot training data as black stars
         y2_ax.plot(train_x.detach().numpy(), train_y[:, 1].detach().numpy(), 'k*')
         # Predictive mean as blue line
         y2_ax.plot(test_x.numpy(), mean[:, 1].numpy(), 'b')
         # Shade in confidence
         y2_ax.fill_between(test_x.numpy(), lower[:, 1].numpy(), upper[:, 1].numpy(), alpha=0.5)
         y2_ax.set_ylim([-3, 3])
         y2_ax.legend(['Observed Data', 'Mean', 'Confidence'])
         y2_ax.set_title('Observed Values (Likelihood)')

         None
```



In [ ]:

6.2 Scalable Multitask GP Regression (w/ KISS-GP)

This notebook demonstrates how to perform a scalable multitask regression.

It does everything that the *standard multitask GP example* does, but using the SKI scalable GP approximation. This can be used on much larger datasets (up to 100,000+ data points).

For more information on SKI, check out the scalable regression examples.

```
In [1]: import math
        import torch
        import gpytorch
        from matplotlib import pyplot as plt

        %matplotlib inline
        %load_ext autoreload
        %autoreload 2
```

6.2.1 Set up training data

In the next cell, we set up the training data for this example. We'll be using 1000 regularly spaced points on [0,1] which we evaluate the function on and add Gaussian noise to get the training labels.

We'll have two functions - a sine function (y1) and a cosine function (y2).

For MTGPs, our `train_targets` will actually have two dimensions: with the second dimension corresponding to the different tasks.

```
In [2]: train_x = torch.linspace(0, 1, 1000)

        train_y = torch.stack([
            torch.sin(train_x * (2 * math.pi)) + torch.randn(train_x.size()) * 0.2,
            torch.cos(train_x * (2 * math.pi)) + torch.randn(train_x.size()) * 0.2,
        ], -1)
```

Set up the model

The model should be somewhat similar to the `ExactGP` model in the simple regression example.

The differences:

1. We're going to wrap ConstantMean with a `MultitaskMean`. This makes sure we have a mean function for each task.
2. Rather than just using a RBFKernel, we're using that in conjunction with a `MultitaskKernel`. This gives us the covariance function described in the introduction.
3. We're using a `MultitaskMultivariateNormal` and `MultitaskGaussianLikelihood`. This allows us to deal with the predictions/outputs in a nice way. For example, when we call MultitaskMultivariateNormal.mean, we get a `n x num_tasks` matrix back.

In addition, we're going to wrap the RBFKernel in a `GridInterpolationKernel`. This approximates the kernel using SKI, which makes GP regression very scalable.

You may also notice that we don't use a ScaleKernel, since the IndexKernel will do some scaling for us. (This way we're not overparameterizing the kernel.)

```
In [3]: class MultitaskGPModel(gpytorch.models.ExactGP):
            def __init__(self, train_x, train_y, likelihood):
                super(MultitaskGPModel, self).__init__(train_x, train_y, likelihood)

                # SKI requires a grid size hyperparameter. This util can help with that
                grid_size = gpytorch.utils.grid.choose_grid_size(train_x)

                self.mean_module = gpytorch.means.MultitaskMean(
                    gpytorch.means.ConstantMean(), num_tasks=2
                )
                self.covar_module = gpytorch.kernels.MultitaskKernel(
                    gpytorch.kernels.GridInterpolationKernel(
                        gpytorch.kernels.RBFKernel(), grid_size=grid_size, num_dims=1,
                    ), num_tasks=2, rank=1
                )

            def forward(self, x):
                mean_x = self.mean_module(x)
                covar_x = self.covar_module(x)
                return gpytorch.distributions.MultitaskMultivariateNormal(mean_x, covar_x)


        likelihood = gpytorch.likelihoods.MultitaskGaussianLikelihood(num_tasks=2)
        model = MultitaskGPModel(train_x, train_y, likelihood)
```

Train the model hyperparameters

```
In [4]: # Find optimal model hyperparameters
        model.train()
        likelihood.train()

        # Use the adam optimizer
        optimizer = torch.optim.Adam([
            {'params': model.parameters()},  # Includes GaussianLikelihood parameters
        ], lr=0.1)

        # "Loss" for GPs - the marginal log likelihood
        mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)
```

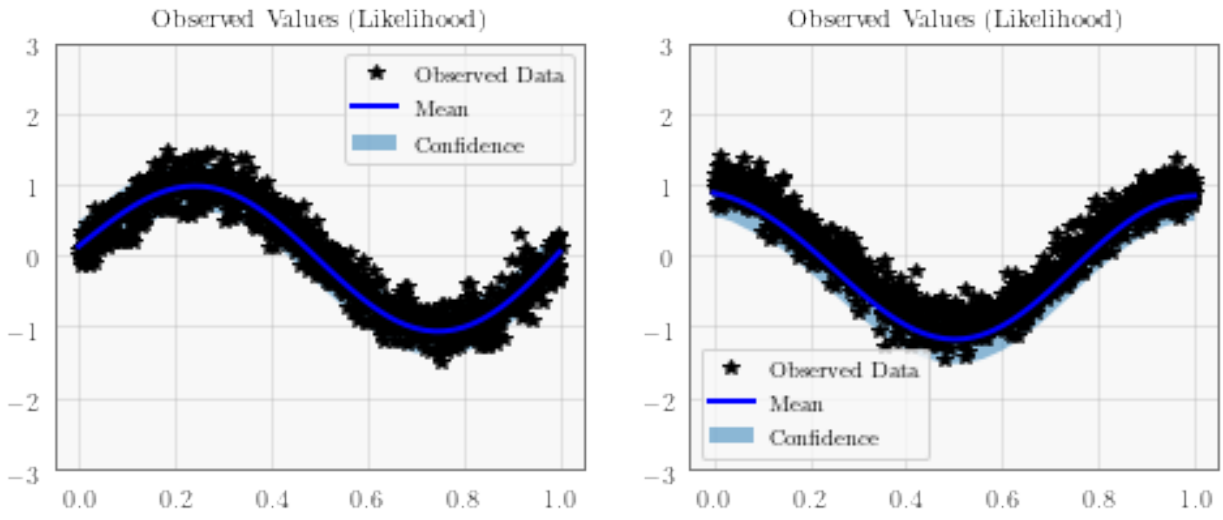
```
        n_iter = 50
        for i in range(n_iter):
            optimizer.zero_grad()
            output = model(train_x)
            loss = -mll(output, train_y)
            loss.backward()
            print('Iter %d/%d - Loss: %.3f' % (i + 1, n_iter, loss.item()))
            optimizer.step()
```

```
Iter 1/50 - Loss: 404.388
Iter 2/50 - Loss: 352.338
Iter 3/50 - Loss: 300.768
Iter 4/50 - Loss: 249.549
Iter 5/50 - Loss: 198.132
Iter 6/50 - Loss: 145.569
Iter 7/50 - Loss: 91.422
Iter 8/50 - Loss: 38.138
Iter 9/50 - Loss: -11.845
Iter 10/50 - Loss: -61.598
Iter 11/50 - Loss: -110.397
Iter 12/50 - Loss: -157.206
Iter 13/50 - Loss: -203.842
Iter 14/50 - Loss: -249.601
Iter 15/50 - Loss: -292.015
Iter 16/50 - Loss: -340.180
Iter 17/50 - Loss: -384.880
Iter 18/50 - Loss: -427.049
Iter 19/50 - Loss: -471.487
Iter 20/50 - Loss: -516.107
Iter 21/50 - Loss: -553.546
Iter 22/50 - Loss: -600.415
Iter 23/50 - Loss: -634.620
Iter 24/50 - Loss: -676.436
Iter 25/50 - Loss: -715.459
Iter 26/50 - Loss: -754.357
Iter 27/50 - Loss: -792.451
Iter 28/50 - Loss: -826.312
Iter 29/50 - Loss: -854.712
Iter 30/50 - Loss: -887.396
Iter 31/50 - Loss: -918.314
Iter 32/50 - Loss: -945.657
Iter 33/50 - Loss: -971.053
Iter 34/50 - Loss: -992.324
Iter 35/50 - Loss: -1014.124
Iter 36/50 - Loss: -1035.698
Iter 37/50 - Loss: -1052.513
Iter 38/50 - Loss: -1065.371
Iter 39/50 - Loss: -1074.025
Iter 40/50 - Loss: -1083.747
Iter 41/50 - Loss: -1090.505
Iter 42/50 - Loss: -1092.604
Iter 43/50 - Loss: -1097.734
Iter 44/50 - Loss: -1096.438
Iter 45/50 - Loss: -1098.831
Iter 46/50 - Loss: -1098.317
Iter 47/50 - Loss: -1094.847
Iter 48/50 - Loss: -1092.078
Iter 49/50 - Loss: -1088.565
```

```
Iter 50/50 - Loss: -1082.591
```

Make predictions with the model

```
In [5]: # Set into eval mode
        model.eval()
        likelihood.eval()

        # Initialize plots
        f, (y1_ax, y2_ax) = plt.subplots(1, 2, figsize=(8, 3))
        # Test points every 0.02 in [0,1]

        # Make predictions
        with torch.no_grad(), gpytorch.fast_pred_var():
            test_x = torch.linspace(0, 1, 51)
            observed_pred = likelihood(model(test_x))
            # Get mean
            mean = observed_pred.mean
            # Get lower and upper confidence bounds
            lower, upper = observed_pred.confidence_region()

        # This contains predictions for both tasks, flattened out
        # The first half of the predictions is for the first task
        # The second half is for the second task

        # Plot training data as black stars
        y1_ax.plot(train_x.detach().numpy(), train_y[:, 0].detach().numpy(), 'k*')
        # Predictive mean as blue line
        y1_ax.plot(test_x.numpy(), mean[:, 0].numpy(), 'b')
        # Shade in confidence
        y1_ax.fill_between(test_x.numpy(), lower[:, 0].numpy(), upper[:, 0].numpy(), alpha=0.5)
        y1_ax.set_ylim([-3, 3])
        y1_ax.legend(['Observed Data', 'Mean', 'Confidence'])
        y1_ax.set_title('Observed Values (Likelihood)')

        # Plot training data as black stars
        y2_ax.plot(train_x.detach().numpy(), train_y[:, 1].detach().numpy(), 'k*')
        # Predictive mean as blue line
        y2_ax.plot(test_x.numpy(), mean[:, 1].numpy(), 'b')
        # Shade in confidence
        y2_ax.fill_between(test_x.numpy(), lower[:, 1].numpy(), upper[:, 1].numpy(), alpha=0.5)
        y2_ax.set_ylim([-3, 3])
        y2_ax.legend(['Observed Data', 'Mean', 'Confidence'])
        y2_ax.set_title('Observed Values (Likelihood)')

        None
```



In [ ]:

6.3 Hadamard Multitask GP Regression

6.3.1 Introduction

This notebook demonstrates how to perform "Hadamard" multitask regression with kernels.IndexKernel.

This differs from the *multitask gp regression example notebook* in one key way: - Here, we assume that we want to learn **one task per input**. For each input, we specify the task of the input that we care about. (The kernel that we learn is expressed as a Hadamard product of an input kernel and a task kernel) - In the other notebook, we assume that we want to learn all tasks per input. (The kernel in that notebook is the Kronecker product of an input kernel and a task kernel).

Multitask regression, first introduced in this paper learns similarities in the outputs simultaneously. It's useful when you are performing regression on multiple functions that share the same inputs, especially if they have similarities (such as being sinusodial).

Given inputs x and x', and tasks i and j, the covariance between two datapoints and two tasks is given by

$$k([x,i],[x',j]) = k_{\text{inputs}}(x,x') * k_{\text{tasks}}(i,j)$$

where k_{inputs} is a standard kernel (e.g. RBF) that operates on the inputs. k

```
In [1]: import math
        import torch
        import gpytorch
        from matplotlib import pyplot as plt

        %matplotlib inline
        %load_ext autoreload
        %autoreload 2
```

Set up training data

In the next cell, we set up the training data for this example. We'll be using 15 regularly spaced points on [0,1] which we evaluate the function on and add Gaussian noise to get the training labels.

We'll have two functions - a sine function (y1) and a cosine function (y2)

```
In [2]: train_x = torch.linspace(0, 1, 100)

        train_y1 = torch.sin(train_x * (2 * math.pi)) + torch.randn(train_x.size()) * 0.2
        train_y2 = torch.cos(train_x * (2 * math.pi)) + torch.randn(train_x.size()) * 0.2
```

6.3.2 Set up the model

The model should be somewhat similar to the `ExactGP` model in the simple regression example.

The differences:

1. The model takes two input: the inputs (x) and indices. The indices indicate which task we want an output for,
2. Rather than just using a RBFKernel, we're using that in conjunction with a IndexKernel
3. We don't use a ScaleKernel, since the IndexKernel will do some scaling for us. (This way we're not overparameterizing the kernel.)

```
In [3]: class MultitaskGPModel(gpytorch.models.ExactGP):
            def __init__(self, train_x, train_y, likelihood):
                super(MultitaskGPModel, self).__init__(train_x, train_y, likelihood)
                self.mean_module = gpytorch.means.ConstantMean()
                self.covar_module = gpytorch.kernels.RBFKernel()

                # We learn an IndexKernel for 2 tasks
                # (so we'll actually learn 2x2=4 tasks with correlations)
                self.task_covar_module = gpytorch.kernels.IndexKernel(num_tasks=2, rank=1)

            def forward(self,x,i):
                mean_x = self.mean_module(x)

                # Get input-input covariance
                covar_x = self.covar_module(x)
                # Get task-task covariance
                covar_i = self.task_covar_module(i)
                # Multiply the two together to get the covariance we want
                covar = covar_x.mul(covar_i)

                return gpytorch.distributions.MultivariateNormal(mean_x, covar)

        likelihood = gpytorch.likelihoods.GaussianLikelihood()

        # Here we want outputs for every input and task
        # This is not the most efficient model for this: it's better to use the model in the ./Multit
        # Since we are learning two tasks we feed in the x_data twice, along with the
        # y_data along with its indices
        train_i_task1 = torch.full_like(train_x, dtype=torch.long, fill_value=0)
        train_i_task2 = torch.full_like(train_x, dtype=torch.long, fill_value=1)

        full_train_x = torch.cat([train_x, train_x])
        full_train_i = torch.cat([train_i_task1, train_i_task2])
        full_train_y = torch.cat([train_y1, train_y2])

        # Here we have two iterms that we're passing in as train_inputs
        model = MultitaskGPModel((full_train_x, full_train_i), full_train_y, likelihood)
```

6.3.3 Training the model

In the next cell, we handle using Type-II MLE to train the hyperparameters of the Gaussian process. The spectral mixture kernel's hyperparameters start from what was specified in `initialize_from_data`.

See the simple regression example for more info on this step.

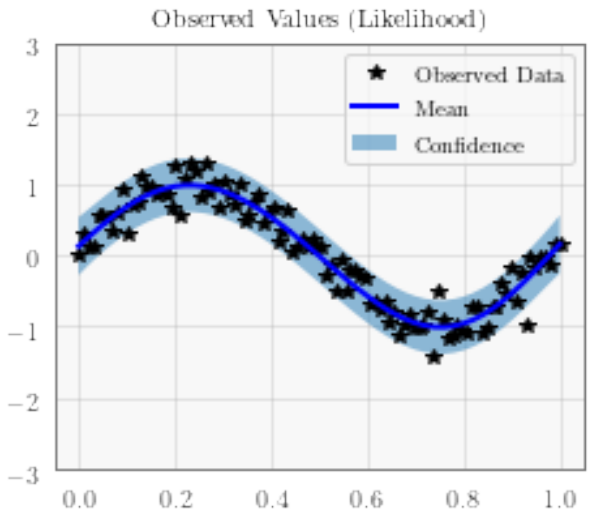
```
In [4]: # Find optimal model hyperparameters
        model.train()
        likelihood.train()

        # Use the adam optimizer
        optimizer = torch.optim.Adam([
            {'params': model.parameters()},  # Includes GaussianLikelihood parameters
        ], lr=0.1)

        # "Loss" for GPs - the marginal log likelihood
        mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)

        for i in range(50):
            optimizer.zero_grad()
            output = model(full_train_x, full_train_i)
            loss = -mll(output, full_train_y)
            loss.backward()
            print('Iter %d/50 - Loss: %.3f' % (i + 1, loss.item()))
            optimizer.step()
```

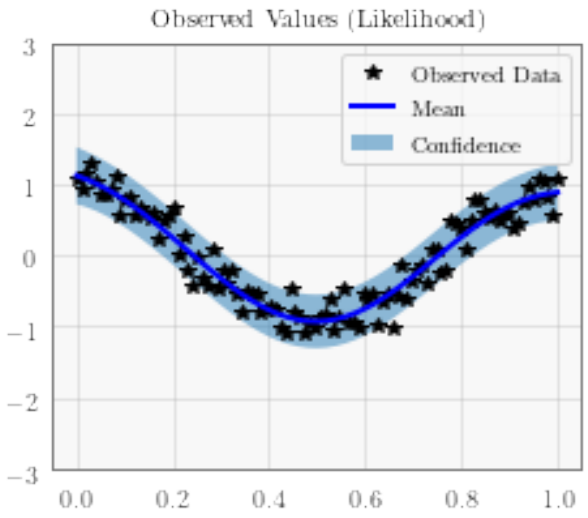
```
Iter 1/50 - Loss: 1.119
Iter 2/50 - Loss: 1.071
Iter 3/50 - Loss: 1.019
Iter 4/50 - Loss: 0.961
Iter 5/50 - Loss: 0.905
Iter 6/50 - Loss: 0.844
Iter 7/50 - Loss: 0.788
Iter 8/50 - Loss: 0.731
Iter 9/50 - Loss: 0.670
Iter 10/50 - Loss: 0.626
Iter 11/50 - Loss: 0.580
Iter 12/50 - Loss: 0.546
Iter 13/50 - Loss: 0.491
Iter 14/50 - Loss: 0.471
Iter 15/50 - Loss: 0.433
Iter 16/50 - Loss: 0.373
Iter 17/50 - Loss: 0.357
Iter 18/50 - Loss: 0.319
Iter 19/50 - Loss: 0.288
Iter 20/50 - Loss: 0.251
Iter 21/50 - Loss: 0.217
Iter 22/50 - Loss: 0.166
Iter 23/50 - Loss: 0.150
Iter 24/50 - Loss: 0.114
Iter 25/50 - Loss: 0.095
Iter 26/50 - Loss: 0.075
Iter 27/50 - Loss: 0.046
Iter 28/50 - Loss: 0.016
Iter 29/50 - Loss: -0.003
Iter 30/50 - Loss: -0.018
Iter 31/50 - Loss: -0.023
Iter 32/50 - Loss: -0.019
Iter 33/50 - Loss: -0.022
```

```
Iter 34/50 - Loss: -0.024
Iter 35/50 - Loss: -0.048
Iter 36/50 - Loss: -0.043
Iter 37/50 - Loss: -0.040
Iter 38/50 - Loss: -0.027
Iter 39/50 - Loss: -0.027
Iter 40/50 - Loss: -0.029
Iter 41/50 - Loss: -0.013
Iter 42/50 - Loss: -0.014
Iter 43/50 - Loss: -0.031
Iter 44/50 - Loss: 0.015
Iter 45/50 - Loss: -0.014
Iter 46/50 - Loss: -0.031
Iter 47/50 - Loss: -0.035
Iter 48/50 - Loss: -0.021
Iter 49/50 - Loss: -0.025
Iter 50/50 - Loss: -0.030
```

6.3.4 Make predictions with the model

```
In [5]: # Set into eval mode
        model.eval()
        likelihood.eval()

        # Initialize plots
        f, (y1_ax, y2_ax) = plt.subplots(1, 2, figsize=(8, 3))

        # Test points every 0.02 in [0,1]
        test_x = torch.linspace(0, 1, 51)
        tast_i_task1 = torch.full_like(test_x, dtype=torch.long, fill_value=0)
        test_i_task2 = torch.full_like(test_x, dtype=torch.long, fill_value=1)

        # Make predictions - one task at a time
        # We control the task we cae about using the indices

        # The gpytorch.fast_pred_var flag activates LOVE (for fast variances)
        # See https://arxiv.org/abs/1803.06058
        with torch.no_grad(), gpytorch.fast_pred_var():
            observed_pred_y1 = likelihood(model(test_x, tast_i_task1))
            observed_pred_y2 = likelihood(model(test_x, test_i_task2))

        # Define plotting function
        def ax_plot(ax, train_y, rand_var, title):
            # Get lower and upper confidence bounds
            lower, upper = rand_var.confidence_region()
            # Plot training data as black stars
            ax.plot(train_x.detach().numpy(), train_y.detach().numpy(), 'k*')
            # Predictive mean as blue line
            ax.plot(test_x.detach().numpy(), rand_var.mean.detach().numpy(), 'b')
            # Shade in confidence
            ax.fill_between(test_x.detach().numpy(), lower.detach().numpy(), upper.detach().numpy(),
            ax.set_ylim([-3, 3])
            ax.legend(['Observed Data', 'Mean', 'Confidence'])
            ax.set_title(title)

        # Plot both tasks
        ax_plot(y1_ax, train_y1, observed_pred_y1, 'Observed Values (Likelihood)')
        ax_plot(y2_ax, train_y2, observed_pred_y2, 'Observed Values (Likelihood)')
```


Observed Values (Likelihood)
Observed Data
Mean
Confidence
3
2
1
0
−1
−2
−3
0.0
0.2
0.4
0.6
0.8
1.0


Observed Values (Likelihood)
Observed Data
Mean
Confidence
3
2
1
0
−1
−2
−3
0.0
0.2
0.4
0.6
0.8
1.0

In [ ]:

CHAPTER 7

Scalable GP Regression (1D)

7.1 Scalable GP Regression in 1D (w/ KISS-GP)

7.1.1 Introduction

For 1D functions, SKI (or KISS-GP) is a great way to scale a GP up to very large datasets (100,000+ data points). Kernel interpolation for scalable structured Gaussian processes (KISS-GP) was introduced in this paper: http://proceedings.mlr.press/v37/wilson15.pdf

SKI is asymptotically very fast (nearly linear), very precise (error decays cubically), and easy to use in GPyTorch! As you will see in this tutorial, it's really easy to apply SKI to an existing model. All you have to do is wrap your kernel module with a `GridInterpolationKernel`.

```
In [1]: import math
        import torch
        import gpytorch
        from matplotlib import pyplot as plt

        # Make plots inline
        %matplotlib inline
```

Set up training data

We'll learn a simple sinusoid, but with lots of training data points. At 1000 points, this is where scalable methods start to become useful.

```
In [2]: train_x = torch.linspace(0, 1, 1000)
        train_y = torch.sin(train_x * (4 * math.pi) + torch.randn(train_x.size()) * 0.2)
```

7.1.2 Set up the model

The model should be somewhat similar to the `ExactGP` model in the simple regression example.

The only difference: we're wrapping our kernel module in a `GridInterpolationKernel`. This signals to GPyTorch that you want to approximate this kernel matrix with SKI.

SKI has only one hyperparameter that you need to worry about: the grid size. For 1D functions, a good starting place is to use as many grid points as training points. (Don't worry - the grid points are really cheap to use!). You can use the `gpytorch.utils.grid.choose_grid_size` helper to get a good starting point.

If you want, you can also explicitly determine the grid bounds of the SKI approximation using the `grid_bounds` argument. However, it's easier if you don't use this argument - then GPyTorch automatically chooses the best bounds for you.

```
In [3]: class GPRegressionModel(gpytorch.models.ExactGP):
            def __init__(self, train_x, train_y, likelihood):
                super(GPRegressionModel, self).__init__(train_x, train_y, likelihood)

                # SKI requires a grid size hyperparameter. This util can help with that
                grid_size = gpytorch.utils.grid.choose_grid_size(train_x)

                self.mean_module = gpytorch.means.ConstantMean()
                self.covar_module = gpytorch.kernels.GridInterpolationKernel(
                    gpytorch.kernels.ScaleKernel(gpytorch.kernels.RBFKernel()),
                    grid_size=grid_size, num_dims=1,
                )

            def forward(self, x):
                mean_x = self.mean_module(x)
                covar_x = self.covar_module(x)
                return gpytorch.distributions.MultivariateNormal(mean_x, covar_x)


        likelihood = gpytorch.likelihoods.GaussianLikelihood()
        model = GPRegressionModel(train_x, train_y, likelihood)
```

7.1.3 Train the model hyperparameters

Even with 1000 points, this model still trains fast! SKI scales (essentially) linearly with data - whereas standard GP inference scales quadratically (in GPyTorch.)

```
In [4]: # Find optimal model hyperparameters
        model.train()
        likelihood.train()

        # Use the adam optimizer
        optimizer = torch.optim.Adam([
            {'params': model.parameters()},  # Includes GaussianLikelihood parameters
        ], lr=0.1)

        # "Loss" for GPs - the marginal log likelihood
        mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)

        training_iterations = 30
        for i in range(training_iterations):
            optimizer.zero_grad()
            output = model(train_x)
            loss = -mll(output, train_y)
            loss.backward()
            print('Iter %d/%d - Loss: %.3f' % (i + 1, training_iterations, loss.item()))
            optimizer.step()
```

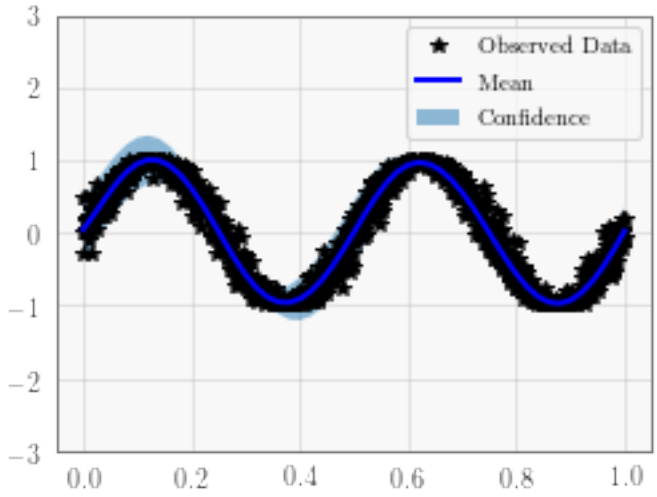
```
Iter 1/30 - Loss: 1.142
Iter 2/30 - Loss: 1.113
Iter 3/30 - Loss: 1.085
Iter 4/30 - Loss: 1.055
Iter 5/30 - Loss: 1.024
Iter 6/30 - Loss: 0.991
Iter 7/30 - Loss: 0.958
Iter 8/30 - Loss: 0.925
Iter 9/30 - Loss: 0.888
Iter 10/30 - Loss: 0.835
Iter 11/30 - Loss: 0.753
Iter 12/30 - Loss: 0.639
Iter 13/30 - Loss: 0.513
Iter 14/30 - Loss: 0.404
Iter 15/30 - Loss: 0.320
Iter 16/30 - Loss: 0.257
Iter 17/30 - Loss: 0.202
Iter 18/30 - Loss: 0.153
Iter 19/30 - Loss: 0.106
Iter 20/30 - Loss: 0.061
Iter 21/30 - Loss: 0.015
Iter 22/30 - Loss: -0.036
Iter 23/30 - Loss: -0.077
Iter 24/30 - Loss: -0.123
Iter 25/30 - Loss: -0.164
Iter 26/30 - Loss: -0.203
Iter 27/30 - Loss: -0.245
Iter 28/30 - Loss: -0.281
Iter 29/30 - Loss: -0.310
Iter 30/30 - Loss: -0.352
```

7.2 Make predictions

SKI is especially well-suited for predictions. It can comnpute predictive means in constant time, and with LOVE enabled (see *this notebook*), predictive variances are also constant time.

```
In [5]: # Put model & likelihood into eval mode
        model.eval()
        likelihood.eval()

        # Initalize plot
        f, ax = plt.subplots(1, 1, figsize=(4, 3))

        # The gpytorch.fast_pred_var flag activates LOVE (for fast variances)
        # See https://arxiv.org/abs/1803.06058
        with torch.no_grad(), gpytorch.fast_pred_var():
            test_x = torch.linspace(0, 1, 51)
            prediction = likelihood(model(test_x))
            mean = prediction.mean
            # Get lower and upper predictive bounds
            lower, upper = prediction.confidence_region()

        # Plot the training data as black stars
        ax.plot(train_x.detach().numpy(), train_y.detach().numpy(), 'k*')
        # Plot predictive means as blue line
        ax.plot(test_x.detach().numpy(), mean.detach().numpy(), 'b')
        # Plot confidence bounds as lightly shaded region
```

```
        ax.fill_between(test_x.detach().numpy(), lower.detach().numpy(), upper.detach().numpy(), alph
        ax.set_ylim([-3, 3])
        ax.legend(['Observed Data', 'Mean', 'Confidence'])

        None
```



In [ ]:

7.3 GPU-accelerated Scalable GP Regression in 1D (w/ KISS-GP)

7.3.1 Introduction

For 1D functions, SKI (or KISS-GP) is a great way to scale a GP up to very large datasets (100,000+ data points). Kernel interpolation for scalable structured Gaussian processes (KISS-GP) was introduced in this paper: http://proceedings.mlr.press/v37/wilson15.pdf

SKI is asymptotically very fast (nearly linear), very precise (error decays cubically), and easy to use in GPyTorch! As you will see in this tutorial, it's really easy to apply SKI to an existing model. All you have to do is wrap your kernel module with a `GridInterpolationKernel`.

This is the same as *the standard KISSGP 1D notebook*, but where everything is super-charged with CUDA. SKI is especially fast on the GPU.

The only thing required for GPU acceleration: `train_x.cuda()`, `train_y.cuda()`, and `model.cuda()`.

In [1]:
```
import math
import torch
import gpytorch
from matplotlib import pyplot as plt

# Make plots inline
%matplotlib inline
```

In [2]:
```
train_x = torch.linspace(0, 1, 1000).cuda()
train_y = torch.sin(train_x * (4 * math.pi) + torch.randn(train_x.size()).cuda() * 0.2)
```

In [3]:
```
class GPRegressionModel(gpytorch.models.ExactGP):
    def __init__(self, train_x, train_y, likelihood):
        super(GPRegressionModel, self).__init__(train_x, train_y, likelihood)
```

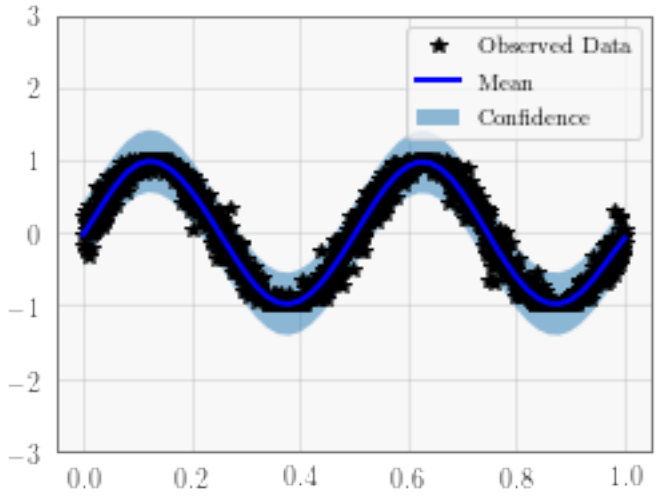
```
                # SKI requires a grid size hyperparameter. This util can help with that
                grid_size = gpytorch.utils.grid.choose_grid_size(train_x)

                self.mean_module = gpytorch.means.ConstantMean()
                self.covar_module = gpytorch.kernels.GridInterpolationKernel(
                    gpytorch.kernels.ScaleKernel(gpytorch.kernels.RBFKernel()),
                    grid_size=grid_size, num_dims=1,
                )

            def forward(self, x):
                mean_x = self.mean_module(x)
                covar_x = self.covar_module(x)
                return gpytorch.distributions.MultivariateNormal(mean_x, covar_x)


        likelihood = gpytorch.likelihoods.GaussianLikelihood()
        model = GPRegressionModel(train_x, train_y, likelihood).cuda()
```

In [4]:
```
        # Find optimal model hyperparameters
        model.train()
        likelihood.train()

        # Use the adam optimizer
        optimizer = torch.optim.Adam([
            {'params': model.parameters()},  # Includes GaussianLikelihood parameters
        ], lr=0.1)

        # "Loss" for GPs - the marginal log likelihood
        mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)

        def train():
            training_iterations = 30
            for i in range(training_iterations):
                optimizer.zero_grad()
                output = model(train_x)
                loss = -mll(output, train_y)
                loss.backward()
                print('Iter %d/%d - Loss: %.3f' % (i + 1, training_iterations, loss.item()))
                optimizer.step()

        %time train()
```

```
Iter 1/30 - Loss: 1.137
Iter 2/30 - Loss: 1.109
Iter 3/30 - Loss: 1.080
Iter 4/30 - Loss: 1.052
Iter 5/30 - Loss: 1.023
Iter 6/30 - Loss: 0.994
Iter 7/30 - Loss: 0.963
Iter 8/30 - Loss: 0.933
Iter 9/30 - Loss: 0.900
Iter 10/30 - Loss: 0.860
Iter 11/30 - Loss: 0.804
Iter 12/30 - Loss: 0.721
Iter 13/30 - Loss: 0.612
Iter 14/30 - Loss: 0.492
Iter 15/30 - Loss: 0.382
Iter 16/30 - Loss: 0.296
Iter 17/30 - Loss: 0.230
Iter 18/30 - Loss: 0.172
```

```
Iter 19/30 - Loss: 0.122
Iter 20/30 - Loss: 0.074
Iter 21/30 - Loss: 0.027
Iter 22/30 - Loss: -0.021
Iter 23/30 - Loss: -0.063
Iter 24/30 - Loss: -0.108
Iter 25/30 - Loss: -0.145
Iter 26/30 - Loss: -0.187
Iter 27/30 - Loss: -0.224
Iter 28/30 - Loss: -0.263
Iter 29/30 - Loss: -0.300
Iter 30/30 - Loss: -0.331
CPU times: user 2.43 s, sys: 100 ms, total: 2.53 s
Wall time: 2.52 s
```

```
In [5]: # Put model & likelihood into eval mode
        model.eval()
        likelihood.eval()

        # Initalize plot
        f, ax = plt.subplots(1, 1, figsize=(4, 3))

        # The gpytorch.fast_pred_var flag activates LOVE (for fast variances)
        # See https://arxiv.org/abs/1803.06058
        with torch.no_grad(), gpytorch.fast_pred_var():
            test_x = torch.linspace(0, 1, 51).cuda()
            prediction = likelihood(model(test_x))
            mean = prediction.mean
            # Get lower and upper predictive bounds
            lower, upper = prediction.confidence_region()

        # Plot the training data as black stars
        ax.plot(train_x.detach().cpu().numpy(), train_y.detach().cpu().numpy(), 'k*')
        # Plot predictive means as blue line
        ax.plot(test_x.detach().cpu().numpy(), mean.detach().cpu().numpy(), 'b')
        # Plot confidence bounds as lightly shaded region
        ax.fill_between(test_x.detach().cpu().numpy(), lower.detach().cpu().numpy(), upper.detach().c
        ax.set_ylim([-3, 3])
        ax.legend(['Observed Data', 'Mean', 'Confidence'])

        None
```



```
In [ ]:
```

CHAPTER 8

Scalable GP Regression (Multidimensional)

8.1 Scalable GP Regression (w/ KISS-GP)

8.1.1 Introduction

For 2-4D functions, SKI (or KISS-GP) can work very well out-of-the-box on larger datasets (100,000+ data points). Kernel interpolation for scalable structured Gaussian processes (KISS-GP) was introduced in this paper: http://proceedings.mlr.press/v37/wilson15.pdf

One thing to watch out for with multidimensional SKI - you can't use as fine-grain of a grid. If you have a high dimensional problem, you may want to try one of the other scalable regression methods.

This is the same as the standard KISSGP 1D notebook, but applied to more dimensions.

```
In [1]: import math
        import torch
        import gpytorch
        from matplotlib import pyplot as plt

        %matplotlib inline
```

Set up train data

Here we're learning a simple sin function - but in 2 dimensions

```
In [2]: # We make an nxn grid of training points spaced every 1/(n-1) on [0,1]x[0,1]
        n = 40
        train_x = torch.zeros(pow(n, 2), 2)
        for i in range(n):
            for j in range(n):
                train_x[i * n + j][0] = float(i) / (n-1)
                train_x[i * n + j][1] = float(j) / (n-1)
        # True function is sin( 2*pi*(x0+x1))
        train_y = torch.sin((train_x[:, 0] + train_x[:, 1]) * (2 * math.pi)) + torch.randn_like(trai
```

8.1.2 The model

As with the 1D case, applying SKI to a multidimensional kernel is as simple as wrapping that kernel with a `GridInterpolationKernel`. You'll want to be sure to set `num_dims` though!

SKI has only one hyperparameter that you need to worry about: the grid size. For 1D functions, a good starting place is to use as many grid points as training points. (Don't worry - the grid points are really cheap to use!). You can use the `gpytorch.utils.grid.choose_grid_size` helper to get a good starting point.

If you want, you can also explicitly determine the grid bounds of the SKI approximation using the `grid_bounds` argument. However, it's easier if you don't use this argument - then GPyTorch automatically chooses the best bounds for you.

```
In [3]: class GPRegressionModel(gpytorch.models.ExactGP):
            def __init__(self, train_x, train_y, likelihood):
                super(GPRegressionModel, self).__init__(train_x, train_y, likelihood)

                # SKI requires a grid size hyperparameter. This util can help with that
                grid_size = gpytorch.utils.grid.choose_grid_size(train_x)

                self.mean_module = gpytorch.means.ConstantMean()
                self.covar_module = gpytorch.kernels.GridInterpolationKernel(
                    gpytorch.kernels.ScaleKernel(
                        gpytorch.kernels.RBFKernel(ard_num_dims=2),
                    ), grid_size=grid_size, num_dims=2
                )

            def forward(self, x):
                mean_x = self.mean_module(x)
                covar_x = self.covar_module(x)
                return gpytorch.distributions.MultivariateNormal(mean_x, covar_x)


        likelihood = gpytorch.likelihoods.GaussianLikelihood()
        model = GPRegressionModel(train_x, train_y, likelihood)
```

8.1.3 Train the model hyperparameters

```
In [4]: # Find optimal model hyperparameters
        model.train()
        likelihood.train()

        # Use the adam optimizer
        optimizer = torch.optim.Adam([
            {'params': model.parameters()},  # Includes GaussianLikelihood parameters
        ], lr=0.1)

        # "Loss" for GPs - the marginal log likelihood
        mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)

        def train():
            training_iterations = 30
            for i in range(training_iterations):
                optimizer.zero_grad()
                output = model(train_x)
                loss = -mll(output, train_y)
                loss.backward()
                print('Iter %d/%d - Loss: %.3f' % (i + 1, training_iterations, loss.item()))
```

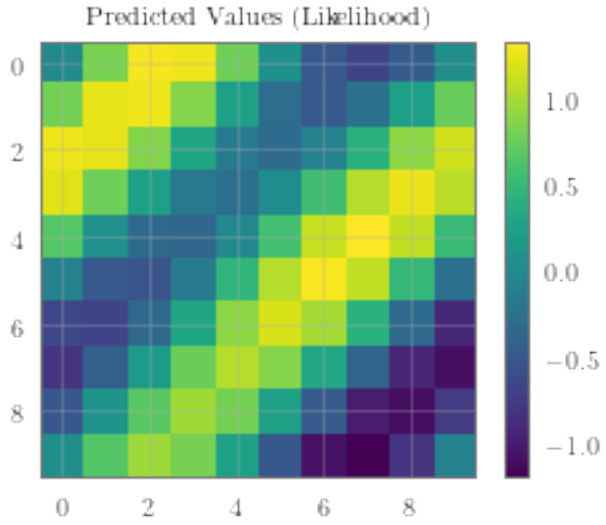
```
                optimizer.step()

        %time train()
```

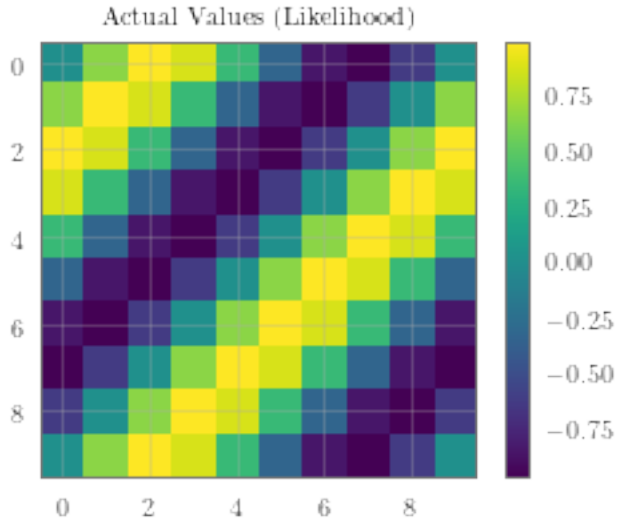
```
Iter 1/30 - Loss: 1.142
Iter 2/30 - Loss: 1.087
Iter 3/30 - Loss: 1.025
Iter 4/30 - Loss: 0.960
Iter 5/30 - Loss: 0.901
Iter 6/30 - Loss: 0.843
Iter 7/30 - Loss: 0.774
Iter 8/30 - Loss: 0.686
Iter 9/30 - Loss: 0.595
Iter 10/30 - Loss: 0.521
Iter 11/30 - Loss: 0.465
Iter 12/30 - Loss: 0.412
Iter 13/30 - Loss: 0.365
Iter 14/30 - Loss: 0.317
Iter 15/30 - Loss: 0.278
Iter 16/30 - Loss: 0.229
Iter 17/30 - Loss: 0.182
Iter 18/30 - Loss: 0.148
Iter 19/30 - Loss: 0.103
Iter 20/30 - Loss: 0.057
Iter 21/30 - Loss: -0.000
Iter 22/30 - Loss: -0.060
Iter 23/30 - Loss: -0.117
Iter 24/30 - Loss: -0.178
Iter 25/30 - Loss: -0.235
Iter 26/30 - Loss: -0.290
Iter 27/30 - Loss: -0.356
Iter 28/30 - Loss: -0.405
Iter 29/30 - Loss: -0.455
Iter 30/30 - Loss: -0.508
CPU times: user 30.4 s, sys: 237 ms, total: 30.7 s
Wall time: 12.1 s
```

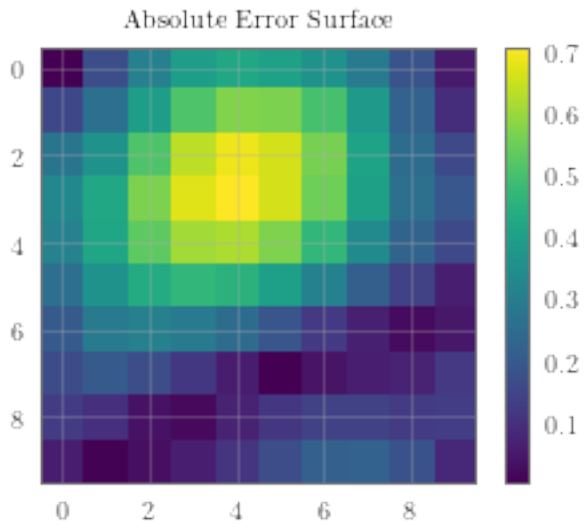
8.1.4 Make predictions with the model

```
In [5]: # Set model and likelihood into evaluation mode
        model.eval()
        likelihood.eval()

        # Generate nxn grid of test points spaced on a grid of size 1/(n-1) in [0,1]x[0,1]
        n = 10
        test_x = torch.zeros(int(pow(n, 2)), 2)
        for i in range(n):
            for j in range(n):
                test_x[i * n + j][0] = float(i) / (n-1)
                test_x[i * n + j][1] = float(j) / (n-1)

        with torch.no_grad(), gpytorch.fast_pred_var():
            observed_pred = likelihood(model(test_x))
            pred_labels = observed_pred.mean.view(n, n)

        # Calc abosolute error
        test_y_actual = torch.sin(((test_x[:, 0] + test_x[:, 1]) * (2 * math.pi))).view(n, n)
        delta_y = torch.abs(pred_labels - test_y_actual).detach().numpy()
```

```
# Define a plotting function
def ax_plot(f, ax, y_labels, title):
    im = ax.imshow(y_labels)
    ax.set_title(title)
    f.colorbar(im)

# Plot our predictive means
f, observed_ax = plt.subplots(1, 1, figsize=(4, 3))
ax_plot(f, observed_ax, pred_labels, 'Predicted Values (Likelihood)')

# Plot the true values
f, observed_ax2 = plt.subplots(1, 1, figsize=(4, 3))
ax_plot(f, observed_ax2, test_y_actual, 'Actual Values (Likelihood)')

# Plot the absolute errors
f, observed_ax3 = plt.subplots(1, 1, figsize=(4, 3))
ax_plot(f, observed_ax3, delta_y, 'Absolute Error Surface')
```







In [ ]:

8.2 Scalable GP Regression (w/ KISS-GP)

8.2.1 Introduction

If the function you are modeling has additive structure across its dimensions, then SKI can be one of the most efficient methods for your problem.

Here, we model the kernel as a sum of kernels that each act on one dimension. Additive SKI (or KISS-GP) can work very well out-of-the-box on larger datasets (100,000+ data points) with many dimensions. This is a strong assumption though, and may not apply to your problem.

This is the same as *the KISSGP Kronecker notebook*, but applied to more dimensions.

```
In [1]: # Imports
        import math
        import torch
        import gpytorch
        import matplotlib.pyplot as plt

        # Inline plotting
        %matplotlib inline
```

Set up train data

Here we're learning a simple sin function - but in 2 dimensions

```
In [2]: # We store the data as a 10k 1D vector
        # It actually represents [0,1]x[0,1] in cartesian coordinates
        n = 100
        train_x = torch.zeros(pow(n, 2), 2)
        for i in range(n):
            for j in range(n):
                # Each coordinate varies from 0 to 1 in n=100 steps
                train_x[i * n + j][0] = float(i) / (n-1)
                train_x[i * n + j][1] = float(j) / (n-1)
```

```
        train_x = train_x.cuda()
        train_y = torch.sin(train_x[:, 0]) + torch.cos(train_x[:, 1]) * (2 * math.pi) + torch.randn_
```

8.2.2 The model

As with the Kronecker example case, applying SKI to a multidimensional kernel is as simple as wrapping that kernel with a `GridInterpolationKernel`. You'll want to be sure to set `num_dims` though!

To use an additive decomposition of the kernel, wrap it in an `AdditiveStructureKernel`.

SKI has only one hyperparameter that you need to worry about: the grid size. For 1D functions, a good starting place is to use as many grid points as training points. (Don't worry - the grid points are really cheap to use, especially with an additive function!).

If you want, you can also explicitly determine the grid bounds of the SKI approximation using the `grid_bounds` argument. However, it's easier if you don't use this argument - then GPyTorch automatically chooses the best bounds for you.

```
In [3]: class GPRegressionModel(gpytorch.models.ExactGP):
            def __init__(self, train_x, train_y, likelihood):
                super(GPRegressionModel, self).__init__(train_x, train_y, likelihood)

                # SKI requires a grid size hyperparameter. This util can help with that
                grid_size = gpytorch.utils.grid.choose_grid_size(train_x)

                self.mean_module = gpytorch.means.ConstantMean()
                self.covar_module = gpytorch.kernels.AdditiveStructureKernel(
                    gpytorch.kernels.GridInterpolationKernel(
                        gpytorch.kernels.ScaleKernel(
                            gpytorch.kernels.RBFKernel(ard_num_dims=2),
                        ), grid_size=grid_size, num_dims=2
                    ), num_dims=2
                )

            def forward(self, x):
                mean_x = self.mean_module(x)
                covar_x = self.covar_module(x)
                return gpytorch.distributions.MultivariateNormal(mean_x, covar_x)


        likelihood = gpytorch.likelihoods.GaussianLikelihood()
        model = GPRegressionModel(train_x, train_y, likelihood).cuda()
```

8.2.3 Train the model hyperparameters

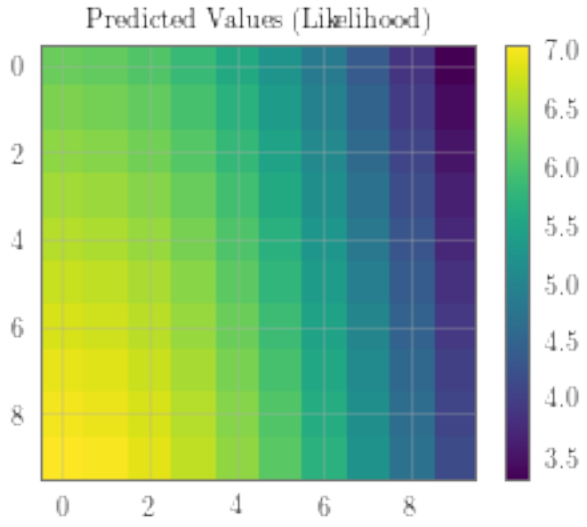
```
In [4]: # Optimize the model
        model.train()
        likelihood.train()

        # Use the adam optimizer
        optimizer = torch.optim.Adam([
            {'params': model.parameters()},  # Includes GaussianLikelihood parameters
        ], lr=0.1)
        num_iter = 20

        # "Loss" for GPs - the marginal log likelihood
```

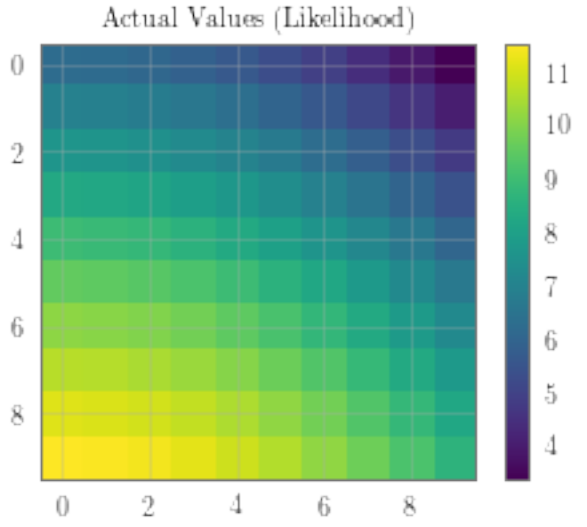
```
        mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)

        # Sometimes we get better performance on the GPU when we don't use Toeplitz math
        # for SKI. This flag controls that
        with gpytorch.settings.use_toeplitz(False):
            for i in range(num_iter):
                optimizer.zero_grad()
                output = model(train_x)
                loss = -mll(output, train_y)
                loss.backward()
                print('Iter %d/%d - Loss: %.3f' % (i + 1, num_iter, loss.data[0]))
                optimizer.step()
```

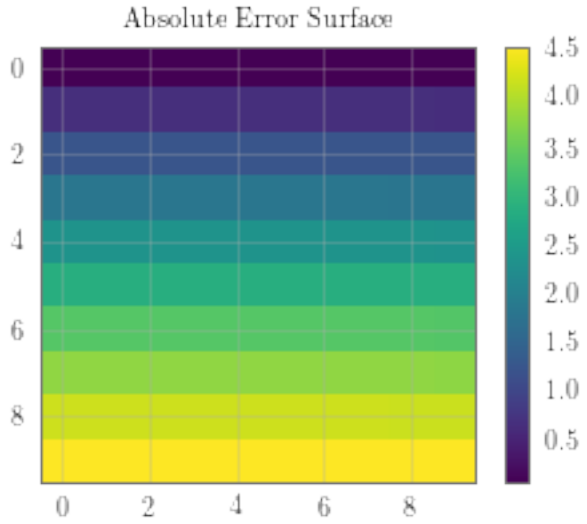
```
Iter 1/20 - Loss: 0.923
Iter 2/20 - Loss: 0.872
Iter 3/20 - Loss: 0.822
Iter 4/20 - Loss: 0.772
Iter 5/20 - Loss: 0.722
Iter 6/20 - Loss: 0.673
Iter 7/20 - Loss: 0.622
Iter 8/20 - Loss: 0.572
Iter 9/20 - Loss: 0.522
Iter 10/20 - Loss: 0.473
Iter 11/20 - Loss: 0.423
Iter 12/20 - Loss: 0.373
Iter 13/20 - Loss: 0.322
Iter 14/20 - Loss: 0.273
Iter 15/20 - Loss: 0.222
Iter 16/20 - Loss: 0.173
Iter 17/20 - Loss: 0.123
Iter 18/20 - Loss: 0.073
Iter 19/20 - Loss: 0.023
Iter 20/20 - Loss: -0.027
```

```
In [5]: # Set into eval mode
        model.eval()
        likelihood.eval()

        # Create 100 test data points
        # Over the square [0,1]x[0,1]
        n = 10
        test_x = torch.zeros(int(pow(n, 2)), 2).cuda()
        for i in range(n):
            for j in range(n):
                test_x[i * n + j][0] = float(i) / (n-1)
                test_x[i * n + j][1] = float(j) / (n-1)

        with torch.no_grad(), gpytorch.fast_pred_var():
            observed_pred = likelihood(model(test_x))
            pred_labels = observed_pred.mean.view(n, n)

        # Calculate the true test values
        test_y_actual = ((torch.sin(test_x.data[:, 0]) + torch.cos(test_x.data[:, 1])) * (2 * math.pi
        test_y_actual = test_y_actual.view(n, n)
        delta_y = torch.abs(pred_labels - test_y_actual)

        # Define a plotting function
        def ax_plot(f, ax, y_labels, title):
            im = ax.imshow(y_labels)
            ax.set_title(title)
```

```
        f.colorbar(im)

    # Make a plot of the predicted values
    f, observed_ax = plt.subplots(1, 1, figsize=(4, 3))
    ax_plot(f, observed_ax, pred_labels.cpu(), 'Predicted Values (Likelihood)')
    # Make a plot of the actual values
    f, observed_ax2 = plt.subplots(1, 1, figsize=(4, 3))
    ax_plot(f, observed_ax2, test_y_actual.cpu(), 'Actual Values (Likelihood)')
    # Make a plot of the errors
    f, observed_ax3 = plt.subplots(1, 1, figsize=(4, 3))
    ax_plot(f, observed_ax3, delta_y.cpu(), 'Absolute Error Surface')
```







In [ ]:

8.3 Deep Kernel Learning GP Regression (w/ KISS-GP)

8.3.1 Overview

In this notebook, we'll give a brief tutorial on how to use deep kernel learning for regression on a medium scale dataset using SKI. This also demonstrates how to incorporate standard PyTorch modules in to a Gaussian process model.

```
In [1]: import math
        import torch
        import gpytorch
        from matplotlib import pyplot as plt

        # Make plots inline
        %matplotlib inline
```

8.3.2 Loading Data

For this example notebook, we'll be using the `elevators` UCI dataset used in the paper. Running the next cell downloads a copy of the dataset that has already been scaled and normalized appropriately. For this notebook, we'll simply be splitting the data using the first 80% of the data as training and the last 20% as testing.

Note: Running the next cell will attempt to download a ~400 KB dataset file to the current directory.

```
In [2]: import urllib.request
        import os.path
        from scipy.io import loadmat
        from math import floor

        if not os.path.isfile('elevators.mat'):
            print('Downloading \'elevators\' UCI dataset...')
            urllib.request.urlretrieve('https://drive.google.com/uc?export=download&id=1jhWL3YUHvXIa

        data = torch.Tensor(loadmat('elevators.mat')['data'])
        X = data[:, :-1]
```

```
X = X - X.min(0)[0]
X = 2 * (X / X.max(0)[0]) - 1
y = data[:, -1]

# Use the first 80% of the data for training, and the last 20% for testing.
train_n = int(floor(0.8*len(X)))

train_x = X[:train_n, :].contiguous().cuda()
train_y = y[:train_n].contiguous().cuda()

test_x = X[train_n:, :].contiguous().cuda()
test_y = y[train_n:].contiguous().cuda()
```

8.3.3 Defining the DKL Feature Extractor

Next, we define the neural network feature extractor used to define the deep kernel. In this case, we use a fully connected network with the architecture `d -> 1000 -> 500 -> 50 -> 2`, as described in the original DKL paper. All of the code below uses standard PyTorch implementations of neural network layers.

```
In [3]: data_dim = train_x.size(-1)

        class LargeFeatureExtractor(torch.nn.Sequential):
            def __init__(self):
                super(LargeFeatureExtractor, self).__init__()
                self.add_module('linear1', torch.nn.Linear(data_dim, 1000))
                self.add_module('relu1', torch.nn.ReLU())
                self.add_module('linear2', torch.nn.Linear(1000, 500))
                self.add_module('relu2', torch.nn.ReLU())
                self.add_module('linear3', torch.nn.Linear(500, 50))
                self.add_module('relu3', torch.nn.ReLU())
                self.add_module('linear4', torch.nn.Linear(50, 2))

        feature_extractor = LargeFeatureExtractor().cuda()
```

8.3.4 Defining the GP Model

We now define the GP model. For more details on the use of GP models, see our simpler examples. This model uses a `GridInterpolationKernel` (SKI) with an RBF base kernel.

The forward method

In deep kernel learning, the forward method is where most of the interesting new stuff happens. Before calling the mean and covariance modules on the data as in the simple GP regression setting, we first pass the input data `x` through the neural network feature extractor. Then, to ensure that the output features of the neural network remain in the grid bounds expected by SKI, we scales the resulting features to be between 0 and 1.

Only after this processing do we call the mean and covariance module of the Gaussian process. This example also demonstrates the flexibility of defining GP models that allow for learned transformations of the data (in this case, via a neural network) before calling the mean and covariance function. Because the neural network in this case maps to two final output features, we will have no problem using SKI.

```
In [4]: class GPRegressionModel(gpytorch.models.ExactGP):
            def __init__(self, train_x, train_y, likelihood):
                super(GPRegressionModel, self).__init__(train_x, train_y, likelihood)
                self.mean_module = gpytorch.means.ConstantMean()
```

```
                    self.covar_module = gpytorch.kernels.GridInterpolationKernel(
                        gpytorch.kernels.ScaleKernel(gpytorch.kernels.RBFKernel(ard_num_dims=2)),
                        num_dims=2, grid_size=100
                    )
                    self.feature_extractor = feature_extractor

                def forward(self, x):
                    # We're first putting our data through a deep net (feature extractor)
                    # We're also scaling the features so that they're nice values
                    projected_x = self.feature_extractor(x)
                    projected_x = projected_x - projected_x.min(0)[0]
                    projected_x = 2 * (projected_x / projected_x.max(0)[0]) - 1

                    mean_x = self.mean_module(projected_x)
                    covar_x = self.covar_module(projected_x)
                    return gpytorch.distributions.MultivariateNormal(mean_x, covar_x)

In [5]: likelihood = gpytorch.likelihoods.GaussianLikelihood().cuda()
        model = GPRegressionModel(train_x, train_y, likelihood).cuda()
```

8.3.5 Training the model

The cell below trains the DKL model above, learning both the hyperparameters of the Gaussian process **and** the parameters of the neural network in an end-to-end fashion using Type-II MLE. We run 20 iterations of training using the `Adam` optimizer built in to PyTorch. With a decent GPU, this should only take a few seconds.

```
In [6]: # Find optimal model hyperparameters
        model.train()
        likelihood.train()

        # Use the adam optimizer
        optimizer = torch.optim.SGD([
            {'params': model.feature_extractor.parameters(), 'weight_decay': 1e-3},
            {'params': model.covar_module.parameters()},
            {'params': model.mean_module.parameters()},
            {'params': model.likelihood.parameters()},
        ], lr=0.1)

        # "Loss" for GPs - the marginal log likelihood
        mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)

        training_iterations = 40
        def train():
            for i in range(training_iterations):
                # Zero backprop gradients
                optimizer.zero_grad()
                # Get output from model
                output = model(train_x)
                # Calc loss and backprop derivatives
                loss = -mll(output, train_y)
                loss.backward()
                print('Iter %d/%d - Loss: %.3f' % (i + 1, training_iterations, loss.item()))
                optimizer.step()

        # See dkl_mnist.ipynb for explanation of this flag
        with gpytorch.settings.use_toeplitz(True):
            %time train()

Iter 1/40 - Loss: 0.942
```

```
Iter 2/40 - Loss: 0.925
Iter 3/40 - Loss: 0.905
Iter 4/40 - Loss: 0.878
Iter 5/40 - Loss: 0.856
Iter 6/40 - Loss: 0.831
Iter 7/40 - Loss: 0.806
Iter 8/40 - Loss: 0.784
Iter 9/40 - Loss: 0.762
Iter 10/40 - Loss: 0.738
Iter 11/40 - Loss: 0.717
Iter 12/40 - Loss: 0.694
Iter 13/40 - Loss: 0.672
Iter 14/40 - Loss: 0.649
Iter 15/40 - Loss: 0.626
Iter 16/40 - Loss: 0.603
Iter 17/40 - Loss: 0.580
Iter 18/40 - Loss: 0.557
Iter 19/40 - Loss: 0.536
Iter 20/40 - Loss: 0.514
Iter 21/40 - Loss: 0.490
Iter 22/40 - Loss: 0.468
Iter 23/40 - Loss: 0.448
Iter 24/40 - Loss: 0.425
Iter 25/40 - Loss: 0.400
Iter 26/40 - Loss: 0.379
Iter 27/40 - Loss: 0.358
Iter 28/40 - Loss: 0.338
Iter 29/40 - Loss: 0.317
Iter 30/40 - Loss: 0.292
Iter 31/40 - Loss: 0.275
Iter 32/40 - Loss: 0.256
Iter 33/40 - Loss: 0.239
Iter 34/40 - Loss: 0.237
Iter 35/40 - Loss: 0.215
Iter 36/40 - Loss: 0.201
Iter 37/40 - Loss: 0.181
Iter 38/40 - Loss: 0.156
Iter 39/40 - Loss: 0.128
Iter 40/40 - Loss: 0.111
CPU times: user 15.2 s, sys: 4.52 s, total: 19.7 s
Wall time: 19.7 s
```

8.3.6 Making Predictions

The next cell gets the predictive covariance for the test set (and also technically gets the predictive mean, stored in `preds.mean()`) using the standard SKI testing code, with no acceleration or precomputation.

```
In [7]: model.eval()
        likelihood.eval()
        with torch.no_grad(), gpytorch.settings.use_toeplitz(False), gpytorch.fast_pred_var():
            preds = model(test_x)
```

```
In [8]: print('Test MAE: {}'.format(torch.mean(torch.abs(preds.mean - test_y))))
```

```
Test MAE: 0.11873025447130203
```

```
In [ ]:
```

8.4 Scalable GP Regression (CUDA) with Fast Predictive Distributions (KISS-GP/LOVE)

8.4.1 Introduction

In this notebook, we'll give a brief tutorial on how to use Lanczos Variance Estimates (LOVE) to achieve fast predictive distributions as described in this paper https://arxiv.org/abs/1803.06058. To see LOVE in use with exact GPs, see `fast_variances_exact_(LOVE).ipynb`. For a tutorial on using the fast sampling mechanism described in the paper, see `fast_sampling_ski_(LOVE).ipynb`.

LOVE is an algorithm for approximating the predictive covariances of a Gaussian process in constant time after linear time precomputation. In this notebook, we will train a deep kernel learning model with SKI on one of the UCI datasets used in the paper, and then compare the time required to make predictions with each model.

NOTE: The timing results reported in the paper compare the time required to compute (co)variances **only**. Because excluding the mean computations from the timing results requires hacking the internals of GPyTorch, the timing results presented in this notebook include the time required to compute predictive means, which are not accelerated by LOVE. Nevertheless, as we will see, LOVE achieves impressive speed-ups.

```
In [1]: import math
        import torch
        import gpytorch
        from matplotlib import pyplot as plt

        # Make plots inline
        %matplotlib inline
```

8.4.2 Loading Data

For this example notebook, we'll be using the `elevators` UCI dataset used in the paper. Running the next cell downloads a copy of the dataset that has already been scaled and normalized appropriately. For this notebook, we'll simply be splitting the data using the first 80% of the data as training and the last 20% as testing.

Note: Running the next cell will attempt to download a ~400 KB dataset file to the current directory.

```
In [2]: import urllib.request
        import os.path
        from scipy.io import loadmat
        from math import floor

        if not os.path.isfile('elevators.mat'):
            print('Downloading \'elevators\' UCI dataset...')
            urllib.request.urlretrieve('https://drive.google.com/uc?export=download&id=1jhWL3YUHvXIa

        data = torch.Tensor(loadmat('elevators.mat')['data'])
        X = data[:, :-1]
        X = X - X.min(0)[0]
        X = 2 * (X / X.max(0)[0]) - 1
        y = data[:, -1]

        # Use the first 80% of the data for training, and the last 20% for testing.
        train_n = int(floor(0.8*len(X)))

        train_x = X[:train_n, :].contiguous().cuda()
        train_y = y[:train_n].contiguous().cuda()
```

```
test_x = X[train_n:, :].contiguous().cuda()
test_y = y[train_n:].contiguous().cuda()
```

8.4.3 Defining the DKL Feature Extractor

Next, we define the deep feature extractor we'll be using for DKL. In this case, we use a fully connected network with the architecture `d -> 1000 -> 500 -> 50 -> 2`, as described in the original DKL paper. All of the code below uses standard PyTorch implementations of neural network layers.

```
In [3]: data_dim = train_x.size(-1)

        class LargeFeatureExtractor(torch.nn.Sequential):
            def __init__(self):
                super(LargeFeatureExtractor, self).__init__()
                self.add_module('linear1', torch.nn.Linear(data_dim, 1000))
                self.add_module('relu1', torch.nn.ReLU())
                self.add_module('linear2', torch.nn.Linear(1000, 500))
                self.add_module('relu2', torch.nn.ReLU())
                self.add_module('linear3', torch.nn.Linear(500, 50))
                self.add_module('relu3', torch.nn.ReLU())
                self.add_module('linear4', torch.nn.Linear(50, 2))

        feature_extractor = LargeFeatureExtractor().cuda()
```

8.4.4 Defining the GP Model

We now define the GP model. For more details on the use of GP models, see our simpler examples. This model uses a `GridInterpolationKernel` (SKI) with an RBF base kernel. The forward method passes the input data `x` through the neural network feature extractor defined above, scales the resulting features to be between 0 and 1, and then calls the kernel.

```
In [4]: class GPRegressionModel(gpytorch.models.ExactGP):
            def __init__(self, train_x, train_y, likelihood):
                super(GPRegressionModel, self).__init__(train_x, train_y, likelihood)

                self.mean_module = gpytorch.means.ConstantMean()
                self.covar_module = gpytorch.kernels.GridInterpolationKernel(
                    gpytorch.kernels.ScaleKernel(gpytorch.kernels.RBFKernel()),
                    grid_size=100, num_dims=2,
                )

                # Also add the deep net
                self.feature_extractor = feature_extractor

            def forward(self, x):
                # We're first putting our data through a deep net (feature extractor)
                # We're also scaling the features so that they're nice values
                projected_x = self.feature_extractor(x)
                projected_x = projected_x - projected_x.min(0)[0]
                projected_x = 2 * (projected_x / projected_x.max(0)[0]) - 1

                # The rest of this looks like what we've seen
                mean_x = self.mean_module(projected_x)
                covar_x = self.covar_module(projected_x)
                return gpytorch.distributions.MultivariateNormal(mean_x, covar_x)
```

```
        likelihood = gpytorch.likelihoods.GaussianLikelihood()
        model = GPRegressionModel(train_x, train_y, likelihood).cuda()
```

8.4.5 Training the model

The cell below trains the DKL model above, finding optimal hyperparameters using Type-II MLE. We run 20 iterations of training using the `Adam` optimizer built in to PyTorch. With a decent GPU, this should only take a few seconds.

It's good to add some L2 regularization to the feature extractor part of the model, but NOT to any other part of the model.

```
In [5]: # Find optimal model hyperparameters
        model.train()
        likelihood.train()

        # Use the adam optimizer
        # Add weight decay to the feature exactractor ONLY
        optimizer = torch.optim.Adam([
            {'params': model.mean_module.parameters()},
            {'params': model.covar_module.parameters()},
            {'params': model.likelihood.parameters()},
            {'params': model.feature_extractor.parameters(), 'weight_decay': 1e-3}
        ], lr=0.1)

        # "Loss" for GPs - the marginal log likelihood
        mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)

        def train(training_iterations=20):
            for i in range(training_iterations):
                optimizer.zero_grad()
                output = model(train_x)
                loss = -mll(output, train_y)
                loss.backward()
                print('Iter %d/%d - Loss: %.3f' % (i + 1, training_iterations, loss.item()))
                optimizer.step()

        # Sometimes we get better performance on the GPU when we don't use Toeplitz math
        # for SKI. This flag controls that
        with gpytorch.settings.use_toeplitz(False):
            %time train()
```

```
Iter 1/20 - Loss: 0.949
Iter 2/20 - Loss: 0.892
Iter 3/20 - Loss: 0.843
Iter 4/20 - Loss: 0.794
Iter 5/20 - Loss: 0.747
Iter 6/20 - Loss: 0.696
Iter 7/20 - Loss: 0.646
Iter 8/20 - Loss: 0.598
Iter 9/20 - Loss: 0.548
Iter 10/20 - Loss: 0.500
Iter 11/20 - Loss: 0.449
Iter 12/20 - Loss: 0.397
Iter 13/20 - Loss: 0.343
Iter 14/20 - Loss: 0.306
Iter 15/20 - Loss: 0.264
Iter 16/20 - Loss: 0.245
Iter 17/20 - Loss: 0.210
Iter 18/20 - Loss: 0.171
```

```
Iter 19/20 - Loss: 0.117
Iter 20/20 - Loss: 0.063
CPU times: user 6.17 s, sys: 2.12 s, total: 8.29 s
Wall time: 8.28 s
```

8.4.6 Make Predictions using Standard SKI Code

The next cell gets the predictive covariance for the test set (and also technically gets the predictive mean, stored in `preds.mean()`) using the standard SKI testing code, with no acceleration or precomputation.

Note: Full predictive covariance matrices (and the computations needed to get them) can be quite memory intensive. Depending on the memory available on your GPU, you may need to reduce the size of the test set for the code below to run. If you run out of memory, try replacing `test_x` below with something like `test_x[:1000]` to use the first 1000 test points only, and then restart the notebook.

```
In [6]: import time

        # Set into eval mode
        model.eval()
        likelihood.eval()
        with torch.no_grad(), gpytorch.settings.use_toeplitz(False):
            start_time = time.time()
            preds = model(test_x[:1000])
            exact_covar = preds.covariance_matrix
            exact_covar_time = time.time() - start_time
```

```
In [7]: print('Time to compute exact mean + covariances: {:.2f}s'.format(exact_covar_time))
```

```
Time to compute exact mean + covariances: 1.61s
```

8.4.7 Clear Memory and any Precomputed Values

The next cell clears as much as possible to avoid influencing the timing results of the fast predictive variances code. Strictly speaking, the timing results above and the timing results to follow should be run in entirely separate notebooks. However, this will suffice for this simple example.

```
In [8]: # Clear as much 'stuff' as possible
        import gc
        gc.collect()
        torch.cuda.empty_cache()
        model.train()
        likelihood.train()
```

```
Out[8]: GaussianLikelihood()
```

8.4.8 Compute Predictions with LOVE, but Before Precomputation

Next we compute predictive covariances (and the predictive means) for LOVE, but starting from scratch. That is, we don't yet have access to the precomputed cache discussed in the paper. This should still be faster than the full covariance computation code above.

In this simple example, we allow a rank 10 root decomposition, although increasing this to rank 20-40 should not affect the timing results substantially.

```
In [9]: # Set into eval mode
        model.eval()
        likelihood.eval()
```

```
        with torch.no_grad(), gpytorch.settings.use_toeplitz(False), gpytorch.beta_features.fast_pred
            start_time = time.time()
            preds = model(test_x[:1000])
            fast_time_no_cache = time.time() - start_time
```

8.4.9 Compute Predictions with LOVE After Precomputation

The above cell additionally computed the caches required to get fast predictions. From this point onwards, unless we put the model back in training mode, predictions should be extremely fast. The cell below re-runs the above code, but takes full advantage of both the mean cache and the LOVE cache for variances.

```
In [10]: with torch.no_grad(), gpytorch.settings.use_toeplitz(False), gpytorch.beta_features.fast_pre
             start_time = time.time()
             preds = model(test_x[:1000])
             fast_covar = preds.covariance_matrix
             fast_time_with_cache = time.time() - start_time
```

```
In [11]: print('Time to compute mean + covariances (no cache) {:.2f}s'.format(fast_time_no_cache))
         print('Time to compute mean + variances (cache): {:.2f}s'.format(fast_time_with_cache))
```

```
Time to compute mean + covariances (no cache) 0.68s
Time to compute mean + variances (cache): 0.03s
```

8.4.10 Compute Error between Exact and Fast Variances

Finally, we compute the mean absolute error between the fast variances computed by LOVE (stored in fast_covar), and the exact variances computed previously.

Note that these tests were run with a root decomposition of rank 10, which is about the minimum you would realistically ever run with. Despite this, the fast variance estimates are quite good. If more accuracy was needed, increasing `max_root_decomposition_size` to 30 or 40 would provide even better estimates.

```
In [12]: print('MAE between exact covar matrix and fast covar matrix: {}'.format((exact_covar - fast_
```

```
MAE between exact covar matrix and fast covar matrix: 0.012722853571176529
```

CHAPTER 9

Scalable GP Classification (1D)

9.1 Scalable GP Classification in 1D (w/ KISS-GP)

This example shows how to use a `GridInducingVariationalGP` module. This classification module is designed for when the inputs of the function you're modeling are one-dimensional.

The use of inducing points allows for scaling up the training data by making computational complexity linear instead of cubic.

In this example, we're modeling a function that is periodically labeled cycling every 1/8 (think of a square wave with period 1/4)

This notebook doesn't use cuda, in general we recommend GPU use if possible and most of our notebooks utilize cuda as well.

Kernel interpolation for scalable structured Gaussian processes (KISS-GP) was introduced in this paper: http://proceedings.mlr.press/v37/wilson15.pdf

KISS-GP with SVI for classification was introduced in this paper: https://papers.nips.cc/paper/6426-stochastic-variational-deep-kernel-learning.pdf

```
In [1]: import math
        import torch
        import gpytorch
        from matplotlib import pyplot as plt
        from math import exp

        %matplotlib inline
        %load_ext autoreload
        %autoreload 2
```

```
In [2]: train_x = torch.linspace(0, 1, 26)
        train_y = torch.sign(torch.cos(train_x * (2 * math.pi)))
```

```
In [3]: class GPClassificationModel(gpytorch.models.GridInducingVariationalGP):
            def __init__(self):
                super(GPClassificationModel, self).__init__(grid_size=32, grid_bounds=[(0, 1)])
```

```
                self.mean_module = gpytorch.means.ConstantMean()
                self.covar_module = gpytorch.kernels.ScaleKernel(
                    gpytorch.kernels.RBFKernel(
                        log_lengthscale_prior=gpytorch.priors.SmoothedBoxPrior(
                            exp(0), exp(3), sigma=0.1, log_transform=True
                        )
                    )
                )

            def forward(self,x):
                mean_x = self.mean_module(x)
                covar_x = self.covar_module(x)
                latent_pred = gpytorch.distributions.MultivariateNormal(mean_x, covar_x)
                return latent_pred


        model = GPClassificationModel()
        likelihood = gpytorch.likelihoods.BernoulliLikelihood()
```

In [4]:
```
        # Find optimal model hyperparameters
        model.train()
        likelihood.train()

        # Use the adam optimizer
        optimizer = torch.optim.SGD([
            {'params': model.parameters()},
            # BernoulliLikelihood has no parameters
        ], lr=0.1)

        # "Loss" for GPs - the marginal log likelihood
        # n_data refers to the amount of training data
        mll = gpytorch.mlls.VariationalMarginalLogLikelihood(likelihood, model, num_data=len(train_y))

        def train():
            num_iter = 200
            for i in range(num_iter):
                optimizer.zero_grad()
                output = model(train_x)
                loss = -mll(output, train_y)
                loss.backward()
                print('Iter %d/%d - Loss: %.3f' % (i + 1, num_iter, loss.item()))
                optimizer.step()

        # Get clock time
        %time train()
```

```
Iter 1/200 - Loss: 6097.150
Iter 2/200 - Loss: 392.698
Iter 3/200 - Loss: 352.528
Iter 4/200 - Loss: 322.668
Iter 5/200 - Loss: 324.150
Iter 6/200 - Loss: 362.064
Iter 7/200 - Loss: 416.016
Iter 8/200 - Loss: 298.819
Iter 9/200 - Loss: 252.523
Iter 10/200 - Loss: 211.480
Iter 11/200 - Loss: 321.593
Iter 12/200 - Loss: 289.411
Iter 13/200 - Loss: 347.734
Iter 14/200 - Loss: 330.747
```

```
Iter 15/200 - Loss: 296.441
Iter 16/200 - Loss: 320.904
Iter 17/200 - Loss: 309.613
Iter 18/200 - Loss: 281.426
Iter 19/200 - Loss: 296.923
Iter 20/200 - Loss: 301.174
Iter 21/200 - Loss: 250.977
Iter 22/200 - Loss: 343.925
Iter 23/200 - Loss: 240.633
Iter 24/200 - Loss: 281.263
Iter 25/200 - Loss: 271.904
Iter 26/200 - Loss: 325.001
Iter 27/200 - Loss: 272.500
Iter 28/200 - Loss: 271.221
Iter 29/200 - Loss: 284.778
Iter 30/200 - Loss: 243.273
Iter 31/200 - Loss: 292.012
Iter 32/200 - Loss: 302.940
Iter 33/200 - Loss: 290.159
Iter 34/200 - Loss: 320.918
Iter 35/200 - Loss: 259.498
Iter 36/200 - Loss: 277.864
Iter 37/200 - Loss: 293.995
Iter 38/200 - Loss: 223.749
Iter 39/200 - Loss: 265.449
Iter 40/200 - Loss: 265.746
Iter 41/200 - Loss: 250.393
Iter 42/200 - Loss: 261.780
Iter 43/200 - Loss: 239.201
Iter 44/200 - Loss: 317.880
Iter 45/200 - Loss: 326.695
Iter 46/200 - Loss: 282.346
Iter 47/200 - Loss: 268.213
Iter 48/200 - Loss: 278.097
Iter 49/200 - Loss: 246.392
Iter 50/200 - Loss: 321.273
Iter 51/200 - Loss: 274.065
Iter 52/200 - Loss: 263.765
Iter 53/200 - Loss: 219.523
Iter 54/200 - Loss: 303.933
Iter 55/200 - Loss: 256.324
Iter 56/200 - Loss: 203.140
Iter 57/200 - Loss: 289.328
Iter 58/200 - Loss: 261.303
Iter 59/200 - Loss: 225.208
Iter 60/200 - Loss: 222.645
Iter 61/200 - Loss: 234.964
Iter 62/200 - Loss: 292.547
Iter 63/200 - Loss: 233.788
Iter 64/200 - Loss: 231.033
Iter 65/200 - Loss: 194.131
Iter 66/200 - Loss: 230.455
Iter 67/200 - Loss: 252.459
Iter 68/200 - Loss: 227.107
Iter 69/200 - Loss: 252.148
Iter 70/200 - Loss: 229.926
Iter 71/200 - Loss: 244.014
Iter 72/200 - Loss: 210.347
Iter 73/200 - Loss: 264.777
```

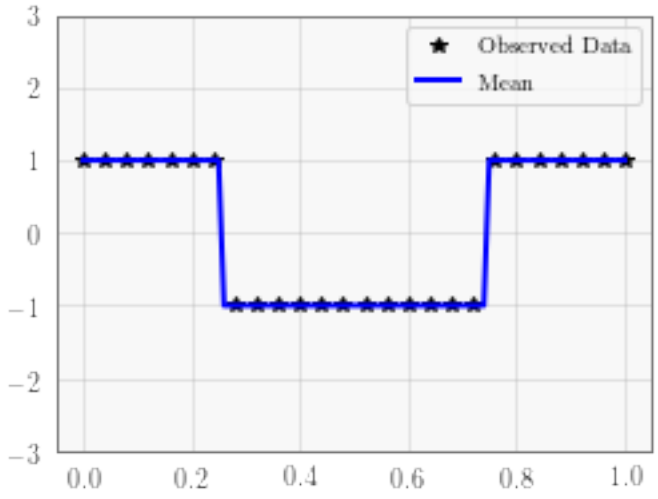
```
Iter 74/200 - Loss: 235.150
Iter 75/200 - Loss: 239.858
Iter 76/200 - Loss: 205.147
Iter 77/200 - Loss: 199.181
Iter 78/200 - Loss: 235.487
Iter 79/200 - Loss: 250.423
Iter 80/200 - Loss: 211.550
Iter 81/200 - Loss: 211.175
Iter 82/200 - Loss: 213.312
Iter 83/200 - Loss: 197.529
Iter 84/200 - Loss: 249.012
Iter 85/200 - Loss: 241.818
Iter 86/200 - Loss: 226.489
Iter 87/200 - Loss: 251.521
Iter 88/200 - Loss: 203.768
Iter 89/200 - Loss: 220.160
Iter 90/200 - Loss: 243.473
Iter 91/200 - Loss: 214.500
Iter 92/200 - Loss: 213.951
Iter 93/200 - Loss: 245.208
Iter 94/200 - Loss: 201.523
Iter 95/200 - Loss: 199.266
Iter 96/200 - Loss: 214.818
Iter 97/200 - Loss: 228.327
Iter 98/200 - Loss: 243.201
Iter 99/200 - Loss: 193.552
Iter 100/200 - Loss: 226.596
Iter 101/200 - Loss: 207.586
Iter 102/200 - Loss: 229.452
Iter 103/200 - Loss: 211.403
Iter 104/200 - Loss: 194.898
Iter 105/200 - Loss: 192.584
Iter 106/200 - Loss: 218.825
Iter 107/200 - Loss: 197.878
Iter 108/200 - Loss: 201.669
Iter 109/200 - Loss: 246.887
Iter 110/200 - Loss: 232.580
Iter 111/200 - Loss: 208.174
Iter 112/200 - Loss: 217.168
Iter 113/200 - Loss: 195.321
Iter 114/200 - Loss: 246.281
Iter 115/200 - Loss: 249.421
Iter 116/200 - Loss: 200.820
Iter 117/200 - Loss: 191.208
Iter 118/200 - Loss: 227.009
Iter 119/200 - Loss: 264.285
Iter 120/200 - Loss: 200.157
Iter 121/200 - Loss: 209.431
Iter 122/200 - Loss: 190.169
Iter 123/200 - Loss: 223.926
Iter 124/200 - Loss: 231.914
Iter 125/200 - Loss: 196.829
Iter 126/200 - Loss: 176.027
Iter 127/200 - Loss: 197.739
Iter 128/200 - Loss: 163.040
Iter 129/200 - Loss: 221.040
Iter 130/200 - Loss: 209.215
Iter 131/200 - Loss: 169.048
Iter 132/200 - Loss: 134.395
```

```
Iter 133/200 - Loss: 194.889
Iter 134/200 - Loss: 239.895
Iter 135/200 - Loss: 207.784
Iter 136/200 - Loss: 224.677
Iter 137/200 - Loss: 185.859
Iter 138/200 - Loss: 194.485
Iter 139/200 - Loss: 198.281
Iter 140/200 - Loss: 177.267
Iter 141/200 - Loss: 177.465
Iter 142/200 - Loss: 196.033
Iter 143/200 - Loss: 143.547
Iter 144/200 - Loss: 187.040
Iter 145/200 - Loss: 203.045
Iter 146/200 - Loss: 192.756
Iter 147/200 - Loss: 180.532
Iter 148/200 - Loss: 175.648
Iter 149/200 - Loss: 191.526
Iter 150/200 - Loss: 166.489
Iter 151/200 - Loss: 220.140
Iter 152/200 - Loss: 167.087
Iter 153/200 - Loss: 148.467
Iter 154/200 - Loss: 220.460
Iter 155/200 - Loss: 160.580
Iter 156/200 - Loss: 196.464
Iter 157/200 - Loss: 185.087
Iter 158/200 - Loss: 148.367
Iter 159/200 - Loss: 158.299
Iter 160/200 - Loss: 187.548
Iter 161/200 - Loss: 181.689
Iter 162/200 - Loss: 172.187
Iter 163/200 - Loss: 191.411
Iter 164/200 - Loss: 167.754
Iter 165/200 - Loss: 138.704
Iter 166/200 - Loss: 162.195
Iter 167/200 - Loss: 186.930
Iter 168/200 - Loss: 182.635
Iter 169/200 - Loss: 158.236
Iter 170/200 - Loss: 160.126
Iter 171/200 - Loss: 180.415
Iter 172/200 - Loss: 187.367
Iter 173/200 - Loss: 163.659
Iter 174/200 - Loss: 184.058
Iter 175/200 - Loss: 216.402
Iter 176/200 - Loss: 169.361
Iter 177/200 - Loss: 183.626
Iter 178/200 - Loss: 174.367
Iter 179/200 - Loss: 157.275
Iter 180/200 - Loss: 171.675
Iter 181/200 - Loss: 192.713
Iter 182/200 - Loss: 158.222
Iter 183/200 - Loss: 173.345
Iter 184/200 - Loss: 150.134
Iter 185/200 - Loss: 189.955
Iter 186/200 - Loss: 170.120
Iter 187/200 - Loss: 200.875
Iter 188/200 - Loss: 140.360
Iter 189/200 - Loss: 136.488
Iter 190/200 - Loss: 201.296
Iter 191/200 - Loss: 163.410
```

```
Iter 192/200 - Loss: 174.225
Iter 193/200 - Loss: 218.408
Iter 194/200 - Loss: 178.131
Iter 195/200 - Loss: 162.437
Iter 196/200 - Loss: 145.230
Iter 197/200 - Loss: 151.984
Iter 198/200 - Loss: 121.274
Iter 199/200 - Loss: 137.730
Iter 200/200 - Loss: 145.839
CPU times: user 18.3 s, sys: 19.7 s, total: 38 s
Wall time: 5.44 s
```

In [5]:
```
# Set model and likelihood into eval mode
model.eval()
likelihood.eval()

# Initialize axes
f, ax = plt.subplots(1, 1, figsize=(4, 3))

with torch.no_grad():
    test_x = torch.linspace(0, 1, 101)
    predictions = likelihood(model(test_x))

ax.plot(train_x.numpy(), train_y.numpy(), 'k*')
pred_labels = predictions.mean.ge(0.5).float().mul(2).sub(1)
ax.plot(test_x.data.numpy(), pred_labels.numpy(), 'b')
ax.set_ylim([-3, 3])
ax.legend(['Observed Data', 'Mean', 'Confidence'])
```

Out[5]: <matplotlib.legend.Legend at 0x7f0bec141080>



In [ ]:

CHAPTER 10

Scalable GP Classification (Multidimensional)

10.1 Scalable Additive-Structure GP Classification (CUDA) (w/ KISS-GP)

10.1.1 Introduction

This example shows how to use a `AdditiveGridInducingVariationalGP` module. This classifcation module is designed for when the function you're modeling has an additive decomposition over dimension. This is equivalent to using a covariance function that additively decomposes over dimensions:

$$k(\mathbf{x}, \mathbf{x}') = \sum_{i=1}^{d} k([\mathbf{x}]_i, [\mathbf{x}']_i)$$

where $[\mathbf{x}]_i$ denotes the ith component of the vector $\mathbf{x}$. Example applications of this include use in Bayesian optimization, and when performing deep kernel learning.

The use of inducing points allows for scaling up the training data by making computational complexity linear instead of cubic in the number of data points.

In this example, we're performing classification on a two dimensional toy dataset that is: - Defined in [-1, 1]x[-1, 1] - Valued 1 in [-0.5, 0.5]x[-0.5, 0.5] - Valued -1 otherwise

The above function doesn't have an obvious additive decomposition, but it turns out that this function is can be very well approximated by the kernel anyways.

```
In [1]: # High-level imports
        import math
        import torch
        import gpytorch
        from matplotlib import pyplot as plt

        # Make inline plots
        %matplotlib inline
```

10.1.2 Generate toy dataset

```
In [2]: n = 201
        train_x = torch.zeros(n ** 2, 2)
        train_x[:, 0].copy_(torch.linspace(-1, 1, n).repeat(n))
        train_x[:, 1].copy_(torch.linspace(-1, 1, n).unsqueeze(1).repeat(1, n).view(-1))
        train_y = (train_x[:, 0].abs().lt(0.5)).float() * (train_x[:, 1].abs().lt(0.5)).float() * 2 -

        train_x = train_x.cuda()
        train_y = train_y.cuda()
```

10.1.3 Define the model

In contrast to the most basic classification models, this model extends `AdditiveGridInducingVariationalGP`. This causes two key changes in the model. First, the base class specifically assumes that the input to `forward`, `x`, is to be additive decomposed. Thus, although the model below defines an `RBFKernel` as the covariance function, because we extend this base class, the additive decomposition discussed above will be imposed.

Second, this model automatically assumes we will be using scalable kernel interpolation (SKI) for each dimension. Because of the additive decomposition, we only provide one set of grid bounds to the base class constructor, as the same grid will be used for all dimensions. It is recommended that you scale your training and test data appropriately.

```
In [6]: # For GP Classification we use the AdditiveGridInducingVariationalGP model
        class GPClassificationModel(gpytorch.models.AdditiveGridInducingVariationalGP):
            def __init__(self):
                super(GPClassificationModel, self).__init__(grid_size=100, grid_bounds=[(-1, 1)], nu
                self.mean_module = gpytorch.means.ConstantMean()
                self.covar_module = gpytorch.kernels.ScaleKernel(
                    gpytorch.kernels.RBFKernel(
                        log_lengthscale_prior=gpytorch.priors.SmoothedBoxPrior(
                            math.exp(0), math.exp(3), sigma=0.1, log_transform=True
                        )
                    )
                )


            def forward(self,x):
                mean_x = self.mean_module(x)
                covar_x = self.covar_module(x)
                latent_pred = gpytorch.distributions.MultivariateNormal(mean_x, covar_x)
                return latent_pred

        # Cuda the model and likelihood function
        model = GPClassificationModel().cuda()
        likelihood = gpytorch.likelihoods.BernoulliLikelihood().cuda()
```

10.1.4 Training the model

Once the model has been defined, the training loop looks very similar to other variational models we've seen in the past. We will optimize the variational lower bound as our objective function. In this case, although variational inference in GPyTorch supports stochastic gradient descent, we choose to do batch optimization due to the relatively small toy dataset.

For an example of using the `AdditiveGridInducingVariationalGP` model with stochastic gradient descent, see the `dkl_mnist` example.

```
In [7]: # Find optimal model hyperparameters
        model.train()
        likelihood.train()

        # Use the adam optimizer
        optimizer = torch.optim.Adam([
            {'params': model.parameters()},
            # BernoulliLikelihood has no parameters
        ], lr=0.1)

        # "Loss" for GPs - the marginal log likelihood
        # n_data refers to the amount of training data
        mll = gpytorch.mlls.VariationalMarginalLogLikelihood(likelihood, model, num_data=len(train_y))

        # Training function
        def train():
            num_iter = 25
            for i in range(num_iter):
                optimizer.zero_grad()
                output = model(train_x)
                loss = -mll(output, train_y)
                loss.backward()
                print('Iter %d/%d - Loss: %.3f' % (i + 1, num_iter, loss.item()))
                optimizer.step()

        # Sometimes we get better performance on the GPU when we don't use Toeplitz math
        # for SKI. This flag controls that
        with gpytorch.settings.use_toeplitz(False):
            %time train()
```

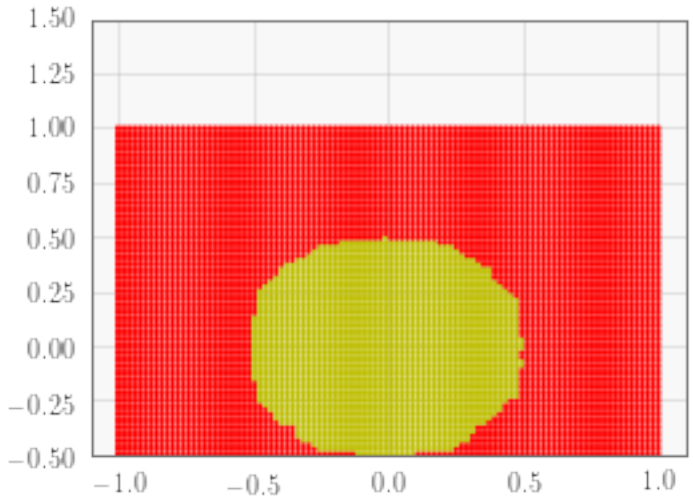
```
Iter 1/25 - Loss: 7.721
Iter 2/25 - Loss: 7.207
Iter 3/25 - Loss: 6.038
Iter 4/25 - Loss: 5.675
Iter 5/25 - Loss: 5.187
Iter 6/25 - Loss: 4.906
Iter 7/25 - Loss: 4.509
Iter 8/25 - Loss: 4.415
Iter 9/25 - Loss: 4.142
Iter 10/25 - Loss: 3.955
Iter 11/25 - Loss: 3.860
Iter 12/25 - Loss: 3.716
Iter 13/25 - Loss: 3.944
Iter 14/25 - Loss: 3.687
Iter 15/25 - Loss: 3.784
Iter 16/25 - Loss: 3.864
Iter 17/25 - Loss: 3.658
Iter 18/25 - Loss: 3.758
Iter 19/25 - Loss: 3.974
Iter 20/25 - Loss: 3.667
Iter 21/25 - Loss: 3.648
Iter 22/25 - Loss: 3.653
Iter 23/25 - Loss: 3.575
Iter 24/25 - Loss: 3.478
Iter 25/25 - Loss: 3.541
CPU times: user 7.08 s, sys: 424 ms, total: 7.51 s
Wall time: 7.5 s
```

10.1.5 Test the model

Next we test the model and plot the decision boundary. Despite the function we are optimizing not having an obvious additive decomposition, the model provides accurate results.

In [8]:
```
# Switch the model and likelihood into the evaluation mode
model.eval()
likelihood.eval()

# Start the plot, 4x3in
f, ax = plt.subplots(1, 1, figsize=(4, 3))

n = 100
test_x = torch.zeros(n ** 2, 2)
test_x[:, 0].copy_(torch.linspace(-1, 1, n).repeat(n))
test_x[:, 1].copy_(torch.linspace(-1, 1, n).unsqueeze(1).repeat(1, n).view(-1))
# Cuda variable of test data
test_x = test_x.cuda()

with torch.no_grad(), gpytorch.settings.use_toeplitz(False):
    predictions = likelihood(model(test_x))

# prob<0.5 --> label -1 // prob>0.5 --> label 1
pred_labels = predictions.mean.ge(0.5).float().mul(2).sub(1).cpu()
# Colors = yellow for 1, red for -1
color = []
for i in range(len(pred_labels)):
    if pred_labels[i] == 1:
        color.append('y')
    else:
        color.append('r')

# Plot data a scatter plot
ax.scatter(test_x[:, 0].cpu(), test_x[:, 1].cpu(), color=color, s=1)
ax.set_ylim([-0.5, 1.5])
```

Out[8]: (-0.5, 1.5)



In [ ]:

CHAPTER 11

Deep Kernel Learning (Regression + Classification)

11.1 Deep Kernel Learning (DenseNet + GP) on CIFAR10/100

In this notebook, we'll demonstrate the steps necessary to train a medium sized DenseNet (https://arxiv.org/abs/1608.06993) on either of two popularly used benchmark dataset in computer vision (CIFAR10 and CIFAR100). We'll be training the DKL model entirely end to end using the standard 300 Epoch training schedule and SGD.

This notebook is largely for tutorial purposes. If your goal is just to get (for example) a trained DKL + CIFAR100 model, we **recommend** that you move this code to a simple python script and run that, rather than training directly out of a python notebook. We find that training is just a bit faster out of a python notebook. We also of course recommend that you increase the size of the DenseNet used to a full sized model if you would like to achieve state of the art performance.

Furthermore, because this notebook involves training an actually reasonably large neural network, it is **strongly recommended** that you have a decent GPU available for this, as with all large deep learning models.

```
In [1]: from torch.optim import SGD, Adam
        from torch.optim.lr_scheduler import MultiStepLR
        import torch.nn.functional as F
        from torch import nn
        import torch
        import torchvision.datasets as dset
        import torchvision.transforms as transforms
        import gpytorch
        import math
```

11.1.1 Set up data augmentation

The first thing we'll do is set up some data augmentation transformations to use during training, as well as some basic normalization to use during both training and testing. We'll use random crops and flips to train the model, and do basic normalization at both training time and test time. To accomplish these transformations, we use standard `torchvision` transforms.

```
In [2]: normalize = transforms.Normalize(mean=[0.5071, 0.4867, 0.4408], std=[0.2675, 0.2565, 0.2761])
        aug_trans = [transforms.RandomCrop(32, padding=4), transforms.RandomHorizontalFlip()]
        common_trans = [transforms.ToTensor(), normalize]
        train_compose = transforms.Compose(aug_trans + common_trans)
        test_compose = transforms.Compose(common_trans)
```

11.1.2 Create DataLoaders

Next, we create dataloaders for the selected dataset using the built in torchvision datasets. The cell below will download either the cifar10 or cifar100 dataset, depending on which choice is made. The default here is cifar10, however training is just as fast on either dataset.

After downloading the datasets, we create standard `torch.utils.data.DataLoaders` for each dataset that we will be using to get minibatches of augmented data.

```
In [3]: dataset = 'cifar10'

        if dataset == 'cifar10':
            d_func = dset.CIFAR10
            train_set = dset.CIFAR10('data', train=True, transform=train_compose, download=True)
            test_set = dset.CIFAR10('data', train=False, transform=test_compose)
            train_loader = torch.utils.data.DataLoader(train_set, batch_size=256, shuffle=True)
            test_loader = torch.utils.data.DataLoader(test_set, batch_size=256, shuffle=False)
            num_classes = 10
        elif dataset == 'cifar100':
            d_func = dset.CIFAR100
            train_set = dset.CIFAR100('data', train=True, transform=train_compose, download=True)
            test_set = dset.CIFAR100('data', train=False, transform=test_compose)
            train_loader = torch.utils.data.DataLoader(train_set, batch_size=256, shuffle=True)
            test_loader = torch.utils.data.DataLoader(test_set, batch_size=256, shuffle=False)
            num_classes = 100
        else:
            raise RuntimeError('dataset must be one of "cifar100" or "cifar10"')
```

```
Files already downloaded and verified
```

11.1.3 Creating the DenseNet Model

With the data loaded, we can move on to defining our DKL model. A DKL model consists of three components: the neural network, the Gaussian process layer used after the neural network, and the Softmax likelihood.

The first step is defining the neural network architecture. To do this, we use a slightly modified version of the DenseNet available in the standard PyTorch package. Specifically, we modify it to remove the softmax layer, since we'll only be needing the final features extracted from the neural network.

```
In [4]: from densenet import DenseNet

        class DenseNetFeatureExtractor(DenseNet):
            def forward(self, x):
                features = self.features(x)
                out = F.relu(features, inplace=True)
                out = F.avg_pool2d(out, kernel_size=self.avgpool_size).view(features.size(0), -1)
                return out

        feature_extractor = DenseNetFeatureExtractor(block_config=(6, 6, 6), num_classes=num_classes)
        num_features = feature_extractor.classifier.in_features
```

11.1.4 Creating the GP Layer

In the next cell, we create the layer of Gaussian process models that are called after the neural network. In this case, we'll be using one GP per feature, as in the SV-DKL paper. The outputs of these Gaussian processes will the be mixed in the softmax likelihood.

```
In [5]: class GaussianProcessLayer(gpytorch.models.AdditiveGridInducingVariationalGP):
            def __init__(self, num_dim, grid_bounds=(-10., 10.), grid_size=64):
                super(GaussianProcessLayer, self).__init__(grid_size=grid_size, grid_bounds=[grid_bou
                                                           num_dim=num_dim, mixing_params=False, sum_
                self.covar_module = gpytorch.kernels.ScaleKernel(
                    gpytorch.kernels.RBFKernel(
                        log_lengthscale_prior=gpytorch.priors.SmoothedBoxPrior(
                            math.exp(-1), math.exp(1), sigma=0.1, log_transform=True
                        )
                    )
                )
                self.mean_module = gpytorch.means.ConstantMean()
                self.grid_bounds = grid_bounds

            def forward(self, x):
                mean = self.mean_module(x)
                covar = self.covar_module(x)
                return gpytorch.distributions.MultivariateNormal(mean, covar)
```

11.1.5 Creating the DKL Model

With both the DenseNet feature extractor and GP layer defined, we can put them together in a single module that simply calls one and then the other, much like building any Sequential neural network in PyTorch. This completes defining our DKL model.

```
In [6]: class DKLModel(gpytorch.Module):
            def __init__(self, feature_extractor, num_dim, grid_bounds=(-10., 10.)):
                super(DKLModel, self).__init__()
                self.feature_extractor = feature_extractor
                self.gp_layer = GaussianProcessLayer(num_dim=num_dim, grid_bounds=grid_bounds)
                self.grid_bounds = grid_bounds
                self.num_dim = num_dim

            def forward(self, x):
                features = self.feature_extractor(x)
                features = gpytorch.utils.grid.scale_to_bounds(features, self.grid_bounds[0], self.gr
                res = self.gp_layer(features)
                return res

        model = DKLModel(feature_extractor, num_dim=num_features).cuda()
        likelihood = gpytorch.likelihoods.SoftmaxLikelihood(num_features=model.num_dim, n_classes=nur
```

11.1.6 Defining Training and Testing Code

Next, we define the basic optimization loop and testing code. This code is entirely analogous to the standard PyTorch training loop. We create a `torch.optim.SGD` optimizer with the parameters of the neural network on which we apply the standard amount of weight decay suggested from the paper, the parameters of the Gaussian process (from which we omit weight decay, as L2 regualrization on top of variational inference is not necessary), and the mixing parameters of the Softmax likelihood.

We use the standard learning rate schedule from the paper, where we decrease the learning rate by a factor of ten 50% of the way through training, and again at 75% of the way through training.

```
In [7]: n_epochs = 300
        lr = 0.1
        optimizer = SGD([
            {'params': model.feature_extractor.parameters()},
            {'params': model.gp_layer.hyperparameters(), 'lr': lr * 0.01},
            {'params': model.gp_layer.variational_parameters()},
            {'params': likelihood.parameters()},
        ], lr=lr, momentum=0.9, nesterov=True, weight_decay=0)
        scheduler = MultiStepLR(optimizer, milestones=[0.5 * n_epochs, 0.75 * n_epochs], gamma=0.1)

        def train(epoch):
            model.train()
            likelihood.train()

            mll = gpytorch.mlls.VariationalMarginalLogLikelihood(likelihood, model, num_data=len(trai

            train_loss = 0.
            for batch_idx, (data, target) in enumerate(train_loader):
                data, target = data.cuda(), target.cuda()
                optimizer.zero_grad()
                output = model(data)
                loss = -mll(output, target)
                loss.backward()
                optimizer.step()
                if (batch_idx + 1) % 25 == 0:
                    print('Train Epoch: %d [%03d/%03d], Loss: %.6f' % (epoch, batch_idx + 1, len(trai

        def test():
            model.eval()
            likelihood.eval()

            correct = 0
            for data, target in test_loader:
                data, target = data.cuda(), target.cuda()
                with torch.no_grad():
                    output = likelihood(model(data))
                    pred = output.probs.argmax(1)
                    correct += pred.eq(target.view_as(pred)).cpu().sum()
            print('Test set: Accuracy: {}/{} ({}%)'.format(
                correct, len(test_loader.dataset), 100. * correct / float(len(test_loader.dataset))
            ))
```

11.1.7 Train the Model

We are now ready to train the model. At the end of each Epoch we report the current test loss and accuracy, and we save a checkpoint model out to a file.

```
In [8]: for epoch in range(1, n_epochs + 1):
            scheduler.step()
            with gpytorch.settings.use_toeplitz(False), gpytorch.settings.max_preconditioner_size(0):
                train(epoch)
                test()
            state_dict = model.state_dict()
            likelihood_state_dict = likelihood.state_dict()
            torch.save({'model': state_dict, 'likelihood': likelihood_state_dict}, 'dkl_cifar_checkpo
```

```
Train Epoch: 1 [025/196], Loss: 8.763647
Train Epoch: 1 [050/196], Loss: 8.670084
Train Epoch: 1 [075/196], Loss: 8.230519
Train Epoch: 1 [100/196], Loss: 7.313843
Train Epoch: 1 [125/196], Loss: 5.461603
Train Epoch: 1 [150/196], Loss: 3.573858
Train Epoch: 1 [175/196], Loss: 3.570160
Test set: Accuracy: 1589/10000 (15%)
Train Epoch: 2 [025/196], Loss: 3.339768
Train Epoch: 2 [050/196], Loss: 3.212200
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-8-30679d80b21c> in <module>()
      2     scheduler.step()
      3     with gpytorch.settings.use_toeplitz(False), gpytorch.settings.max_preconditioner_size(0)
----> 4         train(epoch)
      5         test()
      6     state_dict = model.state_dict()

<ipython-input-7-58b0d751f3e2> in train(epoch)
     20         optimizer.zero_grad()
     21         output = model(data)
---> 22         loss = -mll(output, target)
     23         loss.backward()
     24         optimizer.step()

~/Dropbox/workspace/gpytorch/gpytorch/module.py in __call__(self, *inputs, **kwargs)
    179
    180     def __call__(self, *inputs, **kwargs):
--> 181         outputs = self.forward(*inputs, **kwargs)
    182
    183         if isinstance(outputs, tuple):

~/Dropbox/workspace/gpytorch/gpytorch/mlls/variational_marginal_log_likelihood.py in forward(self, ou
     28         log_likelihood = self.likelihood.variational_log_probability(output, target, **kwargs
     29         kl_divergence = sum(
---> 30             variational_strategy.kl_divergence().sum() for variational_strategy in self.model
     31         ).div(self.num_data)
     32

~/Dropbox/workspace/gpytorch/gpytorch/mlls/variational_marginal_log_likelihood.py in <genexpr>(.0)
     28         log_likelihood = self.likelihood.variational_log_probability(output, target, **kwargs
     29         kl_divergence = sum(
---> 30             variational_strategy.kl_divergence().sum() for variational_strategy in self.model
     31         ).div(self.num_data)
     32

~/Dropbox/workspace/gpytorch/gpytorch/variational/mvn_variational_strategy.py in kl_divergence(self)
     25         log_det_variational_covar = variational_covar.log_det()
     26         trace_plus_inv_quad_form, log_det_prior_covar = prior_covar.inv_quad_log_det(
---> 27             inv_quad_rhs=inv_quad_rhs, log_det=True
     28         )
     29

~/Dropbox/workspace/gpytorch/gpytorch/lazy/lazy_tensor.py in inv_quad_log_det(self, inv_quad_rhs, log
    635             preconditioner=self._preconditioner()[0],
    636             log_det_correction=self._preconditioner()[1],
--> 637         )(*args)     638         return res
```

```
    639

~/Dropbox/workspace/gpytorch/gpytorch/functions/_inv_quad_log_det.py in forward(self, *args)
    130             if self.batch_size is None:
    131                 t_mat = t_mat.unsqueeze(1)
--> 132             eigenvalues, eigenvectors = lanczos_tridiag_to_diag(t_mat)
    133             slq = StochasticLQ()
    134             matrix_size = rhs.size(-2)

~/Dropbox/workspace/gpytorch/gpytorch/utils/lanczos.py in lanczos_tridiag_to_diag(t_mat)
    179     TODO: make the eigenvalue computations done in batch mode.
    180     """
--> 181     return batch_symeig(t_mat)

~/Dropbox/workspace/gpytorch/gpytorch/utils/eig.py in batch_symeig(mat)
     27             evals, evecs = mat[i, j].symeig(eigenvectors=True)
     28             mask = evals.ge(0)
---> 29             eigenvectors[i, j] = evecs * mask.type_as(evecs).unsqueeze(0)
     30             eigenvalues[i, j] = evals.masked_fill_(1 - mask, 1)
     31

KeyboardInterrupt:
```

In [ ]:

CHAPTER 12

gpytorch.models

12.1 Models for Exact GP Inference

12.1.1 ExactGP

12.2 Models for Variational GP Inference

12.2.1 VariationalGP

12.2.2 GridInducingVariationalGP

12.2.3 AdditiveGridInducingVariationalGP

CHAPTER 13

gpytorch.likelihoods

13.1 Likelihood

13.2 Standard Likelihoods

13.2.1 GaussianLikelihood

13.2.2 BernoulliLikelihood

13.3 Specialty Likelihoods

13.3.1 MultitaskGaussianLikelihood

13.3.2 SoftmaxLikelihood

CHAPTER 14

gpytorch.kernels

If you don't know what kernel to use, we recommend that you start out with a `gpytorch.kernels.ScaleKernel(gpytorch.kernels.RBFKernel)`.

14.1 Kernel

14.2 Standard Kernels

14.2.1 CosineKernel

14.2.2 LinearKernel

14.2.3 MaternKernel

14.2.4 PeriodicKernel

14.2.5 RBFKernel

14.2.6 SpectralMixtureKernel

14.2.7 WhiteNoiseKernel

14.3 Composition/Decoration Kernels

14.3.1 AdditiveKernel

14.3.2 ProductKernel

14.3.3 ScaleKernel

14.4 Specialty Kernels

14.4.1 IndexKernel

14.4.2 MultitaskKernel

14.5 Kernels for Scalable GP Regression Methods

14.5.1 AdditiveGridInterpolationKernel

14.5.2 GridKernel

14.5.3 GridInterpolationKernel

14.5.4 InducingPointKernel

14.5.5 MultiplicativeGridInterpolationKernel

CHAPTER 15

gpytorch.means

15.1 Mean

15.2 Standard Means

15.2.1 ZeroMean

15.2.2 ConstantMean

15.3 Specialty Means

15.3.1 MultitaskMean

CHAPTER 16

gpytorch.mlls

16.1 Marginal Log Likelihood

16.2 Exact GP Inference

16.2.1 ExactMarginalLogLikelihood

16.3 Variational GP Inference

16.3.1 VariationalMarginalLogLikelihood

CHAPTER 17

gpytorch.random_variables

17.1 RandomVariable

17.2 GaussianRandomVariable

17.3 MultitaskGaussianRandomVariable

CHAPTER 18

gpytorch.priors

18.1 Prior

18.2 Standard Priors

18.2.1 GammaPrior

18.2.2 InverseWishartPrior

18.2.3 LKJCovariancePrior

18.2.4 MultivariateNormalPrior

18.2.5 NormalPrior

18.2.6 SmoothedBoxPrior

18.2.7 WishartPrior

CHAPTER 19

gpytorch.settings

CHAPTER 20

gpytorch.beta_features

CHAPTER 21

gpytorch.Module

CHAPTER 22

gpytorch.lazy

22.1 LazyTensor

22.2 Kernel LazyTensors

22.3 Structured LazyTensors

22.3.1 BlockDiagLazyTensor

22.3.2 CholLazyTensor

22.3.3 DiagLazyTensor

22.3.4 MatmulLazyTensor

22.3.5 RootLazyTensor

22.3.6 NonLazyTensor

22.3.7 ToeplitzLazyTensor

22.3.8 ZeroLazyTensor

22.4 Composition/Decoration LazyTensors

22.4.1 AddedDiagLazyTensor

22.4.2 ConstantMulLazyTensor

22.4.3 InterpolatedLazyTensor

22.4.4 KroneckerProductLazyTensor

CHAPTER 23

gpytorch.functions

23.1 Functions

CHAPTER 24

gpytorch.utils

24.1 Utilities

24.1.1 Lanczos Utilities

24.1.2 Pivoted Cholesky Utilities

24.1.3 Sparse Utilities

CHAPTER 25

Indices and tables

- genindex
- modindex
- search

CHAPTER 26

Research references

- Gardner, Jacob R., Geoff Pleiss, David Bindel, Kilian Q. Weinberger, and Andrew Gordon Wilson. " GPyTorch: Blackbox Matrix-Matrix Gaussian Process Inference with GPU Acceleration." In NIPS (2018).
- Pleiss, Geoff, Jacob R. Gardner, Kilian Q. Weinberger, and Andrew Gordon Wilson. "Constant-Time Predictive Distributions for Gaussian Processes." In ICML (2018).
- Gardner, Jacob R., Geoff Pleiss, Ruihan Wu, Kilian Q. Weinberger, and Andrew Gordon Wilson. "Product Kernel Interpolation for Scalable Gaussian Processes." In AISTATS (2018).
- Wilson, Andrew G., Zhiting Hu, Ruslan R. Salakhutdinov, and Eric P. Xing. "Stochastic variational deep kernel learning." In NIPS (2016).
- Wilson, Andrew, and Hannes Nickisch. "Kernel interpolation for scalable structured Gaussian processes (KISS-GP)." In ICML (2015).
- Hensman, James, Alexander G. de G. Matthews, and Zoubin Ghahramani. "Scalable variational Gaussian process classification." In AISTATS (2015).