Exercises 7

Message passing

Exercise 7.1
Consider the same problem as in Exercise 6.3.

a) Implement a program that computes $p(x|y=1)$ using message-passing with moment matching.

b) Assume that the prior instead is $x \sim \mathrm{U}[-1,1]$ where $\mathrm{U}[a,b]$ is the uniform distribution defined as

$$\mathrm{U}(x; a, b) = \begin{cases} \frac{1}{b-a} & \text{if} \quad a < x < b, \\ 0 & \text{otherwise} \end{cases}$$

Implement a program that computes $p(x|y=1)$ with this prior musing message passing and moment matching.

Exercise 7.2
Consider the same problem as in Exercise 6.3; this time, however, imagine that you have two t, say t_1 and t_2, each with a different observation y, say y_1 and y_2.

a) Draw a factor graph of the model with the two measurements and identify the messages required to compute $p(x|y_1=1, y_2=1)$.

b) Implement a program that computes $p(x|y_1=1, y_2=1)$ using message-passing with moment matching.

c) Implement a program that computes $p(x|y_1=1, y_2=1)$ using importance sampling.

Solutions 7

Message passing

Solution to Exercise 7.1 a)

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import truncnorm
from scipy.stats import norm


def mutiplyGauss(m1,s1,m2,s2):
    s = 1/(1/s1+1/s2)
    m = (m1/s1+m2/s2)*s
    return m, s

def divideGauss(m1,s1,m2,s2):
    m,s = mutiplyGauss(m1,s1,m2,-s2)
    return m, s

def truncGaussMM(my_a,my_b,m1,s1):
    a, b = (my_a - m1) / np.sqrt(s1), (my_b - m1) / np.sqrt(s1)
    m = truncnorm.mean(a, b, loc=m1, scale=np.sqrt(s1))
    s = truncnorm.var(a, b, loc=m1, scale=np.sqrt(s1))
    return m, s


m0 = 0
s0 = 1
s = 1
y0 = 1

# Message from prior to node x
mu_x_m = m0
mu_x_s = s0

# Message from node x to factor f_xt
mu_x_xt_m = mu_x_m
mu_x_xt_s = mu_x_s

# Message from factor f_xt to node t
mu_xt_t_m = mu_x_xt_m
mu_xt_t_s = mu_x_xt_s + s

# Do moment matching of the marginal of t
if y0==1:
    a, b = 0, 1000
else:
    a, b = -1000, 0

pt_m, pt_s = truncGaussMM(a,b,mu_xt_t_m,mu_xt_t_s)

# Compute the updated message from f_yt to t
```

```
mu_yt_t_m, mu_yt_t_s = divideGauss(pt_m,pt_s,mu_xt_t_m,mu_xt_t_s)

# Compute the  message from t to f_tx
mu_t_xt_m = mu_yt_t_m
mu_t_xt_s = mu_yt_t_s

# Compute the  message from f_tx to x
mu_xt_x_m = mu_t_xt_m
mu_xt_x_s = mu_t_xt_s + s

# Compute the  marginal of x
px_m, px_s = mutiplyGauss(mu_x_m,mu_x_s,mu_xt_x_m,mu_xt_x_s)

print(px_m) # Output: 0.564189583548
print(px_s) # Output: 0.681690113816
```

b) Now we also need to do moment matching in node x due to the uniform message from the prior. As a consequence we need to compute a new updated incoming message $\hat{\mu}_{f_x \to x}(x) = \frac{\hat{p}(x|y=1)}{\hat{\mu}_{f_{xt} \to x}(x)}$. This new updated message will be send through the graph to the node t, where a new moment matching is performed and a new updated $\hat{\mu}_{f_{yt} \to t}$ is computed. This procedure is continued until convergence (which for this problem goes quite fast). See the implementation below.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import truncnorm
from scipy.stats import norm


def mutiplyGauss(m1,s1,m2,s2):
    s = 1/(1/s1+1/s2)
    m = (m1/s1+m2/s2)*s
    return m, s

def divideGauss(m1,s1,m2,s2):
    m,s = mutiplyGauss(m1,s1,m2,-s2)
    return m, s

def truncGaussMM(my_a,my_b,m1,s1):
    a, b = (my_a - m1) / np.sqrt(s1), (my_b - m1) / np.sqrt(s1)
    m = truncnorm.mean(a, b, loc=m1, scale=np.sqrt(s1))
    s = truncnorm.var(a, b, loc=m1, scale=np.sqrt(s1))
    return m, s


m0 = 0
s0 = 1
s = 1
y0 = 1

# Message from prior to node x
mu_x_m = m0
mu_x_s = s0

for j in range(0, 10):

    # Message from node x to factor f_xt
    mu_x_xt_m = mu_x_m
    mu_x_xt_s = mu_x_s

    # Message from factor f_xt to node t
    mu_xt_t_m = mu_x_xt_m
    mu_xt_t_s = mu_x_xt_s + s

    # Do moment matching of the marginal of t
    if y0==1:
        a, b = 0, 1000
    else:
        a, b = -1000, 0

    pt_m, pt_s = truncGaussMM(a,b,mu_xt_t_m,mu_xt_t_s)

```

```
    # Compute the updated message from f_yt to t
    mu_yt_t_m, mu_yt_t_s = divideGauss(pt_m,pt_s,mu_xt_t_m,mu_xt_t_s)

    # Compute the  message from t to f_tx
    mu_t_xt_m = mu_yt_t_m
    mu_t_xt_s = mu_yt_t_s

    # Compute the  message from f_tx to x
    mu_xt_x_m = mu_t_xt_m
    mu_xt_x_s = mu_t_xt_s + s

    # Do moment matching of the marginal of x
    a, b = -1, 1

    px_m, px_s = truncGaussMM(a,b,mu_xt_x_m,mu_xt_x_s)

    # Compute the updated message from f_x to x
    mu_x_m, mu_x_s = divideGauss(px_m,px_s,mu_x_m,mu_x_s)


print(px_m) # Output: 0.23743519877558206
print(px_s) # Output: 0.281149065200401




# Importance sampler

L = 100000 # number of samples
x = np.random.uniform(-1,1,L) #draw from p(x)
t = np.random.normal(size=L)*np.sqrt(s)+x #draw from p(t|x)
y = np.sign(t)
w = (y==y0)

w = L*w/np.sum(w)

# plot a weighted histogram
plt.hist(x,weights=w,bins=150,density=True)
xv = np.linspace(-5,5,1000)
plt.plot(xv,norm.pdf(xv,px_m,px_s))

# Estimate mean and variance
est_mean = np.sum(x*w)/L
est_var = np.sum(w*(est_mean-x)**2)/L

print(est_mean) # Output: 0.24464126486752208
print(est_var) # Output: 0.273835286008221
```

Solution to Exercise 7.2 a) The factor graph with the required messages is given in Figure 7.1.

In this case, the messages incoming to each of the nodes t_1 and t_2 depend on the marginal of the other node, which cannot be computed in closed form (it is a truncated Gaussian as in Exercise 6.3). Therefore, we use moment matching. In this case, we initialize one of the densities arbitrarily to a Gaussian density (say $p(t_2) \approx \mathcal{N}(t_2; 0, 1)$). Then we send up the message μ_1 through the graph until we have μ_4; now, we can compute the marginal distribution $p(t_1)$ with moment matching as shown in the previous exercise. Then, we compute the outgoing message μ_5 that we propagate until we reach μ_8. At this point, we use moment matching again to update the Gaussian approximation of the marginal $p(t_2)$. Then, we keep iterating this procedure until convergence.

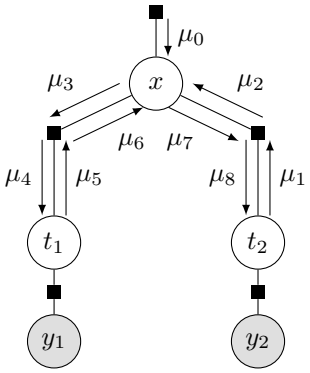

Figure 7.1: Factor graph of the model in Exercise 7.2.

b) The code below implements the message-passing algorithm for Exercise 7.2.

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
from scipy.stats import truncnorm
from scipy.stats import norm
import scipy.integrate as integrate

mpl.rcParams.update({'font.size':22})

class Message:
  "Class representing messages to simplify Gaussian message-passing"
  def __init__(self, m, s):
    self.m = m
    self.s = s

  def __str__(self):
    "For printing"
    return f"({self.m}, {self.s})"

  def __mul__(self, other):
    "Product of Gaussians"
    s = 1/(1/self.s + 1/other.s)
    m = s*(self.m/self.s + other.m/other.s)
    return Message(m,s)

  def __truediv__(self, other):
    "Division of Gaussians"
    s = 1/(1/self.s - 1/other.s)
    m = s*(self.m/self.s - other.m/other.s)
    return Message(m, s)

def approximateTruncatedGaussian(y, mu):
  "Approximate the truncated Gaussian marginal"
  m1 = mu.m
  s1 = mu.s
  if y == 1:
    a, b = 0, 1000
  else:
    a, b = -1000, 0
  a, b = (a - m1) / np.sqrt(s1), (b - m1) / np.sqrt(s1)
  m = truncnorm.mean(a, b, loc=m1, scale=np.sqrt(s1))
  s = truncnorm.var(a, b, loc=m1, scale=np.sqrt(s1))
  return Message(m, s)

def messagePassing(y1, y2, m0, s0, s):
  "Compute expectation propagation"
  mu0 = Message(m0, s0)
  mu1 = Message(0, 1) # Arbitrary initialization

  for _ in range(10):
    mu2 = Message(mu1.m, mu1.s + s)
```

```
    mu3 = mu0*mu2
    mu4 = Message(mu3.m, mu3.s + s)
    pt1 = approximateTruncatedGaussian(y1, mu4)
    mu5 = pt1/mu4
    mu6 = Message(mu5.m, mu5.s + s)
    mu7 = mu0*mu6
    mu8 = Message(mu7.m, mu7.s + s)
    pt2 = approximateTruncatedGaussian(y2, mu8)
    mu1 = pt2 / mu8

  pw = mu0*mu6*mu2
  return pw.m, pw.s


def importanceSampling(y1, y2, m0, s0, s, N):
  "Compute with importance sampling"

  samples = np.zeros(N)
  prior = norm(m0, np.sqrt(s0))
  sigma = np.sqrt(s)

  for i in range(N):
    y1_s = 0
    y2_s = 0
    while not ( y1 == y1_s  and y2 == y2_s):
      w = prior.rvs()
      t1, t2 = norm(w, sigma).rvs(2)
      y1_s = np.sign(t1)
      y2_s = np.sign(t2)
    samples[i] = w

  return samples


def analyticDensity(y1, y2, m0, s0, s):
  "Compute posterior density using numeric integration"
  if y1 == 1:
    p_y1_w = lambda w: 1 - norm(w, np.sqrt(s)).cdf(0)
  else:
    p_y1_w = lambda w: norm(w, np.sqrt(s)).cdf(0)

  if y2 == 1:
    p_y2_w = lambda w: 1 - norm(w, np.sqrt(s)).cdf(0)
  else:
    p_y2_w = lambda w: norm(w, np.sqrt(s)).cdf(0)

  p_w = lambda w: norm(m0, np.sqrt(s0)).pdf(w)
  fun = lambda w: p_y1_w(w)*p_y2_w(w)*p_w(w)

  Z = integrate.quad(fun, m0-10*np.sqrt(s0), m0+10*np.sqrt(s0))

  return lambda w: fun(w)/Z[0]



if __name__ == "__main__":
  m0 = 0
  s0 = 1
  s = 1

  y1 = 1
  y2 = 1

  m1, s1 = messagePassing(y1, y2, m0, s0, s)

  samples = importanceSampling(y1, y2, m0, s0, s, 5000)
  m2 = np.mean(samples)
  s2 = np.var(samples)

  pw = analyticDensity(y1, y2, m0, s0, s)


  plt.hist(samples, density=True, bins=30, label="IS", alpha=0.4)
```

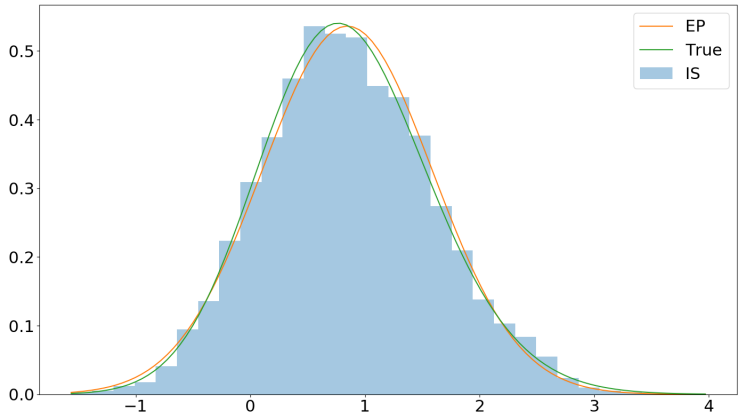

Figure 7.2: Posterior density computed by expectation propagation compared to the posterior computed using Monte Carlo. (The green plot corresponds to the true posterior density computed using numerical integration.)

```
x = np.linspace(min(samples), max(samples), 100)
plt.plot(x, norm(m1, np.sqrt(s1)).pdf(x), label="EP")
plt.plot(x, pw(x), label="True")
plt.legend()
plt.show()

print(m1, s1) # Output: 0.8456133754222313 0.5536594706042675
print(m2, s2) # Output: 0.8406517219154784 0.5534307452873626
```

Figure 7.2 shows the result of the simulation, together with the posterior density computed using numerical integration.

Bibliography

[1] David Barber. *Bayesian reasoning and machine learning*. Cambridge University Press, 2012.

[2] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[3] Kevin B Korb and Ann E Nicholson. *Bayesian artificial intelligence*. CRC press, 2010.

[4] Steffen L Lauritzen and David J Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society: Series B (Methodological)*, 50(2):157–194, 1988.

[5] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.