Variational Auto-Encoders

Stéphane d’Ascoli
Roadmap

1. A reminder on auto-encoders
   a. Basics
   b. Denoising and sparse encoders
   c. Why do we need VAEs?

2. Understanding variational auto-encoders
   a. Key ingredients
   b. The reparametrization trick
   c. The underlying math

3. Applications and perspectives
   a. Disentanglement
   b. Adding a discrete condition
   c. Applications
   d. Comparison with GANs

4. Do it yourself in PyTorch
   a. Build a basic denoising encoder
   b. Build a conditional VAE
Auto-Encoders
Basics

\[ \mathcal{L}(x) = \frac{1}{2} (x - \theta(\phi(x)))^2 \]
Denoising and Sparse Auto-Encoders

Denoising:

Sparse: enforces specialization of hidden units

$$\mathcal{L} (x, \hat{x}) + \lambda \sum_i |a_i^{(h)}|$$

Contractive: enforces that close inputs give close outputs

$$\mathcal{L} (x, \hat{x}) + \lambda \sum_i \left\| \nabla_x a_i^{(h)} (x) \right\|^2$$
Why do we need VAE?

VAE’s are used as generative models: sample a latent vector, decode and you have a new sample.

Q: Why can’t we use normal auto-encoders?
A: If we choose an arbitrary latent vector, we aren’t close to any points in the training set and the reconstruction is garbage!

Q: How can we avoid this?
A: Compactify the latent space!

Q: How can we do this?
A: Two ingredients:
1. Encode into balls rather than points
2. Bring the balls closer together
Variational Auto-Encoders
**Key Ingredients**

**Generative models**: unsupervised learning, aim to learn the distribution underlying the input data

**VAEs**: Map the complicated data distribution to a simpler distribution (encoder) we can sample from (Kingma & Welling 2014) to generate images (decoder)
**First Ingredient: Encode into Distributions**

Q: Why encode into distributions rather than deterministic values?

A1: This creates balls in latent space
A2: This ensures that close points in latent space lead to the same reconstruction. This gives “meaning” to the latent space.
**Second Ingredient : impose structure**

Q: How can I bring the balls together to compactify latent space?
A: Make sure that \( Q(z|x) \) for different \( x \)’s are close together!
**SECOND INGREDIENT: IMPOSE STRUCTURE**

Q: How do we keep the balls close together?
A: By adding springs the balls which pull them towards the center.

Q: How?
A: KL divergence with N(0,1) prior!

\[ \mathcal{L}_{KL} = \mathbb{E}_{x \sim \text{dataset}} \left[ D_{KL} \{ Q_\phi(z|x) || p(z) \} \right], p \sim \mathcal{N}(0,1) \]
The Reparametrization Trick

Q: How can we backpropagate when one of the nodes is non-deterministic?
A: Put the random process outside the network!
The Underlying Information Theory

How can we make a latent variable model?

Choose a nice simple latent distribution $P(z)$, for example $P(z) \sim \mathcal{N}(0, I_d)$, try to find its mapping to real space, $P(x|z)$, by maximizing the likelihood of observing the dataset under this generative process:

$$
\mathcal{L} = \mathbb{E}_{x \sim \mathcal{D}} \log P(x) = \mathbb{E}_{x \sim \mathcal{D}} \log \int_z P(x|z)P(z)dz
$$

To do this, we could parametrize $P(x|z)$ by a neural network and perform gradient ascent on $\mathcal{L}$. Problem: we can’t calculate the integral in $\mathcal{L}$ for arbitrary $P(x|z)$! We could estimate it with sampling but that would be very inaccurate in high dimension.
The Underlying Information Theory

To do this, we could parametrize $P(x|z)$ by a neural network and perform gradient ascent on $\mathcal{L}$. Problem: we can’t calculate the integral in $\mathcal{L}$ for arbitrary $P(x|z)$! We could estimate it with sampling but that would be very inaccurate in high dimension.

The idea is to circumvent this by introduce another distribution $Q(z)$ and exploit the following lower bound, valid for any $Q(z)$:

$$\mathcal{L} \geq \mathbb{E}_{x \sim D} [\mathbb{E}_{z \sim Q} \log P(x|z) - \beta \mathcal{D}_{KL}(Q(z)||P(z))] = \text{ELBO}$$

This time, we have two things to optimize simultaneously: $Q(z|x)$ (the encoder) and $P(x|z)$ (the decoder)! Why is this nicer? Because both terms that appear are tractable
The Underlying Information Theory

\[ \mathcal{L} \geq \mathbb{E}_{x \sim D} \left[ \mathbb{E}_{z \sim Q} \log P(x|z) - \beta D_{KL}(Q(z)||P(z)) \right] = \text{ELBO} \]

- The first term is called the reconstruction loss. Wait, this is silly because we have an intractable integral over \( z \) again! Yes, but this time we have freedom to make \( Q(z) \) dependent on \( x \). Whereas before we had to integrate over the whole latent space, here we only have to integrate over a small ball containing values likely to reconstruct \( x \):

\[ Q(z|x) \sim \mathcal{N}(\mu(x), \Sigma(x)) \tag{1} \]

If the ball is small enough we can estimate the average just by sampling one point!
The Underlying Information Theory

\[ \mathcal{L} \geq \mathbb{E}_{x \sim \mathcal{D}} \left[ \mathbb{E}_{z \sim Q} \log P(x|z) - \beta D_{KL}(Q(z)||P(z)) \right] = \text{ELBO} \]

- The second term is called the regularization term. Since both \( P(z) \) and \( Q(z|x) \) are tractable we have an easy analytic formula:

\[
\mathcal{D}[\mathcal{N}(\mu(x), \Sigma(x))||\mathcal{N}(0, I_d)] = \frac{1}{2} \left( \text{tr}(\Sigma(x)) + (\mu(x))^\top(\mu(x)) - d - \log \det(\Sigma(x)) \right)
\]

(2)
The Underlying Information Theory

Proof of lower bound:

\[ D_{KL}(Q(z)\|P(z|x)) = \sum_z Q(z) \log \frac{Q(z)}{P(z|x)} \]

\[ = \log P(x) + \sum_z Q(z) \log \frac{Q(z)}{P(z)} - \sum_z Q(z) \log P(x|z) \]

\[ = \log P(x) + D_{KL}(Q(z)\|P(z)) - \mathbb{E}_{z \sim Q} \log P(x|z) \]

\[ \text{ELBO} \]

\[ \mathbb{E}_{x \sim D} \log P(x) = \mathbb{E}_{x \sim D} [D_{KL}(Q(z)\|P(z|x)) + \text{ELBO}] \]

\[ \geq \text{ELBO} \]
VAEs in Practice
**Disentanglement : Beta-Vae**

We saw that the objective function is made of a reconstruction and a regularization part.

$$\mathcal{L} = \mathbb{E}_{z \sim Q} \log P(x|z) - \beta D_{KL}(Q(z|x) || P(z))$$

By adding a tuning parameter we can control the tradeoff.

If we increase beta:
- The dimensions of the latent representation are more disentangled
- But the reconstruction loss is less good
Generating Conditionally: CVAEs

Add a one-hot encoded vector to the latent space and use it as categorical variable, hoping that it will encode discrete features in data (digits in MNIST)

Q: The usual reparametrization trick doesn’t work here, because we need to sample discrete values from the distribution! What can we do?
A: Gumbel-Max trick

Q: How do I balance the regularization terms for the continuous and discrete parts?
A: Control the KL divergences independently

\[ \mathcal{L}(\theta, \phi) = \mathbb{E}_{q_{\phi}(z, c|x)}[\log p_{\theta}(x|z, c)] - \gamma D_{KL}(q_{\phi}(z|x) \| p(z)) - C_z - \gamma D_{KL}(q_{\phi}(c|x) \| p(c)) - C_c \]
Applications

Image generation: Dupont et al. 2018

**Comparison with GANS**

<table>
<thead>
<tr>
<th>VAE</th>
<th>GAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Easy metric: reconstruction loss</td>
<td>Metric is hard to interpret</td>
</tr>
<tr>
<td>Interpretable and disentangled latent space</td>
<td>Low interpretability</td>
</tr>
<tr>
<td>Easy to train</td>
<td>Tedious hyperparameter searching</td>
</tr>
<tr>
<td>Noisy generation</td>
<td>Clean generation</td>
</tr>
</tbody>
</table>
Towards a Mix of the Two?

Prominent attributes: White, Male, Curly Hair, Frowning, Eyes Open, Pointy Nose, Flash, Posed Photo, Eyeglasses, Narrow Eyes, Teeth Not Visible, Senior, Receding Hairline.
Do It Yourself In Pytorch
### Auto-Encoder

1. **Example:** a *simple fully-connected* auto-encoder

```python
loss_fn = torch.nn.MSELoss()

def train_model(model, loss_fn, data_loader=None, epochs=1, optimizer=None):
    model.train()
    for epoch in range(epochs):
        for batch_idx, (data, _) in enumerate(data_loader):
            data = data.view([-1, 784])
            optimizer.zero_grad()
            output = model(data)
            loss = loss_fn(output, data)
            loss.backward()
            optimizer.step()
            if batch_idx % 50 == 0:
                print('Train Epoch: {} [{}/{} ({:.0f}%)]
                      Loss: {:.6f}'.format(
                    epoch, batch_idx * len(data), len(data_loader.dataset),
                    100. * batch_idx / len(data_loader), loss.data.item()))
```

2. **DIY:** implement a *denoising convolutional* auto-encoder for MNIST

```python
class AutoEncoder(nn.Module):
    def __init__(self, input_dim, encoding_dim):
        super(AutoEncoder, self).__init__()
        self.encoder = nn.Linear(input_dim, encoding_dim)
        self.decoder = nn.Linear(encoding_dim, input_dim)

    def forward(self, x):
        encoded = F.relu(self.encoder(x))
        decoded = self.decoder(encoded)
        return decoded
```
Variational Auto-Encoder

1. Example: a simple VAE

```python
class VAE(nn.Module):
    def __init__(self, image_size=784, h_dim=400, z_dim=20):
        super(VAE, self).__init__()
        self.fc1 = nn.Linear(image_size, h_dim)
        self.fc2 = nn.Linear(h_dim, z_dim)
        self.fc3 = nn.Linear(h_dim, z_dim)
        self.fc4 = nn.Linear(z_dim, h_dim)
        self.fc5 = nn.Linear(h_dim, image_size)

    def encode(self, x):
        h = F.relu(self.fc1(x))
        return self.fc2(h), self.fc3(h)

    def reparameterize(self, mu, log_var):
        std = torch.exp(log_var / 2)
        eps = torch.randn_like(std)
        return mu + eps * std

    def decode(self, z):
        h = F.relu(self.fc4(z))
        return torch.sigmoid(self.fc5(h))

    def forward(self, x):
        mu, log_var = self.encode(x)
        z = self.reparameterize(mu, log_var)
        x_reconst = self.decode(z)
        return x_reconst, mu, log_var
```

2. DIY: implement a conditional VAE for MNIST

```python
def train(model, data_loader, num_epochs):
    for epoch in range(num_epochs):
        for i, (x, _) in enumerate(data_loader):

            # Forward pass
            x = x.to(device).view(-1, image_size)
            x_reconst, mu, log_var = model(x)

            # Compute reconstruction loss and kl divergence
            reconstr_loss = F.binary_cross_entropy(x_reconst, x, reduction='sum')
            kl_div = -0.5 * torch.sum(1 + log_var - mu.pow(2) - log_var.exp())

            # Backprop and optimize
            loss = reconstr_loss + kl_div
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()```
Questions