Autoencoder
and
Variational Autoencoder

Physics 361 Machine Learning in Physics
Jacky Yip
hyip2@wisc.edu
3/7/2024
A quick look at the autoencoder
A quick look at the autoencoder

\[ \text{encoder} \quad e_\theta(x) \]

latent vector

\[ \text{decoder} \quad d_\phi(z) \]

\[ \text{loss} = \| x - \hat{x} \|_2^2 \]
Why autoencoder (AE)

• Compression / dimensionality reduction
  • Curse of dimensionality
    • Exponential increase in required # of samples / peaking in predictive power
    • Loss of meaning for the distance function
  • Nonlinear analogue of PCA

• Applications
  • Layer-wise (pre)training for deep networks (depreciated due to batch normalization & ResNet)
  • Training data preprocessing (compression, denoising)
  • Anomaly detection
  • As a generative model (esp. variational autoencoder)

\[
\frac{V_{\text{hypersphere}}}{V_{\text{hypercube}}} = \frac{\pi^{d/2}}{d^{2d-1} \Gamma(d/2)} \rightarrow 0
\]
AE – architectures

- Dimension of the latent space
  - Undercomplete & overcomplete autoencoders

- The encoder & decoder are just compression & decompression functions to be approximated by NNs

- Architecture depends on how data is represented
  - A vector: multilayer perceptron
  - A field: convolutional neural network
  - A graph: graph neural network
AE – implementations

**MLP**

```python
class autoencoder(nn.Module):
    def __init__(self):
        super(autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(28 * 28, 128),
            nn.ReLU(True),
            nn.Linear(128, 64),
            nn.ReLU(True),
            nn.Linear(64, 12),
            nn.ReLU(True),
            nn.Linear(12, 3))
        self.decoder = nn.Sequential(
            nn.Linear(3, 12),
            nn.ReLU(True),
            nn.Linear(12, 64),
            nn.ReLU(True),
            nn.Linear(64, 128),
            nn.ReLU(True),
            nn.Linear(128, 28 * 28),
            nn.Tanh())

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

**CNN**

```python
class autoencoder(nn.Module):
    def __init__(self):
        super(autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 16, 3, stride=1, padding=1),  # b, 16, 10, 10
            nn.ReLU(True),
            nn.MaxPool2d(2, stride=2),  # b, 16, 5, 5
            nn.Conv2d(16, 8, 3, stride=1, padding=1),  # b, 8, 3, 3
            nn.ReLU(True),
            nn.MaxPool2d(2, stride=1)  # b, 8, 2, 2
        )
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(8, 16, 3, stride=2),  # b, 16, 5, 5
            nn.ReLU(True),
            nn.ConvTranspose2d(16, 8, 3, stride=1, padding=1),  # b, 8, 15, 15
            nn.ReLU(True),
            nn.ConvTranspose2d(8, 1, 2, stride=2, padding=1),  # b, 1, 28, 28
            nn.Tanh()
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```
AE – implementations

```python
model = autoencoder().cuda()
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate,
                          weight_decay=1e-5)

for epoch in range(num_epochs):
    total_loss = 0
    for data in dataloader:
        img, _ = data
        img = Variable(img).cuda()
        # ===============forward=======================
        output = model(img)
        loss = criterion(output, img)
        # ===============backward======================
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        total_loss += loss.data
```
AE – reconstruction examples

MNIST dataset

AE

MLP
blurry?

CNN
more details?
AE – latent space

Latent space of AE

MNIST dataset
AE – latent space

- **Clustering**: same digits are close to each other in the latent space
- **Empty space**: regions outside of clusters cannot be used for data generation – the latent space is not regularized

MNIST dataset

- a lot of random garbage
AE – regularizations

Vanilla: \( L(\theta, \phi) := \mathbb{E}_{x \sim \mu_{\text{ref}}} \left[ d(x, D_{\theta}(E_{\phi}(x))) \right] \)

With regularizations:

- Denoising: map noised data to data
  \[
  L(\theta, \phi) = \mathbb{E}_{x \sim \mu_{X}, T \sim \mu_{T}} \left[ d(x, (D_{\theta} \circ E_{\phi} \circ T)(x)) \right]
  \]

- Contractive: small change in encoder output for small change in input
  Add \( L_{\text{contractive}}(\theta, \phi) = \mathbb{E}_{x \sim \mu_{\text{ref}}} \left\| \nabla_x E_{\phi}(x) \right\|_F^2 \)

- Sparsity regularization for overcomplete AEAs: a way of compressing by deactivating neurons instead of imposing an explicit bottleneck
  Add \( L_{\text{sparsity}}(\theta, \phi) = \mathbb{E}_{x \sim \mu_{X}} \left[ \sum_{k=1:K} w_k s(\hat{\rho}_k, \rho_k(x)) \right] \)
  where \( \rho_k(x) = \frac{1}{n} \sum_{i=1}^{n} a_{k,i}(x) \)
  \[
  s(\rho, \hat{\rho}) = KL(\rho || \hat{\rho})
  \]
Variational AE – completely regularizing the latent space

- Regions outside of the distribution cannot be used for data generation
- We must restrict ourselves within the distribution
- **Learn the distribution directly!**
Variational AE

force the latent distribution to be unit Gaussian, the encoder learns to map data to the mean and std vectors

easy to work with (sampling, math, ...)

2 2 2

as desired

note: before training is done, each x will have its own Gaussian

reconstruction loss = \| x - \hat{x} \|_2^2 = \| x - d_\phi(z) \|_2^2

similarity loss = KL Divergence = D_{KL}(\mathcal{N}(\mu_x, \sigma_x) \parallel \mathcal{N}(0, I))

loss = reconstruction loss + similarity loss
Variational AE

Data generator:

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

as desired

sampling

latent vector

decoder $d_\phi(z)$

trained

\hat{x}

generated data
VAE – reparameterization trick

without reparameterization:
backpropagation?
VAE – reparameterization trick

without reparameterization:
backpropagation?

with reparameterization
backpropagation OK!

Sample $z$ from $\mathcal{N}(\mu(X), \Sigma(X))$

stochastic node

Sample $\epsilon$ from $\mathcal{N}(0, I)$

external input
VAE – generated data examples
VAE – latent space

Latent space of VAE with KL loss

MNIST dataset

looks like a unit Gaussian... more or less
class VAE(nn.Module):
    def __init__(self):
        super(VAE, self).__init__()
        self.fc1 = nn.Linear(784, 400)
        self.fc21 = nn.Linear(400, 20)
        self.fc22 = nn.Linear(400, 20)
        self.fc3 = nn.Linear(20, 400)
        self.fc4 = nn.Linear(400, 784)
    
    def encode(self, x):
        h1 = F.relu(self.fc1(x))
        return self.fc21(h1), self.fc22(h1)
    
    def reparameterize(self, mu, logvar):
        std = logvar.mul(0.5).exp_()
        if torch.cuda.is_available():
            eps = torch.cuda.FloatTensor(std.size()).normal_(0)
        else:
            eps = torch.FloatTensor(std.size()).normal_(0)
        eps = Variable(eps)
        return eps.mul(std).add_(mu)
    
    def decode(self, z):
        h3 = F.relu(self.fc3(z))
        return F.sigmoid(self.fc4(h3))
    
    def forward(self, x):
        mu, logvar = self.encode(x)
        z = self.reparameterize(mu, logvar)
        return self.decode(z), mu, logvar

def loss_function(recon_x, x, mu, logvar):
    recon_x = generating images
    x: origin images
    mu: latent mean
    logvar: latent log variance
    BCE = reconstruction_function(recon_x, x) # mse loss
    # loss = 0.5 * sum(1 + log(sigma^2) - mu^2 - sigma^2)
    KLD_element = mu.pow(2).add_(logvar.exp()).mul_(-1).add_(1).add_(logvar)
    KLD = torch.sum(KLD_element).mul_(-0.5)
    # KL divergence
    return BCE + KLD

for epoch in range(num_epochs):
    model.train()
    train_loss = 0
    for batch_idx, data in enumerate(dataloader):
        img, _ = data
        img = img.view(img.size(0), -1)
        img = Variable(img)
        optimizer.zero_grad()
        recon_batch, mu, logvar = model(img)
        loss = loss_function(recon_batch, img, mu, logvar)
        loss.backward()
        train_loss += loss.data[0]
        optimizer.step()
        if batch_idx % 100 == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]
              Loss: {:.6f}'.format(epoch, batch_idx * len(img), len(dataloader.dataset), 100. * batch_idx / len(dataloader), train_loss / len(img)))
**VAE – a Bayesian understanding**

1. We want to generate data from the distribution $p(X)$
   - We only have samples (training data), hard to guess a formula for sampling

2. In VAE the data generator is the decoder, and we decide to sample the latent distribution $p(Z)$
   - We can write $p(X) = \sum_{Z} p(X|Z)p(Z) = \text{decoder} \cdot \text{latent distribution}$
     - we want this to be easy to sample

3. The encoder generates the latent variable. In terms of probability, the encoder is $p(Z|X) = \text{encoder} = \text{encoder’s posterior given the input X}$
   - we fix a parameterization of the posterior, and have the encoder spit out the parameters according to input

4. Now think about what happens if we train only with reconstruction loss
VAE – a Bayesian understanding

1. We want to generate data from the distribution $p(X)$
   • We only have samples (training data), hard to guess a formula for sampling

2. In VAE the data generator is the decoder, and we decide to sample the latent distribution $p(Z)$
   • We can write $p(X) = \sum_Z p(X|Z)p(Z) = \text{decoder} \bullet \text{latent distribution}$

3. The encoder generates the latent variable. In terms of probability, the encoder is $p(Z|X) = \text{encoder} = \text{encoder’s posterior given the input X}$
   we fix a parameterization of the posterior, and have the encoder spit out the parameters according to input

4. Now think about what happens if we train only with reconstruction loss
   the encoder will learn to be “deterministic” by, e.g., setting std to zero! ➔ just an AE!
VAE – a Bayesian understanding

\[ p(Z|X) = \text{encoder} \]

\[ p(X) = \sum_z p(X|Z)p(Z) = \text{decoder \• latent distribution} \]

for any \( X \)

• The solution is to fix \( p(Z|X) = \mathcal{N}(0, I) \) by the similarity loss

• This is great, because:
  
  • \( p(Z) = \sum_X p(Z|X)p(X) = \sum_X \mathcal{N}(0, I)p(X) = \mathcal{N}(0, I) \sum_X p(X) = \mathcal{N}(0, I) \)

is indeed what we plan to sample \( Z \) from

\[ D_{KL}(\mathcal{N}(\mu_x, \sigma_x) \parallel \mathcal{N}(0, I)) \]

Gaussian is a simple choice; a uniform distribution probably won’t work
VAE – a Bayesian understanding

- And the training dynamics is right
  - The reconstruction loss is counteracting the similarity loss!

\[
\text{reconstruction loss} > \text{similarity loss} \\
\text{learning} \Rightarrow \text{reconstruction loss} \downarrow + \text{similarity loss} \uparrow \\
\text{[lower the std (increase KL) makes it easier to reconstruct]}
\]

\[
\text{similarity loss} > \text{reconstruction loss} \\
\text{learning} \Rightarrow \text{similarity loss} \downarrow + \text{reconstruction loss} \uparrow \\
\text{[increase the std makes it harder to reconstruct]}
\]

reconstruction loss hates noise (std); similarity loss wants noise