AIMS CDT Signal Processing Lab Session 4

Gaussian Processes on Graphs

Xiaowen Dong (xiaowen@robots.ox.ac.uk), Yin-Cong Zhi (yin-cong.zhi@eng.ox.ac.uk)

Introduction

Acknowledgement: This lab session was first created by Yin-Cong Zhi in the Machine Learning Research Group in the Department of Engineering Science at Oxford University.

In this lab, we will be looking into Gaussian processes (GPs) on graphs. GPs are Bayesian non-parametric models, and they have unique properties in being able to predict with uncertainty, and generally performing better when there are less training data. In terms of machine learning on graphs, GPs are effective alternatives to GNNs on smaller graphs and can sometimes outperform them.

GPs can be implemented using GPyTorch or Pyro. For graph-based GPs, we will however use a more convenient package called `gpflow`; this is a tensorflow based library, therefore this lab will be make use of Tensorflow where appropriate. Compared to GNNs, GPs are generally slower to train, so we recommend running this in Colab, and change runtime type to (T4) GPU.

GPflow: https://www.gpflow.org/
GPyTorch: https://gpytorch.ai/
Pyro for GPs: https://pyro.ai/examples/gp.html

```
!pip install gpflow
```

```
!python -c "import torch; print(torch.__version__)"
```

```
!pip install pyg_lib torch_scatter torch_sparse torch_cluster torch_spline_conv torch_geometric -f https://data.pyg.org/whl/torch-2.0.0+cu118.html
```

```
# Import required packages
import numpy as np
import matplotlib as plt
import torch
import numpy as np
import matplotlib as plt
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore")
import torch_geometric
from torch_geometric.datasets import Planetoid
from torch_geometric.transforms import ToUndirected
```

We will continue to work with the Cora citation dataset from Lab 1.

```
dataset = Planetoid(root='data', name='Cora', pre_transform=ToUndirected())
print('Feature dimension:', dataset.data.x.size(1))
print('Number of classes:', dataset.data.y.max()+1)
print('Number of classes:', dataset.data.y.max().numpy()+1)
```

```
# data loading
import tensorflow as tf
import gpflow
from gpflow.utilities import positive, print_summary

train_x = tf.convert_to_tensor(dataset.data.x[dataset.train_mask], tf.float64)
train_y = tf.convert_to_tensor(dataset.data.y[dataset.train_mask])
val_x = tf.convert_to_tensor(dataset.data.x[dataset.val_mask], tf.float64)
val_y = dataset.data.y.float()[dataset.val_mask]
test_x = tf.convert_to_tensor(dataset.data.x[dataset.test_mask], tf.float64)
test_y = dataset.data.y.float()[dataset.test_mask]
num_classes = 7
```

```
# graph construction
from torch_geometric.utils import to_networkx
A = to_networkx(dataset.data, to_undirected=True)  # to_undirected=True
A = tf.convert_to_tensor(nx.to_numpy_array(A), dtype=float)
```

A Brief Overview of Gaussian Processes

A GP is a collection of random variables any finite subset of which has a joint Gaussian distribution. It can be conceptually viewed as an infinite dimensional function, and when defined on any finite set of points these will have a multivariate Gaussian distribution. Assuming a Gaussian process on some input data $\mathbf{x}_1, \ldots, \mathbf{x}_N$ is equivalent to setting a Gaussian prior on the data, such that $\mathbf{f} = [f(\mathbf{x}_1), \ldots, f(\mathbf{x}_N)]^\top$ has the following distribution

$$\mathbf{f} \sim \mathcal{N}(0, \mathbf{K}),$$

with the predictive model taking the form:

$$y_i = f(\mathbf{x}_i) + \epsilon_i$$

The key element is the covariance matrix $\mathbf{K} \in \mathbb{R}^{N \times N}$ with $\mathbf{K}_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$ specified by a positive-definite kernel function $k(\cdot, \cdot)$. Choosing the kernel is therefore the most central element of model selection, as this fully specifies the prior and how the model makes prediction on unknown points. A common choice (on non-graph data) is the squared exponential (RBF) kernel:

$$k_{\text{RBF}}(\mathbf{x}, \mathbf{x}') = \sigma^2 \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\ell^2}\right),$$

where σ^2 is the variance to fit towards the magnitude of the data, ℓ is the lengthscale controlling the similarity between points, the interpretation being the larger lengthscale the more similar the predictions $f(\mathbf{x})$ and $f(\mathbf{x}')$ will be (and vice versa).

In building a GP on graphs, we will need to implement a kernel on graphs (more later).

In Bayesian modelling, we predict with a distribution. Let $f_* = f(\mathbf{x}_{new})$ denote our model on a novel test point, then we are interested in obtaining the predictive distribution

$$P(f_* | \mathcal{D}) = \int P(f_* | f) P(f | \mathcal{D}) \, \mathrm{d}f$$

with

$$P(\mathcal{D} | f) \propto P(\mathcal{D} | \mathbf{f}) P(\mathbf{f}),$$

with $P(\mathbf{f})$ being our chosen prior.

In the regression case, we assume a Gaussian likelihood

$$P(\mathcal{D} | \mathbf{f}) = \prod_{i=1}^{N} \mathcal{N}(y_i | f(\mathbf{x}_i), \sigma^2)$$

and everything in the integral is tractable, leading to a closed-form solution to the predictive distribution

$$P(f_* | \mathcal{D}) = \mathcal{N}(m_*, \Sigma_*)$$

with

$$\begin{aligned} m_* &= \mathbf{K}_{*\mathbf{f}} (\mathbf{K} + \sigma \mathbf{I})^{-1} \mathbf{y} \\ \Sigma_* &= \mathbf{K}_{**} - \mathbf{K}_{*\mathbf{f}} (\mathbf{K} + \sigma^2 \mathbf{I})^{-1} \mathbf{K}_{\mathbf{f}*} \end{aligned}$$

Here μ_* is used for a point prediction, and Σ_* is used to produce confidence intervals (the detailed derivation is omitted here but can be found in various textbooks and online resources).

In this lab, we are using GPs for multiclass classification, which assumes a more complex likelihood. This is explained as follows. Nodes in the Cora graph take 1 of 7 possible classes (from 0 to 6). For GPs, the labels are one-hot encoded: for instance if $y = 2$, we convert this into $(0, 0, 1, 0, 0, 0, 0)^\top$. We then use 7 independent GPs to predict the likelihood of being in each of the 7 classes, combining them with the following density

$$\kappa_i(\mathbf{f}) = \begin{cases} 1 - \epsilon, & \text{if } i = \arg\max_j f_j \\ \frac{\epsilon}{C-1}, & \text{otherwise} \end{cases}$$

for $\mathbf{f} = \{\mathbf{f}_1, \ldots, \mathbf{f}_C\}$, where ϵ is taken to be a small value (usually 10^{-3}). The likelihood $P(\mathcal{E} | \mathbf{f})$ is then written as

$$\prod_{i=1}^{N} \prod_{j=1}^{C} \pi(C_{ij}) \kappa_j(\mathbf{f}^i.$$

This is called the robust maximum likelihood function. It is used in Gaussian processes instead of the softmax for numerical stability, and it also performs better generally.

This, however, makes computing the predictive distribution $P(f_* | \mathcal{E})$ intractable as the integral involves non-Gaussian terms, leading to the need for variational inference. Variational inference uses surrogate GPs - $q(\mathbf{f})$ - to approximate the shape of the non-Gaussian parts of the integral - $P(\mathbf{f} | \mathcal{E})$. In this case we need 7 separate surrogates for each class. Let $q(\mathbf{f}) = \{q_1(\mathbf{f}_1), \ldots, q_C(\mathbf{f}_C)\}$ consists of the 7 GPs, each with its own mean and covariance

$$q_c(\mathbf{f}_c) = \mathcal{N}(\mathbf{m}_c, \mathbf{S}_c).$$

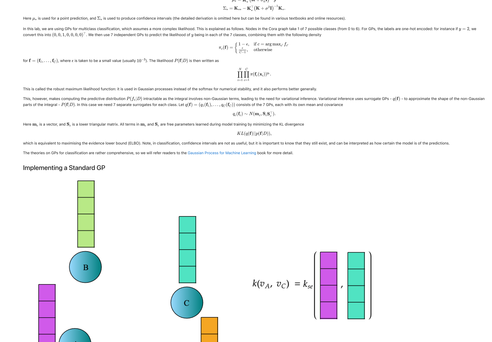
Here $\mathbf{S}_c$ is a vector, and $\mathbf{S}_c$ is a lower triangular matrix. All terms in $\mathbf{m}_c$ and $\mathbf{S}_c$ are free parameters learned during model training by minimizing the KL-divergence

$$KL(q(\mathbf{f}) \| P(\mathbf{f} | \mathcal{E})),$$

which is equivalent to maximising the evidence lower bound (ELBO). Note, in classification, confidence intervals are not as useful, but it is important to know that they still exist, and can be interpreted as how certain the model is of the predictions.

The theories on GPs for classification are rather comprehensive, so we will refer readers to the Gaussian Process for Machine Learning book for more detail.

Implementing a Standard GP



We will start by building a GP that train on just the node features, ignoring the graph. First thing to do when specifying a GP is to choose a kernel function, by specifying the prior in `gpflow`.

```
kernel = gpflow.kernels.SquaredExponential(lengthscales = 1., variance = 1.)
```

Other options include

- `gpflow.kernels.Polynomial(offset = 1., variance = 1.)`
- `gpflow.kernels.Matern12(lengthscales = 1., variance = 1.)`
- `gpflow.kernels.Matern32(lengthscales = 1., variance = 1.)`
- `gpflow.kernels.Matern52(lengthscales = 1., variance = 1.)`

`gpflow` also implements the robust maximum likelihood function.

```
likelihood = gpflow.likelihoods.RobustMax(num_classes)
likelihood = gpflow.likelihoods.MultiClass(
    num_classes, invlink=invlink
)
```

Here, we will build a simple variational GP (VGP). Note, `gpflow` will automatically one-hot encode the labels, but you need to specify the number of latent (surrogate) GPs for the variational distribution (`num_latent_gps`) corresponding to the number of classes. All variational distributions and priorities are implemented within `gpflow`.

```
data = (train_x, train_y.reshape(-1,1))
model = gpflow.models.VGP(
    data,
    kernel=kernel,
    likelihood=likelihood,
    num_latent_gps=num_classes
)

print_summary(model, fmt="notebook")
```

During model training, to inspect the model at each epoch `gpflow` utilizes a `step_callback` function, and the one that we will use for this lab is provided below. Like Lab 3, we will mainly evaluate on the validation set.

```
def step_callback(step, variables=None, values=None):
    pred = model.predict_y(val_x)[0]
    pred_class = tf.argmax(pred, axis=1)
    val_acc = tf.reduce_sum(tf.cast(pred_class == val_y, tf.float64)).numpy()/len(val_y)
    print("Epoch {}/{}: ELBO (model.elbo(): {}, Val Acc = {}".format(
    step, maxiter, model.elbo(), val_acc))
    if step % 10 == 0:
        print_summary(model, fmt="notebook")
```

Model training involves learning the optimal kernel parameters and variational distribution parameters, which are collectively called GP hyperparameters.

```
def optimize_tf(model, step_callback, lr=0.1, num_epochs=100):
    opt = tf.optimizers.Adam(learning_rate=lr)
    for epoch_no in range(num_epochs):
        with tf.GradientTape(watch_accessed_variables=False) as tape:
            tape.watch(model.trainable_variables)
            loss = model.training_loss()
            grads = tape.gradient(loss, model.trainable_variables)
        opt.apply_gradients(zip(grads, model.trainable_variables))
        step_callback(epoch_no)

optimize_tf(model, step_callback, lr=0.1, num_epochs=100)
```

```
# final prediction
pred = model.predict_y(val_x)
pred_class = tf.argmax(pred, axis=1)
tf.reduce_sum(tf.cast(pred_class == val_y, tf.float64)).numpy()/len(val_y)
```

- Can you describe what each component of the model summary does?
- Inspecting `.prod`, why are there 2 lengths and what are their functionalities? (Note, in classification, the second lenscale is not used because it is not as meaningful, but it is still good for you to know of its existence)

A Graph Only GP

A graph GP will require a kernel that measures node similarity based on the graph. This is trickier because nodes also contain features, and we need to combine that with the graph structure. The first step is to implement a kernel that only uses the graph information, and measures node similarity only based on graph connectivity.

A reminder: kernels on graph are built using the graph Laplacian $\mathbf{g}(\mathbf{L})$, and each kernel is defined by a function on the eigenvalues of $\mathbf{L}$ as:

$$\mathbf{U} \mathbf{g}(\mathbf{\Lambda}) \mathbf{U}^\top$$

For some g, this can be expressed as a function on the Laplacian directly $\mathbf{g}(\mathbf{L})$: in this lab, we will make use of one such function that shares the same structure as spectral filtering on graphs (also called regularized Laplacian):

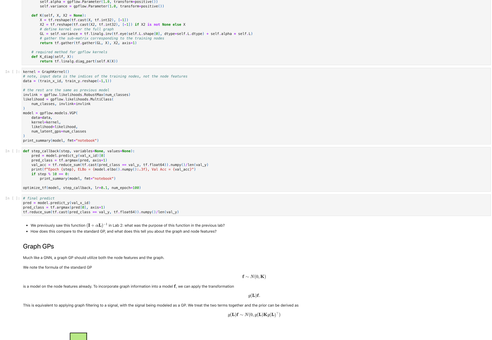
$$\mathbf{K}_\sigma = \sigma^2 \mathbf{U} \text{diag}([1 + \sigma\lambda_1, \ldots, 1/(1 + \sigma\lambda_N)] \mathbf{U}^\top = \sigma^2 (\mathbf{I} + \sigma \mathbf{L})^{-1}.$$

The addition of the variance term σ^2 is to allow the model to fit towards the magnitude of the data and is a necessary parameterization for all GP kernels. It will be optimized as a model parameter along with σ.

Other options:

- $\exp(-\sigma \tilde{\mathbf{L}}), \sigma \geq 0$ - Diffusion
- $(a\mathbf{I} - \tilde{\mathbf{L}})^p, a \geq 3$ - p-step random walk
- $(\cos(\frac{\pi}{4}\tilde{\mathbf{L}}))^p$
- https://arxiv.org/pdf/2012.11964.pdf

Note, to compute the similarity between any two nodes i and j, we need to compute this kernel for the whole graph before taking the ij-th element of $\mathbf{K}_\sigma$.



This kernel follows a slightly different implementation. Recall we are not using node features, so nodes are identified by their indices, and selecting the covariance between only the training nodes is carried out using the `gather` function in tensorflow. The input data to the model is therefore the indices of the training nodes, where the labels are the same as before.

```
# defining the inputs for graph kernel = indices of the nodes
train_inputs = tf.cast(tf.where(dataset.val_mask == True), tf.float64)
val_x_id = tf.cast(tf.where(dataset.val_mask == True), tf.float64)
test_x_id = tf.cast(tf.where(dataset.test_mask == True), tf.float64)
```

```
class GraphKernel(gpflow.kernels.Kernel):
    def __init__(self,
                 laplacian_function_place=None):
        self.g = tf.convert_to_tensor(1., dtype=tf.float64)
        self.sigma = gpflow.Parameter(1.0, transform=positive())
        self.variance = gpflow.Parameter(1.0, transform=positive())

    def K(self, X, X2 = None):
        X = tf.cast(tf.squeeze(X, 1), dtype=int32)
        X2 = tf.reshape(tf.cast(X2, tf.int32), [-1]) if X2 is not None else X
        # define kernel over the full graph
        base_kernel = self.base_kernel(self.sigma*tf.eye(A.shape[0], dtype=self.g.dtype) + self.alpha * self.L)
        # gather the submatrix corresponding to the training nodes
        return tf.gather(tf.gather(base_kernel, X2, axis=1), X, axis=0)

    # required method for gpflow kernels
    def K_diag(self, X):
        return tf.linalg.diag_part(self.K(X))
```

```
kernel = GraphKernel()
# data = indices of the indices of the training nodes, not the node features
data = (train_x_id, train_y.reshape(-1,1))
```

```
# the rest are the same as previous model
likelihood = gpflow.likelihoods.RobustMax(num_classes)
likelihood = gpflow.likelihoods.MultiClass(
    num_classes, invlink=invlink
)

model = gpflow.models.VGP(
    data,
    kernel=kernel,
    likelihood=likelihood,
    num_latent_gps=num_classes
)

print_summary(model, fmt="notebook")
```

```
# here we will evaluate on both the validation and test sets
def step_callback(step, variables=None, values=None):
    pred = model.predict_y(train_x_id)[0]
    pred_class = tf.argmax(pred, axis=1)
    val_acc = tf.reduce_sum(tf.cast(pred_class == val_y, tf.float64)).numpy()/len(val_y)
    test_pred = model.predict_y(test_x_id)[0]
    test_class = tf.argmax(test_pred, axis=1)
    test_acc = tf.reduce_sum(tf.cast(test_class == test_y, tf.float64)).numpy()/len(test_y)
    print("Epoch {}/{}: ELBO (model.elbo(): {}, Val Acc = {}, Test Acc = {}".format(
    step, maxiter, model.elbo(), val_acc, test_acc))
    if step % 10 == 0:
        print_summary(model, fmt="notebook")

optimize_tf(model, step_callback, lr=0.1, num_epochs=100)
```

```
# final prediction
pred = model.predict_y(val_x_id)
pred_class = tf.argmax(pred, axis=1)
print("Val Acc = {}".format(tf.reduce_sum(tf.cast(pred_class == val_y, tf.float64)).numpy()/len(val_y)))

pred = model.predict_y(test_x_id)
pred_class = tf.argmax(pred, axis=1)
print("Test Acc = {}".format(tf.reduce_sum(tf.cast(pred_class == test_y, tf.float64)).numpy()/len(test_y)))
```

- How do the 3 GPs in this example compare to the three models introduced in Lab 3? Which model performed the best and why?
- Can you interpret what $\mathbf{g}(\mathbf{L})\mathbf{k}$ is doing? What's the connection between signal processing and kernel methods?

```
# your data = (x, y) should have dimensions (: (64k)(), y: (64k))
```

(Optional) If you are interested in more flexible GPs for node classification on graphs, you can look into the extension of spectral graph wavelets here, which introduces kernels that consists of band-pass (so typical in wavelets) along with the low-pass we are familiar with. This can be beneficial for handling more complex data and classification tasks. The work also includes methods of scalability to avoid the cubic complexity of computing the graph Fourier basis/kernels.

[Conclusion] You are now done with this lab session. Please ask a lab demonstrator to evaluate your work.