Advanced machine learning and data analysis for the physical sciences

Morten Hjorth-Jensen[1,2]

[1]Department of Physics and Center for Computing in Science Education, University of Oslo, Norway
[2]Department of Physics and Astronomy and Facility for Rare Isotope Beams, Michigan State University, East Lansing, Michigan, USA

April 23, 2024

Plans for the week April 22-26, 2024

Deep generative models.

1. Variational Autoencoders (VAE), Mathematics and codes, continuation from last week
2. Diffusion models
3. Reading recommendation:
   (a) Goodfellow et al chapter 20.10-20-14
   (b) Calvin Luo `https://calvinyluo.com/2022/08/26/diffusion-tutorial.html`

Mathematics of VAEs

We want to train the marginal probability with some latent varriables $\boldsymbol{h}$

$$p(\boldsymbol{x};\boldsymbol{\Theta}) = \int d\boldsymbol{h}p(\boldsymbol{x},\boldsymbol{h};\boldsymbol{\Theta}),$$

for the continuous version (see previous slides for the discrete variant).

Using the KL divergence

In practice, for most $\boldsymbol{h}$, $p(\boldsymbol{x}|\boldsymbol{h};\boldsymbol{\Theta})$ will be nearly zero, and hence contributes almost nothing to our estimate of $p(\boldsymbol{x})$.

The key idea behind the variational autoencoder is to attempt to sample values of $\boldsymbol{h}$ that are likely to have produced $\boldsymbol{x}$, and compute $p(\boldsymbol{x})$ just from those.

This means that we need a new function $Q(\boldsymbol{h}|\boldsymbol{x})$ which can take a value of $\boldsymbol{x}$ and give us a distribution over $\boldsymbol{h}$ values that are likely to produce $\boldsymbol{x}$. Hopefully the space of $\boldsymbol{h}$ values that are likely under Q will be much smaller than the space of all $\boldsymbol{h}$'s that are likely under the prior $p(\boldsymbol{h})$. This lets us, for example, compute $E_{\boldsymbol{h}\sim Q}p(\boldsymbol{x}|\boldsymbol{h})$ relatively easily. Note that we drop $\boldsymbol{\Theta}$ from here and for notational simplicity.

Kullback-Leibler again

However, if $\boldsymbol{h}$ is sampled from an arbitrary distribution with PDF $Q(\boldsymbol{h})$, which is not $\mathcal{N}(0, I)$, then how does that help us optimize $p(\boldsymbol{x})$?

The first thing we need to do is relate $E_{\boldsymbol{h}\sim Q}P(\boldsymbol{x}|\boldsymbol{h})$ and $p(\boldsymbol{x})$. We will see where Q comes from later.

The relationship between $E_{\boldsymbol{h}\sim Q}p(\boldsymbol{x}|\boldsymbol{h})$ and $p(\boldsymbol{x})$ is one of the cornerstones of variational Bayesian methods. We begin with the definition of Kullback-Leibler divergence (KL divergence or $\mathcal{D}$) between $p(\boldsymbol{h}|\boldsymbol{x})$ and $Q(\boldsymbol{h})$, for some arbitrary Q (which may or may not depend on $\boldsymbol{x}$):

$$\mathcal{D}\left[Q(\boldsymbol{h})\|p(\boldsymbol{h}|\boldsymbol{x})\right] = E_{\boldsymbol{h}\sim Q}\left[\log Q(\boldsymbol{h}) - \log p(\boldsymbol{h}|\boldsymbol{x})\right].$$

And applying Bayes rule

We can get both $p(\boldsymbol{x})$ and $p(\boldsymbol{x}|\boldsymbol{h})$ into this equation by applying Bayes rule to $p(\boldsymbol{h}|\boldsymbol{x})$

$$\mathcal{D}\left[Q(\boldsymbol{h})\|p(\boldsymbol{h}|\boldsymbol{x})\right] = E_{\boldsymbol{h}\sim Q}\left[\log Q(\boldsymbol{h}) - \log p(\boldsymbol{x}|\boldsymbol{h}) - \log p(\boldsymbol{h})\right] + \log p(\boldsymbol{x}).$$

Here, $\log p(\boldsymbol{x})$ comes out of the expectation because it does not depend on $\boldsymbol{h}$. Negating both sides, rearranging, and contracting part of $E_{\boldsymbol{h}\sim Q}$ into a KL-divergence terms yields:

$$\log p(\boldsymbol{x}) - \mathcal{D}\left[Q(\boldsymbol{h})\|p(\boldsymbol{h}|\boldsymbol{x})\right] = E_{\boldsymbol{h}\sim Q}\left[\log p(\boldsymbol{x}|\boldsymbol{h})\right] - \mathcal{D}\left[Q(\boldsymbol{h})\|P(\boldsymbol{h})\right].$$

Rearraning

Using Bayes rule we obtain

$$E_{\boldsymbol{h}\sim Q}\left[\log p(y_i|\boldsymbol{h}, x_i)\right] = E_{\boldsymbol{h}\sim Q}\left[\log p(\boldsymbol{h}|y_i, x_i) - \log p(\boldsymbol{h}|x_i) + \log p(y_i|x_i)\right]$$

Rearranging the terms and subtracting $E_{\boldsymbol{h}\sim Q}\log Q(\boldsymbol{h})$ from both sides gives

$$\begin{aligned}\log P(y_i|x_i) - E_{\boldsymbol{h}\sim Q}\left[\log Q(\boldsymbol{h}) - \log p(\boldsymbol{h}|x_i, y_i)\right] = \\ E_{\boldsymbol{h}\sim Q}\left[\log p(y_i|\boldsymbol{h}, x_i) + \log p(\boldsymbol{h}|x_i) - \log Q(\boldsymbol{h})\right]\end{aligned}$$

Note that $\boldsymbol{x}$ is fixed, and Q can be *any* distribution, not just a distribution which does a good job mapping $\boldsymbol{x}$ to the $\boldsymbol{h}$'s that can produce X.

Inferring the probability

Since we are interested in inferring $p(\boldsymbol{x})$, it makes sense to construct a Q which *does* depend on $\boldsymbol{x}$, and in particular, one which makes $\mathcal{D}\left[Q(\boldsymbol{h}) \| p(\boldsymbol{h}|\boldsymbol{x})\right]$ small

$$\log p(\boldsymbol{x}) - \mathcal{D}\left[Q(\boldsymbol{h}|\boldsymbol{x}) \| p(\boldsymbol{h}|\boldsymbol{x})\right] = E_{\boldsymbol{h}\sim Q}\left[\log p(\boldsymbol{x}|\boldsymbol{h})\right] - \mathcal{D}\left[Q(\boldsymbol{h}|\boldsymbol{x}) \| p(\boldsymbol{h})\right].$$

Hence, during training, it makes sense to choose a Q which will make $E_{\boldsymbol{h}\sim Q}[\log Q(\boldsymbol{h}) - \log p(\boldsymbol{h}|x_i, y_i)]$ (a $\mathcal{D}$-divergence) small, such that the right hand side is a close approximation to $\log p(y_i|y_i)$.

Central equation of VAEs

This equation serves as the core of the variational autoencoder, and it is worth spending some time thinking about what it means.

1. The left hand side has the quantity we want to maximize, namely $\log p(\boldsymbol{x})$ plus an error term.
2. The right hand side is something we can optimize via stochastic gradient descent given the right choice of Q.

Setting up SGD

So how can we perform stochastic gradient descent?

First we need to be a bit more specific about the form that $Q(\boldsymbol{h}|\boldsymbol{x})$ will take. The usual choice is to say that $Q(\boldsymbol{h}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{h}|\mu(\boldsymbol{x};\vartheta), \Sigma(;\vartheta))$, where μ and Σ are arbitrary deterministic functions with parameters ϑ that can be learned from data (we will omit ϑ in later equations). In practice, μ and Σ are again implemented via neural networks, and Σ is constrained to be a diagonal matrix.

More on the SGD

The name variational "autoencoder" comes from the fact that μ and Σ are "encoding" $\boldsymbol{x}$ into the latent space $\boldsymbol{h}$. The advantages of this choice are computational, as they make it clear how to compute the right hand side. The last term—$\mathcal{D}\left[Q(\boldsymbol{h}|\boldsymbol{x}) \| p(\boldsymbol{h})\right]$—is now a KL-divergence between two multivariate Gaussian distributions, which can be computed in closed form as:

$$\mathcal{D}[\mathcal{N}(\mu_0, \Sigma_0) \| \mathcal{N}(\mu_1, \Sigma_1)] =$$
$$\tfrac{1}{2}\left(\mathrm{tr}\left(\Sigma_1^{-1}\Sigma_0\right) + \left(\mu_1 - \mu_0\right)^{\top}\Sigma_1^{-1}(\mu_1 - \mu_0) - k + \log\left(\tfrac{\det \Sigma_1}{\det \Sigma_0}\right)\right)$$

where k is the dimensionality of the distribution.

Simplification

In our case, this simplifies to:

$$\mathcal{D}[\mathcal{N}(\mu(X),\Sigma(X))\|\mathcal{N}(0,I)] = \\ \frac{1}{2}\left(\operatorname{tr}\left(\Sigma(X)\right) + \left(\mu(X)\right)^\top\left(\mu(X)\right) - k - \log\det\left(\Sigma(X)\right)\right).$$

Terms to compute

The first term on the right hand side is a bit more tricky. We could use sampling to estimate $E_{z\sim Q}\left[\log P(X|z)\right]$, but getting a good estimate would require passing many samples of z through f, which would be expensive. Hence, as is standard in stochastic gradient descent, we take one sample of z and treat $\log P(X|z)$ for that z as an approximation of $E_{z\sim Q}\left[\log P(X|z)\right]$. After all, we are already doing stochastic gradient descent over different values of X sampled from a dataset D. The full equation we want to optimize is:

$$E_{X\sim D}\left[\log P(X) - \mathcal{D}\left[Q(z|X)\|P(z|X)\right]\right] = \\ E_{X\sim D}\left[E_{z\sim Q}\left[\log P(X|z)\right] - \mathcal{D}\left[Q(z|X)\|P(z)\right]\right].$$

Computing the gradients

If we take the gradient of this equation, the gradient symbol can be moved into the expectations. Therefore, we can sample a single value of X and a single value of z from the distribution $Q(z|X)$, and compute the gradient of:

$$\log P(X|z) - \mathcal{D}\left[Q(z|X)\|P(z)\right]. \quad (1)$$

We can then average the gradient of this function over arbitrarily many samples of X and z, and the result converges to the gradient.

There is, however, a significant problem $E_{z\sim Q}\left[\log P(X|z)\right]$ depends not just on the parameters of P, but also on the parameters of Q.

In order to make VAEs work, it is essential to drive Q to produce codes for X that P can reliably decode.

$$E_{X\sim D}\left[E_{\epsilon\sim\mathcal{N}(0,I)}[\log P(X|z=\mu(X)+\Sigma^{1/2}(X)*\epsilon)] - \mathcal{D}\left[Q(z|X)\|P(z)\right]\right].$$

Code examples using Keras

Code taken from https://keras.io/examples/generative/vae/

```
"""
Title: Variational AutoEncoder
Author: [fchollet](https://twitter.com/fchollet)
Date created: 2020/05/03
Last modified: 2023/11/22
Description: Convolutional Variational AutoEncoder (VAE) trained on MNIST digits.
Accelerator: GPU
```

```
"""

"""
## Setup
"""

import os

os.environ["KERAS_BACKEND"] = "tensorflow"

import numpy as np
import tensorflow as tf
import keras
from keras import layers

"""
## Create a sampling layer
"""


class Sampling(layers.Layer):
    """Uses (z_mean, z_log_var) to sample z, the vector encoding a digit."""

    def call(self, inputs):
        z_mean, z_log_var = inputs
        batch = tf.shape(z_mean)[0]
        dim = tf.shape(z_mean)[1]
        epsilon = tf.random.normal(shape=(batch, dim))
        return z_mean + tf.exp(0.5 * z_log_var) * epsilon


"""
## Build the encoder
"""

latent_dim = 2

encoder_inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(32, 3, activation="relu", strides=2, padding="same")(encoder_inputs)
x = layers.Conv2D(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Flatten()(x)
x = layers.Dense(16, activation="relu")(x)
z_mean = layers.Dense(latent_dim, name="z_mean")(x)
z_log_var = layers.Dense(latent_dim, name="z_log_var")(x)
z = Sampling()([z_mean, z_log_var])
encoder = keras.Model(encoder_inputs, [z_mean, z_log_var, z], name="encoder")
encoder.summary()

"""
## Build the decoder
"""

latent_inputs = keras.Input(shape=(latent_dim,))
x = layers.Dense(7 * 7 * 64, activation="relu")(latent_inputs)
x = layers.Reshape((7, 7, 64))(x)
x = layers.Conv2DTranspose(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Conv2DTranspose(32, 3, activation="relu", strides=2, padding="same")(x)
decoder_outputs = layers.Conv2DTranspose(1, 3, activation="sigmoid", padding="same")(x)
decoder = keras.Model(latent_inputs, decoder_outputs, name="decoder")
decoder.summary()
```

```
"""
## Define the VAE as a `Model` with a custom `train_step`
"""


class VAE(keras.Model):
    def __init__(self, encoder, decoder, **kwargs):
        super().__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder
        self.total_loss_tracker = keras.metrics.Mean(name="total_loss")
        self.reconstruction_loss_tracker = keras.metrics.Mean(
            name="reconstruction_loss"
        )
        self.kl_loss_tracker = keras.metrics.Mean(name="kl_loss")

    @property
    def metrics(self):
        return [
            self.total_loss_tracker,
            self.reconstruction_loss_tracker,
            self.kl_loss_tracker,
        ]

    def train_step(self, data):
        with tf.GradientTape() as tape:
            z_mean, z_log_var, z = self.encoder(data)
            reconstruction = self.decoder(z)
            reconstruction_loss = tf.reduce_mean(
                tf.reduce_sum(
                    keras.losses.binary_crossentropy(data, reconstruction),
                    axis=(1, 2),
                )
            )
            kl_loss = -0.5 * (1 + z_log_var - tf.square(z_mean) - tf.exp(z_log_var))
            kl_loss = tf.reduce_mean(tf.reduce_sum(kl_loss, axis=1))
            total_loss = reconstruction_loss + kl_loss
        grads = tape.gradient(total_loss, self.trainable_weights)
        self.optimizer.apply_gradients(zip(grads, self.trainable_weights))
        self.total_loss_tracker.update_state(total_loss)
        self.reconstruction_loss_tracker.update_state(reconstruction_loss)
        self.kl_loss_tracker.update_state(kl_loss)
        return {
            "loss": self.total_loss_tracker.result(),
            "reconstruction_loss": self.reconstruction_loss_tracker.result(),
            "kl_loss": self.kl_loss_tracker.result(),
        }


"""
## Train the VAE
"""

(x_train, _), (x_test, _) = keras.datasets.mnist.load_data()
mnist_digits = np.concatenate([x_train, x_test], axis=0)
mnist_digits = np.expand_dims(mnist_digits, -1).astype("float32") / 255

vae = VAE(encoder, decoder)
vae.compile(optimizer=keras.optimizers.Adam())
vae.fit(mnist_digits, epochs=30, batch_size=128)
```

```
"""
## Display a grid of sampled digits
"""

import matplotlib.pyplot as plt


def plot_latent_space(vae, n=30, figsize=15):
    # display a n*n 2D manifold of digits
    digit_size = 28
    scale = 1.0
    figure = np.zeros((digit_size * n, digit_size * n))
    # linearly spaced coordinates corresponding to the 2D plot
    # of digit classes in the latent space
    grid_x = np.linspace(-scale, scale, n)
    grid_y = np.linspace(-scale, scale, n)[::-1]

    for i, yi in enumerate(grid_y):
        for j, xi in enumerate(grid_x):
            z_sample = np.array([[xi, yi]])
            x_decoded = vae.decoder.predict(z_sample, verbose=0)
            digit = x_decoded[0].reshape(digit_size, digit_size)
            figure[
                i * digit_size : (i + 1) * digit_size,
                j * digit_size : (j + 1) * digit_size,
            ] = digit

    plt.figure(figsize=(figsize, figsize))
    start_range = digit_size // 2
    end_range = n * digit_size + start_range
    pixel_range = np.arange(start_range, end_range, digit_size)
    sample_range_x = np.round(grid_x, 1)
    sample_range_y = np.round(grid_y, 1)
    plt.xticks(pixel_range, sample_range_x)
    plt.yticks(pixel_range, sample_range_y)
    plt.xlabel("z[0]")
    plt.ylabel("z[1]")
    plt.imshow(figure, cmap="Greys_r")
    plt.show()


plot_latent_space(vae)

"""
## Display how the latent space clusters different digit classes
"""


def plot_label_clusters(vae, data, labels):
    # display a 2D plot of the digit classes in the latent space
    z_mean, _, _ = vae.encoder.predict(data, verbose=0)
    plt.figure(figsize=(12, 10))
    plt.scatter(z_mean[:, 0], z_mean[:, 1], c=labels)
    plt.colorbar()
    plt.xlabel("z[0]")
    plt.ylabel("z[1]")
    plt.show()


(x_train, y_train), _ = keras.datasets.mnist.load_data()
x_train = np.expand_dims(x_train, -1).astype("float32") / 255
```

```
plot_label_clusters(vae, x_train, y_train)
```

Code in PyTorch for VAEs

```
import torch
from torch.autograd import Variable
import numpy as np
import torch.nn.functional as F
import torchvision
from torchvision import transforms
import torch.optim as optim
from torch import nn
import matplotlib.pyplot as plt
from torch import distributions

class Encoder(torch.nn.Module):
    def __init__(self, D_in, H, latent_size):
        super(Encoder, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, H)
        self.enc_mu = torch.nn.Linear(H, latent_size)
        self.enc_log_sigma = torch.nn.Linear(H, latent_size)

    def forward(self, x):
        x = F.relu(self.linear1(x))
        x = F.relu(self.linear2(x))
        mu = self.enc_mu(x)
        log_sigma = self.enc_log_sigma(x)
        sigma = torch.exp(log_sigma)
        return torch.distributions.Normal(loc=mu, scale=sigma)


class Decoder(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(Decoder, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)


    def forward(self, x):
        x = F.relu(self.linear1(x))
        mu = torch.tanh(self.linear2(x))
        return torch.distributions.Normal(mu, torch.ones_like(mu))

class VAE(torch.nn.Module):
    def __init__(self, encoder, decoder):
        super(VAE, self).__init__()
        self.encoder = encoder
        self.decoder = decoder

    def forward(self, state):
        q_z = self.encoder(state)
        z = q_z.rsample()
        return self.decoder(z), q_z


transform = transforms.Compose(
    [transforms.ToTensor(),
     # Normalize the images to be -0.5, 0.5
     transforms.Normalize(0.5, 1)]
    )
mnist = torchvision.datasets.MNIST('./', download=True, transform=transform)

input_dim = 28 * 28
batch_size = 128
num_epochs = 100
learning_rate = 0.001
hidden_size = 512
latent_size = 8

if torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

dataloader = torch.utils.data.DataLoader(
    mnist, batch_size=batch_size,
    shuffle=True,
    pin_memory=torch.cuda.is_available())

print('Number of samples: ', len(mnist))

encoder = Encoder(input_dim, hidden_size, latent_size)
decoder = Decoder(latent_size, hidden_size, input_dim)

vae = VAE(encoder, decoder).to(device)

optimizer = optim.Adam(vae.parameters(), lr=learning_rate)
for epoch in range(num_epochs):
    for data in dataloader:
        inputs, _ = data
        inputs = inputs.view(-1, input_dim).to(device)
        optimizer.zero_grad()
        p_x, q_z = vae(inputs)
        log_likelihood = p_x.log_prob(inputs).sum(-1).mean()
        kl = torch.distributions.kl_divergence(
            q_z,
            torch.distributions.Normal(0, 1.)
        ).sum(-1).mean()
        loss = -(log_likelihood - kl)
        loss.backward()
        optimizer.step()
        l = loss.item()
    print(epoch, l, log_likelihood.item(), kl.item())
```

Variational diffusion models