Machine Learning Workshop 2

Variational Autoencoder

Jonathan Guymont

National Bank of Canada, 2018

Outline

1. Autoencoders
2. Variational autoencoder
3. Anomaly detection and autoencoder

Autoencoders

Autoencoers are neural network that are trained to learn how to map their input to their input. Internally, it has an hidden layer $\boldsymbol{h}$ that contains a lossy summary of the relevant feature for the task.

Autoencoders

An autoencoder can be seen has a two parts network

- Encoder function: $\boldsymbol{z} = f_{\phi}(\boldsymbol{x})$
- Decoder function: $\tilde{\boldsymbol{x}} = g_{\theta}(\boldsymbol{z})$
- ϕ and θ are set of learned parameters

Autoencoders

The simplest autoencoder is a one layer MLP:

$$
\begin{aligned}
\mathbf{z} =& \operatorname{relu}\left(\mathbf{W}_{xz}\mathbf{x} + \mathbf{b}_{xz}\right) \\
\tilde{\mathbf{x}} =& \operatorname{sigmoid}\left(\mathbf{W}_{zx}\mathbf{z} + \mathbf{b}_{zx}\right)
\end{aligned} \tag{1}
$$

Pytorch simple autoencoder

```
class Autoencoder:
        def __init__(self, **kargs):
            """constructor"""
            pass

        def encoder(self, x):
            pass

        def decoder(self, z)
            pass

        def forward(self, x):
            pass
```

Parameter initialization

```
class Autoencoder:
    def __init__(self, x_dim, z_dim):
        # encoder parameters \phi
        self.Wxz = xavier_init(size=[x_dim, z_dim])
        self.bxz = Variable(torch.zeros(z_dim), requires_grad=True)
        # decoder parameters \theta
        self.Wzx = xavier_init(size=[z_dim, x_dim])
        self.bzx = Variable(torch.zeros(x_dim), requires_grad=True)

    def encoder(self, x):
        ...
    def decoder(self, z):
        ...
    def forward(self, x):
        ...
```

Encoder $f_\phi(x)$

$$\mathbf{z} = \mathrm{relu}\left(\mathbf{W}_{xz}\mathbf{x} + \mathbf{b}_{xz}\right) \tag{2}$$

$$\phi = \{\mathbf{W}_{xz}, \mathbf{b}_{xz}\} \tag{3}$$

```
class Autoencoder:
    def __init__(self, x_dim, z_dim):
        ...
    def encoder(self, x):
        z = F.relu(self.Wxz @ z + self.bxz.repeat(x.size(0), 1))
        return z

    def decoder(self, z):
        ...
    def forward(self, x):
        ...
```

Decoder $g_\theta(x)$

$$\mathbf{z} = \sigma\left(\mathbf{W}_{zx}\mathbf{z} + \mathbf{b}_{zx}\right) \tag{4}$$
$$\boldsymbol{\theta} = \{\mathbf{W}_{zx}, \mathbf{b}_{zx}\} \tag{5}$$

```
class Autoencoder:
    def __init__(self, x_dim, z_dim):
        ...
    def encoder(self, x):
        ...
    def decoder(self, z):
        x_recon = F.sigmoid(z @ self.Wzx + self.bzx.repeat(z.size(0), 1))
        return x_recon

    def forward(self, x):
        ...
```

Forward propagation

```
class Autoencoder:
    def __init__(self, x_dim, z_dim):
        ...
    def encoder(self, x):
        ...
    def decoder(self, z):
        ...
    def forward(self, x):
        z = self.encoder(x)
        x_recon = self.decoder(z)
        return x_recon
```

Pytorch simple autoencoder

```
class Autoencoder:
    def __init__(self, x_dim, z_dim):
        # encoder parameters
        Wxz = xavier_init(size=[x_dim, z_dim])
        bxz = Variable(torch.zeros(z_dim), requires_grad=True)
        # decoder parameters
        Wzx = xavier_init(size=[h_dim, x_dim])
        bzx = Variable(torch.zeros(X_dim), requires_grad=True)

    def encoder(self, x):
        z = F.relu(x @ self.Wxh + self.bxh.repeat(x.size(0), 1))
        return z

    def decoder(self, z):
        x_recon = F.sigmoid(z @ self.Wzx + self.bzx.repeat(z.size(0), 1))
        return x_recon

    def forward(self, x):
        z = self.encoder(x)
        x_recon = self.decoder(z)
        return x_recon
```

Autoencoder - Loss Function

If you treat the problem like a *regression*[1], use the mean square error between the input and the reconstruction

$$\mathcal{L} = \sum_{i=1}^{d} (x_i - \tilde{x}_i)^2 \tag{6}$$

If you treat the problem like a *density estimation*, use minus the loglikelihood of the reconstruction

$$\mathcal{L} = -\sum_{i=1}^{d} x_i \log \tilde{x}_i + (1 - x_i) \log(1 - \tilde{x}_i) \tag{7}$$

[1] If you do regression, you don't have to apply sigmoid in the decoder.

Negative Loglikelihood Loss

Recall that the density of a Bernoulli distribution is given by

$$\mathrm{Bernoulli}(x; p) = p^x (1-p)^{1-x} \tag{8}$$

where $p \equiv p(x = 1)$.

Negative Loglikelihood Loss

If your have binary features, i.e. $x_i \in \{0, 1\}$ for $i = 1, ..., d$, then the output of the decoder can be interpreted has the parameter of a Bernoulli. Thus the likelihood of the input x can be compute has

$$p(x|z) = \prod_{i=1}^{d} p(x_i|z) = \prod_{i=1}^{d} \tilde{x}_i^{x_i}(1 - \tilde{x}_i)^{1-x_i} \tag{9}$$

Note that this works only if $\tilde{x}_i \in (0, 1)$. To ensure it, apply sigmoid element wise on the output of the decoder.

```
# in pytorch
loss = functional.binary_cross_entropy(param, x)
```

Negative Loglikelihood Loss

If your don't have binary features, e.g. $x_i \in \mathbb{R}$ for $i = 1, ..., d$, you need to binarize your input.

Generative Models

Represent the probability distribution of either $P(X, Y)$ or $P(X)$. In the case of *density estimation*, we are looking for a representation of

$$x \sim P_\theta(X)$$

For example, $x \sim \mathcal{N}(x; \boldsymbol{\mu}_{\text{mle}}, \boldsymbol{\sigma}_{\text{mle}})$

Problem: Most parametric distribution make strong (and often wrong) assumption about the distribution.

Latent variable models

We can model the distribution of x as a function of a latent variable z

$$p(x) = \int p_\theta(x|z)p(z)dz$$

where the distribution of z is chosen. A typical choice for z is

$$z \sim \mathcal{N}(z; \mathbf{0}, \boldsymbol{I})$$

Then we can train a model to learn a good representation of $p_\theta(x|z)$ with stochastic gradient.

Latent variable models

Once we have a good representation of $p_\theta(x|z)$, we can sample from $p(x)$ by first sampling

$$z' \sim p(z)$$

and then sampling

$$x' \sim p(x|z')$$

Latent variable models

Problem 1: To learn $p_\theta(x|z)$ using stochastic gradient, we need to know a good mapping

$$f : \mathcal{Z} \times \Theta \mapsto \mathcal{X}$$

In other word when we sample $x \sim p(x|z')$ we need to know which x is likely to be generated by this particular z' in order to train the model.

Latent variable models

Solution: the prior of the latent space can be written has

$$p(z) = \int p(z|x)p(x)dx$$

During training, We can sample z by sampling

$$x' \sim p(x)$$

and then

$$z \sim p(z|x')$$

The training set comes from $p(x)$ so we can sample from it. This will reduce the space of the latent variable a lot and allow the model to learn efficiently.

Latent variable models

Problem 2: $p_\theta(z|x)$ is intractable.

Solution: use an approximation $q_\phi(z|x)$

To summarize

- Sample $x \sim D_n$
- Sample $z \sim q_\phi(z|x)$
- Sample $\tilde{x} \sim p_\theta(x|z)$

The parameter to learn are ϕ and θ and they should be learn such that the marginal likelihood $p(x)$ is maximized.

Latent variable models

Before looking at how we can train this model efficiently, let's take a closer look at how it works concretely.

Probabilistic Encoder $q_\phi(z|x)$

- Example: Gaussian MLP as encoder
  - $\boldsymbol{h} = \text{relu}\left(xW_{xh} + b_{xh}\right)$
  - $\mu = hW_{hz}^{(1)} + b_{hz}^{(1)}$
  - $\log \sigma^2 = hW_{hz}^{(2)} + b_{hz}^{(2)}$
- $q_\phi(z|x) = N(z; \mu, \sigma^2)$
- $\phi = \{W_{hz}^{(1)}, b_{hz}^{(1)}, W_{hz}^{(2)}, b_{hz}^{(2)}, W_{xh}, b_{xh}\}$

```
class VAE:
    ...
    def encoder(self, x):
        # Encoder network. Return the parameter of q(z|x)
        h = relu(x @ self.Wxh + self.bxh.repeat(x.size(0), 1))
        mu = h @ self.Whz_mu + self.bhz_mu.repeat(x.size(0), 1)
        log_var = h @ self.Whz_var + self.bhz_var.repeat(x.size(0), 1)
        reture mu, log_var
```

Sampling $z \sim q_\phi(z|x)$

- Example: Gaussian MLP as encoder
  - $\boldsymbol{\mu}_z, \boldsymbol{\sigma}_z = \text{encoder}(x)$
  - $\epsilon \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$
  - $z = \boldsymbol{\mu}_z + \boldsymbol{\sigma}_z \odot \boldsymbol{\epsilon}$

```
class VAE:
...
def _sample_z(self, mu, log_var):
    epsilon = Variable(torch.randn(mu.size()).to(self._device)
    sigma = torch.exp(log_var / 2)
    return mu + sigma * epsilon
```

Probabilistic Decoder $p_\theta(x|z)$

- Example: Bernoulli MLP as decoder
  - $h = \mathrm{relu}\left(W_{zh}x + b_{zh}\right)$
  - $\gamma = \sigma\left(W_{zx}h + b_{zx}\right)$
- $p_\theta(x|z) = \mathrm{Bernoulli}(x; \boldsymbol{\gamma})$
- $\boldsymbol{\theta} = \{W_{zh}, b_{zh}, W_{zx}, b_{zx}\}$

```
class VAE:
    ...
    def decoder(self, z):
        # Decoder network. Reconstruct the input from
        # the latent variable z
        h = relu(z @ self.Whx + self.bhx.repeat(x.size(0), 1))
        gamma = h @ self.Whx_mu + self.bhx.repeat(x.size(0), 1)
        reture gamma
```

Training VAE

We need $p_\theta(x|z)$ to be such that the marginal likelihood $p(x)$ is maximized.

In other word the lost function should be

$$-\log p(x^{(1)}, ..., x^{(N)}) = -\sum_{i=1}^{N} \log p(x^{(i)})$$

Kullback-Leibler divergence

$$\mathcal{D}_{KL}[q(x)||p(x)] = \mathrm{E}_{x\sim q(x)}[\log q(x) - \log p(x)] \tag{10}$$

Gibbs' inequality:

$$\mathcal{D}_{KL}[q(x)||p(x)] \geq 0 \tag{11}$$

Training VAE

Now let's compute de Kullback-Leibler divergence $\mathcal{D}_{KL}$ between $p(z|x)$ and $q(z|x)$

$$
\begin{aligned}
&\mathcal{D}_{KL}[q(z|x)||p(z|x)] \\
&\qquad =\mathrm{E}_{z\sim q(z|x)}[\log q(z|x) - \log p(z|x)] \\
&\qquad =\mathrm{E}_{z\sim q(z|x)}[\log q(z|x) - \log p(x|z) - \log p(z) + \log p(x)] \\
&\qquad = \log p(x) + \mathrm{E}_{z\sim q(z|x)}[\log q(z|x) - \log p(z)] - \mathrm{E}_{z\sim q(z|x)} \log p(x|z) \\
&\qquad = \log p(x) - \mathcal{D}_{KL}[q(z|x)||p(z|x)] - \mathrm{E}_{z\sim q(z|x)} \log p(x|z)
\end{aligned}
$$

Training VAE

$$\log p(x) = \mathrm{E}_{z \sim q(z|x)} \log p(x|z) - \mathcal{D}_{KL}[q(z|x)||p(z)] + \mathcal{D}_{KL}[q(z|x)|| \log p(z|x)] \tag{12}$$

Because of Gibbs' inequality we have

$$\log p(x) \geq \mathrm{E}_{z \sim q(z|x)} \log p(x|z) - \mathcal{D}_{KL}[q(z|x)||p(z)]$$

Hence, our loss function is

$$\mathcal{L} = \mathrm{E}_{z \sim q(z|x)} \log p(x|z) - \mathcal{D}_{KL}[q(z|x)||p(z)]$$

Solution of $\mathcal{D}_{KL}[q(z|x)||p(z)]$

$$\mathcal{D}_{KL}[q(x)||p(x)] = \int q(x)(\log q(x) - \log p(x))dx$$

Training VAE

Suppose $z \in \mathbb{R}^J$ is normal

$$\int q(z|x) \log q(z|x) dz = \int \mathcal{N}(z;\mu,\sigma) \log \mathcal{N}(z;\mu,\sigma) = -\frac{J}{2}\log 2\pi - \frac{1}{2}\sum_{j=1}^{J}(1 + \log \sigma_j^2) \tag{13}$$

$$\int q(z|x) \log p(z) dz = \int \mathcal{N}(z;\mu,\sigma) \log \mathcal{N}(z;0,I) = -\frac{J}{2}\log 2\pi - \frac{1}{2}\sum_{j=1}^{J}(\mu_j^2 + \sigma_j^2) \tag{14}$$

Training VAE

$$
\begin{aligned}
\mathcal{D}_{KL}[q(x)||p(x)] =& (13) - (14) \\
=& \frac{1}{2} \sum \mu_j^2 + \sigma_j^2 - 1 - \log \sigma_j^2
\end{aligned}
\tag{15}
$$

```
def kl_divergence(mu, log_sigma):
    sigma = torch.exp(log_sigma)
    return .5 * torch.sum(mu**2 + sigma**2 - 1 - 2*log_sigma, axis=1)
```

Experiment

Anomaly detection and autoencoder

Some anomaly detection methods:

- Statistical: A data point is defined as an anomaly if the density of it being generated from the model is below a threshold
- Proximity based: A data point is defined as an anomaly if it is *islolated* (e.g. far from clusters centroid)
- Deviation based: Use reconstruction error to detect anomaly (e.g. k-most significant principle component (PCA) and autoencoders based methods)

Anomaly detection

Algorithm 1 Pseudocode for Batch Gradient Descent

Require: Learning rate ϵ_k
Require: Initial parameter $\boldsymbol{w}_0$
Require: Number of epochs T
  for $i = 1$ to T **do**
    Compute gradient $\boldsymbol{g}_t = \frac{1}{m} \nabla_w \sum_i L(h_{w_{t-1}}(\boldsymbol{x}^{(i)}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{w}_t = \boldsymbol{w}_{t-1} - \epsilon \boldsymbol{g}_t$
  end for