A Bestiary of Autoencoders

Bastian Rieck

Pseudomanifold

Motivation

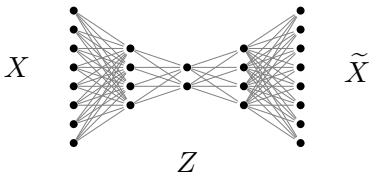
‘A bestiary (from *bestiarum vocabulum*) is a compendium of beasts.’

In this talk

- What are autoencoders?
- What type of autoencoder architectures exist?
- How to use them in practice?

Autoencoders

Dramatis personæ



Terminology

- $X \in \mathbb{R}^D$: input data
- $Z \in \mathbb{R}^d$: latent representation
- $\widetilde{X} \in \mathbb{R}^D$: reconstructed data

Properties

- Typically, $D \gg d$.
- Use loss function $\mathcal{L}\left(X, \widetilde{X}\right)$ to measure quality of reconstruction.

Why autoencoders?

- Encoder–decoder architecture (we learn the *identity* function).
- 'Middle' layer serves as *bottleneck* or *latent representation*.
- Latent representations can be used for visualisation, interpolation, clustering, and much more.

A simple autoencoder

- Encoder: linear transformation $\mathbb{R}^D \to \mathbb{R}^2$
- Decoder: linear transformation $\mathbb{R}^2 \to \mathbb{R}^D$
- Loss function: mean squared error, $\mathcal{L}\left(X, \widetilde{X}\right) := \left\| X - \widetilde{X} \right\|_2^2$

A simple autoencoder

Some reconstructions

0 epochs

A simple autoencoder

Some reconstructions

10 epochs

A simple autoencoder

Some reconstructions

20 epochs

A simple autoencoder

Some reconstructions

30 epochs

A simple autoencoder

Some reconstructions

40 epochs

A simple autoencoder

Some reconstructions

50 epochs

But how to train this?

PyTorch Lightning

Overall training loop

```
from .data import MNISTDataModule
from .models import LinearAutoencoder
from pytorch_lightning import Trainer

if __name__ == '__main__':
    dm = MNISTDataModule()
    dm.prepare_data()
    dm.setup()

    model = LinearAutoencoder(input_dim=dm.input_dim, bottleneck_dim=2)

    # There's a lot more parameters for a `trainer` class, but this is
    # sufficient for our brief example.
    trainer = Trainer(max_epochs=50)
    trainer.fit(model, dm)
```

PyTorch Lightning

Defining a model

```
import torch.nn as nn
import torch.nn.functional as F
import pytorch_lightning as pl

class LinearAutoencoder(pl.LightningModule):
  def __init__(self, input_dim, bottleneck_dim=2, lr=0.01):
      super().__init__()

      self.encoder = nn.Linear(input_dim, bottleneck_dim)
      self.decoder = nn.Linear(bottleneck_dim, input_dim)

      self.lr = 0.01
      self.loss_fn = F.mse_loss
```

PyTorch Lightning

Defining a model, continued

```
    def forward(self, x):
        z = self.encoder(x)
        x_hat = self.decoder(z)
        loss = self.loss_fn(x_hat, x)
        return x_hat, z, loss

    def training_step(self, batch, batch_idx):
        x, y = batch

        x = x.view(x.size(0), -1)
        _, _, loss = self(x)

        self.log('train_loss', loss)
        return loss

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.parameters(), lr=self.lr)
        return optimizer
```

PyTorch Lightning

More complex operations

Some highlights

- Early stopping
- Learning rate adjustments
- Model checkpoints
- Distributed training

Variational autoencoders

We can make the autoencoder learn the *parameters* of our input distribution to sample *new* data points from it.

Variational autoencoders

The basic skeleton

```
class BetaVAE(pl.LightningModule):
    def __init__(self, input_dim, bottleneck_dim=32, beta=1, lr=1e-3):
        super(BetaVAE, self).__init__()

        self.bottleneck_dim = bottleneck_dim

        # The specific choice of encoder/decoder is irrelevant here.
        self.encoder = ...
        self.decoder = ...

        # Final layer for encoding the parameters of the distribution
        self.fc_mu = nn.Linear(16, bottleneck_dim)
        self.fc_logvar = nn.Linear(16, bottleneck_dim)

        # For sampling values that parametrise our distribution
        self.fc_z = nn.Linear(bottleneck_dim, 16)
```

Variational autoencoders

Encoding and decoding

```
def encode(self, x):
    x = self.encoder(x)
    x = x.view(x.shape[0], -1)
    return self.fc_mu(x), self.fc_logvar(x)

def sample(self, mu, logvar):
    std = torch.exp(0.5 * logvar)  # e^(1/2 * log(std^2))
    eps = torch.randn_like(std)    # random ~ N(0, 1)
    return eps.mul(std).add_(mu)

def decode(self, z):
    z = self.fc_z(z)
    z = z.view(z.shape[0], -1)
    return self.decoder(z)

def forward(self, x):
    mu, logvar = self.encode(x)
    z = self.sample(mu, logvar)
    x_hat = self.decode(z)
    return x_hat, mu, self.loss_fn(x_hat, x, mu, logvar)
```

Variational autoencoders

Loss function

```
def loss_fn(self, x_hat, x, mu, logvar):
    reconstruction_loss = F.mse_loss(
        x_hat,
        x,
        reduction='sum' # We reduce it ourselves later on!
    )

    # This follows the loss provided by Kingma and Welling in
    # 'Auto-Encoding Variational Bayes', Appendix B. The loss
    # assumes that all distributions are Gaussians.
    kl = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())

    return (reconstruction_loss + self.beta * kl) / x.shape[0]
```

The loss term boils down to a Kullback–Leibler divergence between the prior distribution and the posterior distribution:

$$\mathrm{KL}\left(q_\phi(\mathbf{z}) \parallel p_\theta(\mathbf{z})\right)$$

Variational autoencoders

Some reconstructions

0 epochs

Variational autoencoders

Some reconstructions

10 epochs

Variational autoencoders

Some reconstructions

20 epochs

Variational autoencoders

Some reconstructions

30 epochs

Variational autoencoders

Some reconstructions

40 epochs

Variational autoencoders

Some reconstructions

50 epochs

Sampling from the learned distribution

Pick $z \sim \mathcal{N}(0, 1)$ and *decode* it. This yields a valid image of the distribution!

Finis coronat opus

- Taming the dragon: PyTorch Lightning makes working with neural networks easy
- Autoencoders are available in many flavours (variational, topological, ...)
- *Acta non verba*: try them out yourselves!

`https://github.com/Pseudomanifold/bestiary-autoencoders`