CS-E4820 Machine Learning: Advanced Probabilistic Methods (spring 2021)

Pekka Marttinen, Santosh Hiremath, Tianyu Cui, Yogesh Kumar, Zheyang Shen, Alexander Aushev, Khaoula El Mekkaoui, Shaoxiong Ji, Alexander Nikitin, Sebastiaan De Peuter, Joakim Järvinen.

Exercise 8, due on Tuesday March 30 at 23:00.

Problem 1: Minimize KL divergence using PyTorch

PyTorch is a powerful auto-differentiation framework that allows us to do any optimization, as long as we can define the objective function and corresponding optimization variables. It has been widely used for Bayesian deep learning. In this exercise, we will study how to use PyTorch to fit a Gaussian distribution to a known Mixture of Gaussian by minimizing their KL divergence, and compare the difference between the forward and reverse form of the KL.

Recall that the KL divergence between two distributions $q(x)$ and $p(x)$ is defined as:

$$\mathrm{KL}[q(x)|p(x)] = \int q(x) \log \frac{q(x)}{p(x)} dx.$$

This is typically called the **Reverse KL** which we have used before in the course (like in Variational Bayes). If the probability density functions of $q(x)$ and $p(x)$ are known, and we can get samples from $q(x)$, an unbiased estimator of KL divergence is:

$$\mathrm{KL}[q(x)|p(x)] \approx \log \frac{q(x_i)}{p(x_i)} = \log q(x_i) - \log p(x_i),$$

where $x_i \sim q(x)$. We will use above estimator for this exercise.

There is also a **Forward KL**: $\mathrm{KL}[p(x)|q(x)]$ defined as:

$$\mathrm{KL}[p(x)|q(x)] = \int p(x) \log \frac{p(x)}{q(x)} dx,$$

which is used in other inference algorithms such as Expectration Propogation which is not within the scope of this course.

Let $p(x \mid \pi) = \pi\mathcal{N}(0,1) + (1-\pi)\mathcal{N}(8,1)$ where $\pi \sim \mathrm{Bernoulli}(0.4)$ be the true mixture distribution which we want to fit using a Gaussian $q(x;\mu,\sigma)$. We want to estimate μ and σ using both the forwared and reverse KL.

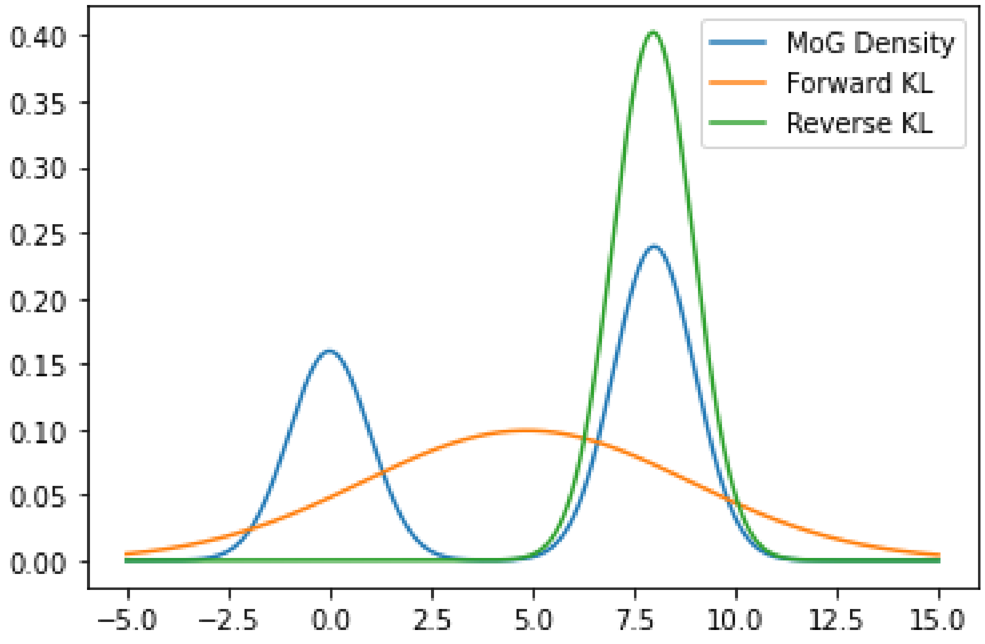
Complete the template below with the relevant code.

[1]:
```
import numpy as np
import math
import torch
import torch.nn as nn
import torch.optim as optim
import torch.distributions as Dis
import matplotlib
import matplotlib.pyplot as plt

class Gaussian:
"""
    This represents q(x)
    Gaussian distribution is parametrized by mean (mu) and standard␣
↪deviation. The standard deviation is
    parametrized as sigma = log(1 + exp(rho)) to make it positive all␣
↪the time. A sample from the distribution
    can be obtained by first sampling from a unit Gaussian, shifting the␣
↪samples by the mean and scaling by the
    standard deviation: w = mu + log(1 + exp(rho)) * epsilon
    """
def __init__(self, mu, rho):
self.mean = mu
self.rho = rho

@property
def std_dev(self):
return torch.log1p(torch.exp(self.rho))

def sample(self, num_samples = 1):
# Sample num_samples data points from Gaussian distribution
# Return a tensor contains all the samples

# Sample num_samples datapoints from N(0,1)
epsilon = Dis.Normal(0,1).sample([num_samples])

# Scale and shift epsilon
# samples = ? # EXERCISE

### BEGIN SOLUTION
samples = self.mean + self.std_dev * epsilon
### END SOLUTION

return samples

def logprob(self, samples):
# Compute the log probability of each sample under Gaussian distribution
# Return a tensor containing the log probability of all samples

# logp = ? # EXERCISE
### BEGIN SOLUTION
logp = -math.log(math.sqrt(2 * math.pi)) - torch.log(self.std_dev) -␣
↪((samples - self.mean) ** 2) / (2 * self.std_dev ** 2)
### END SOLUTION
```

```
        return logp

        class MoG:
        """
            This represents p(x).
            In this example, mixture of two Gaussian distribution is constructed␣
↪by 2 Gaussian distributions
            N(0,2) and N(8,1), and each datapoint is from N(0,2) with␣
↪probability p = 0.4 and from N(8,1) with
            probability 0.6.
            """
        def __init__(self, mu_1=0., sigma_1=1., mu_2=8., sigma_2=1., prob = 0.4):
        self.mean_1 = torch.tensor(mu_1)
        self.sigma_1 = torch.tensor(sigma_1)
        self.mean_2 = torch.tensor(mu_2)
        self.sigma_2 = torch.tensor(sigma_2)
        self.prob = torch.tensor(prob)

        def sample(self, num_samples = 1):
        # Sample num_samples data points from MoG distribution
        # Return a tensor contains all the samples

        # sample from N(0, 2)
        # sample form N(8, 1)
        # sample from Bern(0.4)
        # Combine the three to from a sample form mixture
        # sample_gaussian_1 = ? # EXERCISE
        # sample_gaussian_2 = ? # EXERCISE
        # sample_bernoulli = ? # EXERCISE
        # samples  = ? # EXERCISE

        ### BEGIN SOLUTION
        sample_gaussian_1 = Dis.Normal(self.mean_1, self.sigma_1).
↪sample([num_samples])
        sample_gaussian_2 = Dis.Normal(self.mean_2, self.sigma_2).
↪sample([num_samples])
        sample_bernoulli = Dis.Bernoulli(probs = self.prob).sample([num_samples])
        samples = sample_bernoulli * sample_gaussian_1 + (1. - sample_bernoulli)␣
↪* sample_gaussian_2
        ### END SOLUTION

        return samples

        def logprob(self, samples):

        # Compute the log probability of each sample under the MoG distribution
        # Return a tensor containing the log probability of all samples
        # logp = ? # EXERCISE

        # BEGIN SOLUTION
        prob_1 = torch.exp(-math.log(math.sqrt(2 * math.pi)) - torch.log(self.
↪sigma_1) - ((samples - self.mean_1) ** 2) / (2 * self.sigma_1 ** 2)) * self.
↪prob
        prob_2 = torch.exp(-math.log(math.sqrt(2 * math.pi)) - torch.log(self.
↪sigma_2) - ((samples - self.mean_2) ** 2) / (2 * self.sigma_2 ** 2)) * (1 -␣
↪self.prob)
        logp = torch.log(prob_1 + prob_2)
        ### END SOLUTION

        return logp

        class KL_divergence(nn.Module):
        def __init__(self):
        super(KL_divergence, self).__init__()
        # define the mean and standard deviation as parameters, and␣
↪initialization
        self.mu = nn.Parameter(torch.Tensor(1).uniform_(-2., 12.))
        self.rho = nn.Parameter(torch.Tensor(1).uniform_(1.0, 5.0))

        self.gaussian = Gaussian(self.mu, self.rho)
        self.mog = MoG()

        def compute_forwardKL(self):
        num_samples = torch.tensor(1000)

        # compute the forward KL divergence between p and q of num_samples data␣
↪points
        # Return the estimated forward KL divergence


        # sample form MoG
        # compute forware KL

        # samples = ? # EXERCISE
        # fkl = ? # EXERCISE

        ### BEGIN SOLUTION
        samples = self.mog.sample(num_samples)
        fkl = (self.mog.logprob(samples).sum() - self.gaussian.logprob(samples).
↪sum()) / num_samples
        ### END SOLUTION

        return fkl
```

```
def compute_reverseKL(self):
num_samples = torch.tensor(1000)
# compute the reverse KL divergence between p and q with num_samples␣
↪data points
# Return the estimated reverse KL divergence

# sample form Gaussian
# compute reverse KL

# samples = ? # EXERCISE
# rkl = ? # EXERCISE

### BEGIN SOLUTION
samples = self.gaussian.sample(num_samples)
rkl = (self.gaussian.logprob(samples).sum() - self.mog.logprob(samples).
↪sum()) / num_samples
### END SOLUTION
return rkl

# Optimize the KL by using gradient descent
def optimization(kl, forward = False, learning_rate = 0.1, num_epoch =␣
↪1000):
parameters = set(kl.parameters())
optimizer = optim.Adam(parameters, lr = learning_rate, eps=1e-3)

for epoch in range(num_epoch):
optimizer.zero_grad()
if forward:
loss = kl.compute_forwardKL()
else:
loss = kl.compute_reverseKL()

loss.backward()
optimizer.step()

if (epoch % 100) == 0:
print('EPOACH %d: KL: %.4f.'% (epoch+1, loss))

print('Optimizing reverse KL')
torch.manual_seed(0)
kl_reverse = KL_divergence()
optimization(kl_reverse, forward = False)
Gaussian_reverse= kl_reverse.gaussian

print('Optimizing forward KL')
kl_forward = KL_divergence()
```

```
        optimization(kl_forward, forward= True)
        Gaussian_forward = kl_forward.gaussian

        # Plot the pdf of Gaussian fitted from forward KL and reverse KL, and␣
↪also the ground truth pdf from MoG
        x_plot = torch.linspace(-5., 15., 1000)
        density_mog = torch.exp(kl_forward.mog.logprob(x_plot)).detach().numpy()
        density_Gaussian_forward = torch.exp(Gaussian_forward.logprob(x_plot)).
↪detach().numpy()
        density_Gaussian_reverse = torch.exp(Gaussian_reverse.logprob(x_plot)).
↪detach().numpy()

        fig, ax = plt.subplots()
        ax.plot(x_plot, density_mog)
        ax.plot(x_plot, density_Gaussian_forward)
        ax.plot(x_plot, density_Gaussian_reverse)

        ax.legend(('MoG Density','Forward KL', 'Reverse KL'))
```



Problem 2: VB for a factor analysis model (1/2)

The data set consists of D-dimensional vectors $x_n \in \mathbb{R}^D$, for $n = 1, \ldots, N$. We model the data using factor analysis with K-dimensional factors $z_n \in \mathbb{R}^K$. In detail, the model is specified as follows:

$$\begin{aligned}
\boldsymbol{x}_n &\sim \mathcal{N}_D(\boldsymbol{W}\boldsymbol{z}_n, \mathrm{diag}(\boldsymbol{\psi})^{-1}), \quad n = 1, \ldots, N, \\
\psi_d &\sim \mathrm{Gamma}(a, b), \quad d = 1, \ldots, D, \\
\boldsymbol{w}_d &\sim \mathcal{N}_K(\boldsymbol{0}, \alpha \boldsymbol{I}), \quad d = 1, \ldots, D, \\
\boldsymbol{z}_n &\sim \mathcal{N}_K(\boldsymbol{0}, \boldsymbol{I}), \quad n = 1, \ldots, N.
\end{aligned}$$

Here, $\boldsymbol{W}$ is a $D \times K$ factor loading matrix and $\boldsymbol{w}_d$ is the dth row of $\boldsymbol{W}$ written as a column vector. Parameter ψ_d^{-1} is the variance for the dth dimension in the observed data and $\mathrm{diag}(\psi)$ denotes a diagonal matrix with elements $\boldsymbol{\psi} = (\psi_1, \ldots, \psi_D)^T$ on the diagonal.

We approximate the posterior $p(\boldsymbol{\psi}, \boldsymbol{Z}, \boldsymbol{W}|\boldsymbol{X})$ using the mean-field approximation:

$$q(\Theta) = \prod_{d=1}^{D} q(\boldsymbol{w}_d) \prod_{n=1}^{N} q(\boldsymbol{z}_n) \prod_{d=1}^{D} q(\psi_d).$$

1 Write the logarithm of the joint distribution, $\log p(\boldsymbol{\psi}, \boldsymbol{Z}, \boldsymbol{W}, \boldsymbol{X})$.

2 Remove from the logarithm of the joint distribution all terms that do not depend on $\boldsymbol{z}_n$.

3 Show that the updated factor $q(\boldsymbol{z}_n)$ is equal to

$$q(\boldsymbol{z}_n) = \mathcal{N}_K(\boldsymbol{\mu}_n, \boldsymbol{K}_n),$$

where

$$\begin{aligned}
\boldsymbol{K}_n &= \left[I + \sum_{d=1}^{D} \langle \psi_d \rangle \left\langle \boldsymbol{w}_d \boldsymbol{w}_d^T \right\rangle \right]^{-1} \quad \text{and} \\
\boldsymbol{\mu}_n &= \boldsymbol{K}_n \left\langle \boldsymbol{W}^T \right\rangle \mathrm{diag}(\langle \boldsymbol{\psi} \rangle) \boldsymbol{x}_n.
\end{aligned}$$

Here $\langle \cdot \rangle$ is used as a shorthand for the expectation of a variable with respect to its factor, e.g., $\langle \boldsymbol{\psi} \rangle = \mathbb{E}_{q(\boldsymbol{\psi})}[\boldsymbol{\psi}]$ etc.

Hint 1: Try to write the log joint as

$$-\frac{1}{2} \boldsymbol{z}_n^T \boldsymbol{A} \boldsymbol{z}_n + \boldsymbol{b}^T \boldsymbol{z}_n$$

for some $\boldsymbol{A}$ and $\boldsymbol{b}$, after which you can apply the 'completing the square' technique.

Hint 2: Suppose $\boldsymbol{A}$ is an $N \times M$ matrix. Further suppose that $\boldsymbol{D}$ is an $N \times N$ diagonal matrix, $\boldsymbol{D}$ =diag$(d_1, \ldots, d_N)$. Then $\boldsymbol{A}^T \boldsymbol{D} \boldsymbol{A}$ can be written as

$$\boldsymbol{A}^T\, \boldsymbol{D} \boldsymbol{A} = \sum_{n=1}^{N} d_n \boldsymbol{a}_n \boldsymbol{a}_n^T,$$

where $\boldsymbol{a}_n$ is the nth row of $\boldsymbol{A}$ written as a column vector.

Hint 3: Recall that expectation is a linear operator, i.e. $\mathbb{E}(aX + bY) = a\mathbb{E}(X) + b\mathbb{E}(Y)$. Further, if some random variables A and B are independent, then $\mathbb{E}_{q(A)q(B)}(AB) = \mathbb{E}_{q(A)}(A)\mathbb{E}_{q(B)}(B)$.

Problem 3: VB for a factor analysis model (2/2)

For the factor analysis model considered in Problem 2, derive the update for factor $q(\boldsymbol{w}_d)$. The updated factor should be given in terms of the following expectations: $\langle \psi_d \rangle, \langle \boldsymbol{z}_n \rangle, \langle \boldsymbol{z}_n \boldsymbol{z}_n^T \rangle$, which have been calculated using the current values of the other factors for all d, n.

Hint: A multivariate Gaussian with a diagonal covariance can be expressed as a product of independent univariate Gaussians, which allows you to simplify the formulas.