Time Series Analysis and Recurrent Neural Network
Giacomo Barzon - 3626438
Exercise 12

January 29, 2020

1 Laplace Approximation

Laplace's method:

$$\int e^{Mf(x)}dx = \sqrt{\frac{2\pi}{M|f''(x_0)|}}e^{Mf(x_0)} \quad as\ M \to \infty$$

where x_0 is the global maximum of $f(x)$.
Apply Laplace's method to approximate the integral:

$$N! = \int_0^\infty e^{-t}t^N dt$$

solution:

We can rewrite the integral using the logarithm properties:

$$N! = \int_0^\infty e^{N\ln t - t}dt = \int_0^\infty e^{N(lnt - t/N)}dt$$

In this way the integrand is in the form $e^{Nf(x)}$ where $f(x) = \ln t - t/N$.
So we have to compute the absolute maximum of $f(x)$:

$$f'(x) = \frac{1}{t} - \frac{1}{N} \to x_0 = N$$

$$f''(x) = -\frac{1}{t^2} \to f''(x_0) = -\frac{1}{N^2} < 0$$

By applying the Laplace's method finally we obtain:

$$N! \approx \sqrt{2\pi N}e^{N(\ln N - 1)} \approx \sqrt{2\pi N}e^{N\ln N} \ for\ N \to \infty$$

2 Variational Autoencoder

The VAE is optimizing the ELBO. The approximate posterior is assumed as a Gaussian $q(Z|X) = N(\mu, \Sigma)$, with diagonal covariance Σ. The parameters μ, Σ are the output of a neural network (also called encoder). The prior $p(Z) = N(0, I)$ is assumed to be unit Gaussian. The distribution $p(X|Z)$ is also a neural network (decoder). In our example, the images of the digits are binary: 1 for a white pixel, 0 for a black pixel. Hence, we take $p(X|Z)$ for each pixel as a Bernoulli distribution, the decoder output is a vector, each dimension characterizing the probability of a pixel being white.

a) Show mathematically that minimizing the Kullback-Leibler divergence KL $[q(Z|X)||p(Z|X)]$ between the approximate posterior $q(Z|X)$ and the true posterior $p(Z|X)$ is equivalent to maximizing the evidence lower bound:

$$ELBO = E_{z\sim q(Z|X)}[log(p(X|Z))] - KL(q(Z|X)||p(Z))$$

b) Derive the $KL(q(Z|X)||p(Z))$ in the case of $q(Z|X) = N(\mu, \Sigma)$, where Σ is diagonal and assuming $p(Z) = N(0, I)$. This is a more general result of what we showed in exercise 5.

c) Confirm that maximizing the likelihood of a Bernoulli distribution $p(X|Z) = \phi^x(1-\phi)^{(1-x)}$ (where ϕ is the predicted probability of a pixel being white) is giving rise to the cross-entropy loss $L(x, \phi) = xlog(\phi) + (1-x)log(1-\phi)$, for target x and prediction ϕ.

d) Implement the ELBO as a loss function for the VAE in the given code template. Use results from task 2b for the KL-divergence term and from task 2c for the reconstruction term.

e) Train the VAE. After training, plot samples from the learned distribution $p(X) = p(X|Z)p(Z)$

solution:

a)

$$ELBO = \int q(Z|X)log(p(X|Z))dZ + \int q(Z|X)log\frac{p(Z)}{q(Z|X)}dZ$$

$$= \int q(Z|X)log\frac{p(X|Z)p(Z)}{q(Z|X)}dZ = \int q(Z|X)log\frac{p(Z|X)p(X)}{q(Z|X)}dZ$$

$$= \int q(Z|X)log(p(X))dZ + \int q(Z|X)log\frac{p(Z|X)}{q(Z|X)}dZ$$

$$= log(p(X)) - KL(q(Z|X)||p(Z|X))$$

where we have used the Bayes theorem:

$$p(X|Z)p(Z) = p(X, Z) = p(Z|X)p(X)$$

Then:

$$\max_{q(Z|X)} ELBO = \max_{q(Z|X)} [log(p(x)) - KL(q(Z|X)||p(Z|X))] = \min_{q(Z|X)} KL(q(Z|X)||p(Z|X))$$

since $p(x)$ doesn't depend on the proposal density $q(Z|X)$.

b)

Assuming $Z = (Z_1, ..., Z_M) \in R^M$ and σ_i^2 the diagonal elements of the variance matrix Σ:

$$KL(q(Z|X)||p(Z)) = -\int q(Z|X) log \frac{p(Z)}{q(Z|X)} dZ$$

$$\int q(Z|X) log(q(Z|X)) dZ =$$

$$= -\frac{M}{2} log(2\pi) - \frac{1}{2} log|\Sigma| + \int (2\pi)^{-\frac{M}{2}} |\Sigma|^{-\frac{1}{2}} exp\left[-\frac{1}{2}\sum_{i=1}^{M}\frac{(Z_i - \mu_i)^2}{\sigma_i^2}\right]\left(-\frac{1}{2}\sum_{i=1}^{M}\frac{(Z_i - \mu_i)^2}{\sigma_i^2}\right) dZ_1...dZ_M$$

$$= -\frac{M}{2} log(2\pi) - \frac{1}{2} log|\Sigma| - \frac{1}{2}\sum_{i=1}^{M}\int (2\pi)^{-\frac{1}{2}} exp\left[-\frac{1}{2}Y_i^2\right] Y_i^2 dY_i$$

$$= -\frac{M}{2} log(2\pi) - \frac{1}{2} log|\Sigma| - \frac{1}{2} tr(I_M)$$

$$\int q(Z|X) log(p(Z)) dZ =$$

$$= -\frac{M}{2} log(2\pi) + \int (2\pi)^{-\frac{M}{2}} |\Sigma|^{-\frac{1}{2}} exp\left[-\frac{1}{2}\sum_{i=1}^{M}\frac{(Z_i - \mu_i)^2}{\sigma_i^2}\right]\left(-\frac{1}{2}\sum_{i=1}^{M} Z_i^2\right) dZ_1...dZ_M$$

$$= -\frac{M}{2} log(2\pi) - \frac{1}{2}\sum_{i=1}^{M}\int (2\pi)^{-\frac{1}{2}} exp\left[-\frac{1}{2}Y_i^2\right] (Y_i + \mu_i)^2 dY_i$$

$$= -\frac{M}{2} log(2\pi) - \frac{1}{2}\sum_{i=1}^{M}\int (2\pi)^{-\frac{1}{2}} exp\left[-\frac{1}{2}Y_i^2\right] (\sigma_i Y_i + \mu_i)^2 dY_i$$

$$= -\frac{M}{2} log(2\pi) - \frac{1}{2}\mu^T\mu - \frac{1}{2} tr(\Sigma)$$

where we have used the ansatz $Z_i \rightarrow Y_i = \frac{Z_i - \mu_i}{\sigma_i}$ and the properties of the moments of a gaussian distribution:

$$\int_{-\infty}^{\infty} (2\pi)^{-\frac{1}{2}} e^{-\frac{y^2}{2}} = 1 \ (normalization)$$

$$\int_{-\infty}^{\infty} (2\pi)^{-\frac{1}{2}} e^{-\frac{y^2}{2}} y = 0 \ (mean)$$

$$\int_{-\infty}^{\infty} (2\pi)^{-\frac{1}{2}} e^{-\frac{y^2}{2}} y^2 = 1 \ (variance)$$

By putting all together we obtain:

$$KL(q(Z|X)||p(Z)) = -\frac{1}{2} log|\Sigma| - \frac{1}{2} tr(I_M - \Sigma) + \frac{1}{2}\mu^T \mu$$

$$= \frac{1}{2} \sum_{i=1}^{M} \left[\sigma_i^2 + \mu_i^2 - 1 - log(\sigma_i^2) \right]$$

c)

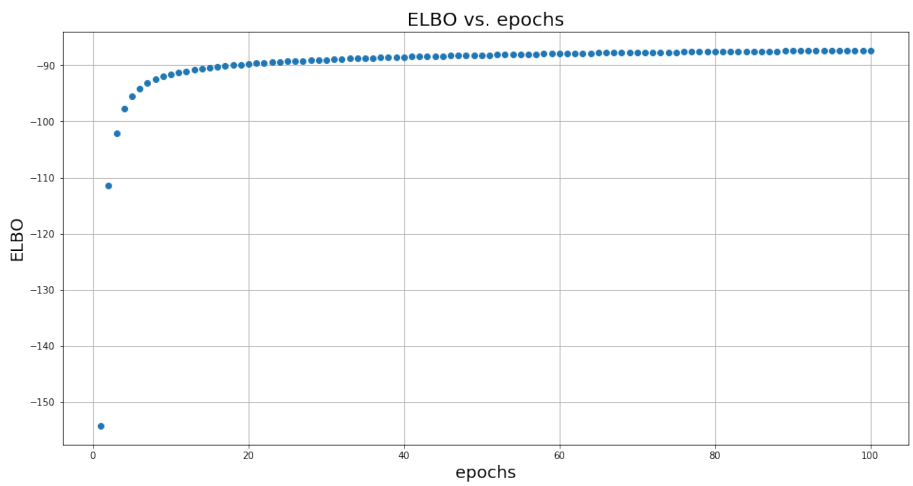
Since the logarithm is a monotonic function, $\max p(X|Z)$ is equivalent to $\max log(p(X|Z))$, that is:

$$log(p(X|Z)) = X log(\phi) + (1 - X) log(1 - \phi) = L(X, \phi)$$

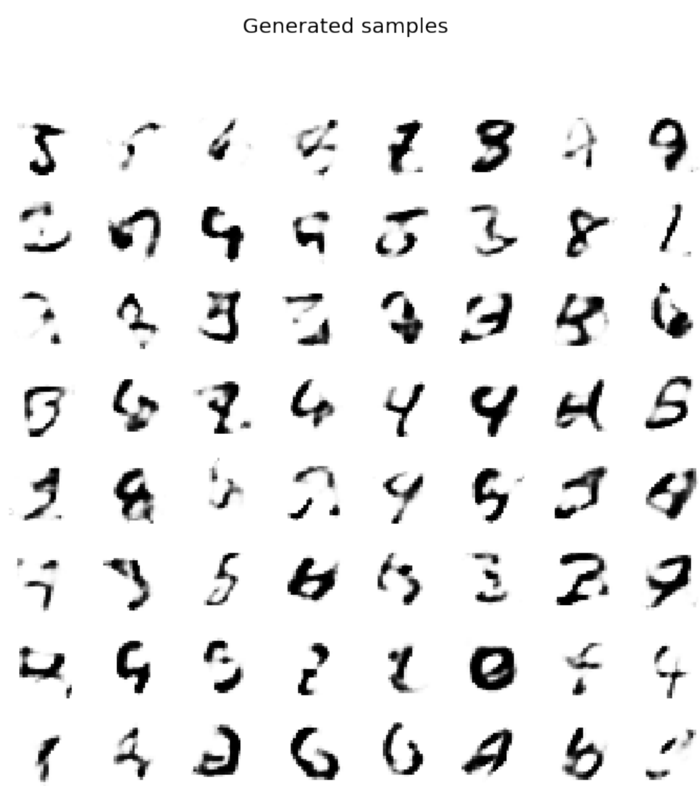
d)

```
def loss_function(x, x_sample, z_mu, z_logvar):
    rec_loss = F.binary_cross_entropy(x_sample, x, reduction='sum')
    kl_loss = 0.5 * tc.sum(z_mu**2 + tc.exp(z_logvar) - tc.ones(dim_z)
    - z_logvar)
    return rec_loss + kl_loss
```

e)



ELBO vs. epochs
−90
−100
−110
−120
−130
−140
−150
ELBO
0
20
40
60
80
100
epochs



Generated samples

Appendix

```
import torch as tc
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import matplotlib.pyplot as plt


class Encoder(nn.Module):
    def __init__(self, dim_x, dim_h, dim_z):
        super().__init__()
        self.linear = nn.Linear(dim_x, dim_h)
        self.mu = nn.Linear(dim_h, dim_z)
        self.logvar = nn.Linear(dim_h, dim_z)

    def forward(self, x):
        h = F.relu(self.linear(x))
        z_mu = self.mu(h)
        z_logvar = self.logvar(h)
        return z_mu, z_logvar


class Decoder(nn.Module):
    def __init__(self, dim_z, dim_h, dim_x):
        super().__init__()
        self.linear1 = nn.Linear(dim_z, dim_h)
        self.linear2 = nn.Linear(dim_h, dim_x)

    def forward(self, z):
        h = F.relu(self.linear1(z))
        x = tc.sigmoid(self.linear2(h))
        return x


class VAE(nn.Module):
    def __init__(self, enc, dec):
        super().__init__()
        self.enc = enc
        self.dec = dec

    def forward(self, x):
        z_mu, z_logvar = self.enc(x)
```

```
        z_sample = reparametrize(z_mu, z_logvar)
        x_sample = self.dec(z_sample)
        return x_sample, z_mu, z_logvar


def reparametrize(z_mu, z_logvar):
    # NOTE: the 'reparametrization trick' will be treated next week
    # essentially it is sampling from a Gaussian distribution,
    # but still being differentiable w.r.t. the parameters mu, Sigma
    std = tc.exp(z_logvar)
    eps = tc.randn_like(std)
    x_sample = eps.mul(std).add_(z_mu)
    return x_sample


def loss_function(x, x_sample, z_mu, z_logvar):
    rec_loss = F.binary_cross_entropy(x_sample, x, reduction='sum')
    kl_loss = 0.5 * tc.sum(z_mu**2 + tc.exp(z_logvar) - tc.ones(dim_z)
    - z_logvar)
    return rec_loss + kl_loss


def train():
    model.train()
    train_loss = 0
    for i, (x, _) in enumerate(train_loader):
        x = x.to(device)
        x = x.view(-1, 28 * 28)
        optimizer.zero_grad()
        x_sample, z_mu, z_logvar = model(x)
        loss = loss_function(x, x_sample, z_mu, z_logvar)
        loss.backward()
        train_loss += loss.item()
        optimizer.step()
    return train_loss


def test():
    model.eval()
    test_loss = 0
    with tc.no_grad():  # no need to track the gradients here
        for i, (x, _) in enumerate(test_loader):
            x = x.to(device)
            x = x.view(-1, 28 * 28)
            z_sample, z_mu, z_var = model(x)
            loss = loss_function(x, z_sample, z_mu, z_var)
```

```
            test_loss += loss.item()
    return test_loss


def generate():
    sample = tc.randn(generated, dim_z)
    return model.dec.forward(sample)


if __name__ == '__main__':
    batch_size = 64         # number of data points in each batch
    n_epochs = 100          # times to run the model on complete data
    dim_x = 28 * 28         # size of each input
    dim_h = 256             # hidden dimension
    dim_z = 50              # latent vector dimension
    lr = 1e-3               # learning rate
    generated = 64          # number of generated images

    # import dataset
    device = tc.device('cuda' if tc.cuda.is_available() else 'cpu')
    transforms = transforms.Compose([transforms.ToTensor()])
    train_set = datasets.MNIST('./data', train=True, download=True, transform=tra
    test_set = datasets.MNIST('./data', train=False, download=True, transform=tra
    train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=True)
    test_loader = DataLoader(test_set, batch_size=batch_size)

    # initialize VAE model
    encoder = Encoder(dim_x, dim_h, dim_z)
    decoder = Decoder(dim_z, dim_h, dim_x)
    model = VAE(encoder, decoder).to(device)

    # initialize optimizer
    optimizer = optim.Adam(model.parameters(), lr=lr)

    # network training
    ELBO = []

    for epoch in range(n_epochs):
        train_loss = train()
        test_loss = test()
        train_loss /= len(train_set)
        test_loss /= len(test_set)
        print(f'Epoch {epoch+1}, Train Loss: {train_loss:.2f},
        Test Loss: {test_loss:.2f}')
        ELBO.append(-train_loss)
```

```
# plot ELBO
fig1 = plt.subplots(figsize=[16,8])

plt.plot(list(range(1,n_epochs+1)), ELBO, 'o')
plt.xlabel('epochs', fontsize = 18)
plt.ylabel('ELBO', fontsize = 18)
plt.grid()
plt.title('ELBO vs. epochs', fontsize=20)
plt.show(fig1)

# plot generated images
generated_images = generate()
generated_images = generated_images.detach().numpy()

fig, ax = plt.subplots(8,8)
fig.set_size_inches(12,12)
fig.suptitle('Generated samples', fontsize=20)

for i in range(8):
    for j in range(8):
        ax[i,j].imshow(generated_images[8*i+j].reshape(28,28),
        cmap='gray_r', vmin=0, vmax=1)
        ax[i,j].axis('off')
plt.show(block=True)
```