Midterm Project: β-VAE
Armaan Kohli - ECE471 Computational Graphs for Machine Learning
Autumn 2019

Remarks

We attempted to replicate results from *β-VAE: Learning Basic Visual Concepts with a Constrained Variational Framework*[1]. Published in 2017 at ICLR, the team at Google DeepMind demonstrated that variational autoencoders, first described in [2], had the ability to produce 'disentangled' representations by augmenting the loss function described in [2] with an additional hyper-parameter, β. In a follow-up publication, [3], DeepMind explains this phenomena further. Effectively, the parameter β finds latent components which make different contributions to the log-likelihood term of the objective function in [2], and that latent components correspond to features that are qualitatively different. Below is the augmented ELBO objective presented in [1] for reference.

$$\mathcal{F}(\theta, \phi, \beta; \boldsymbol{x}, \boldsymbol{z}) \geq \mathcal{L}(\theta, \phi; \boldsymbol{x}, \boldsymbol{z}) = \mathbb{E}_{q_\phi(z|x)}[log_{p_\theta}(\boldsymbol{x}|\boldsymbol{z})] - \beta D_{KL}(q_\phi(\boldsymbol{z}|\boldsymbol{x})||p(\boldsymbol{z})) \tag{1}$$

In their publication, DeepMind illustrates the effectiveness of their network on the 3DChairs and CelebA datasets. We have attempted to replicate Fig. 1 and Fig. 2 of [1] for the β-VAE in order to demonstrate that our implementation of the network can faithfully replicate their results.

There were several parameters needed for implementation that the paper neglected to mention. The authors didn't state their batch size, training time/gradient steps or compute resources used. So, we opted to use a gpu and train for as long as possible, saving checkpoints and outputs along the way. They also didn't mention the dimensionality of the latent space, nor how they sampled or traversed the latent space to generate Fig. 1 and Fig. 2 in [1]. As such, we tried two different methods to replicate their results: We took random samples from a 7 dimensional latent space and we took the first n samples from the same latent space. We found that these unknown parameters had a significant impact on our results.

To see the full codebase, please visit github.com/armaank/bVAE. See the *Code* section for selected code snippets. We elected to implement our version of β-VAE in pytorch, for the learning experience and to cut time spent making a dataloader (since we were working with multiple datasets) in tensorflow, which by contrast, is easily done in pytorch. The CelebA dataset was trained using Ali's computer (GTX2070), and the 3DChairs dataset was trained on both a P100 on a Google Cloud VM instance and Ali's computer.

Results & Discussion

3DChairs

After training for 500,000k gradient steps (amounting to approximately 50 hours of training time), we have the following results for the 3DChairs Dataset

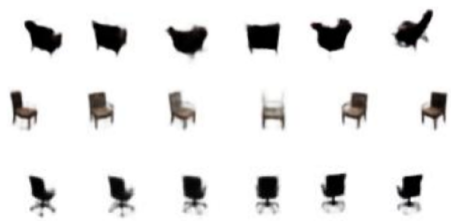

Figure 1: Qualitative results examining the disentangling performance of β-VAE. Here, we see the VAE is able to disentangle azimuthal direction from other factors.

Figure 2: Qualitative results examining the disentangling performance of β-VAE. Here, we see the VAE is able to disentangle width from other factors.

Figure 3: Qualitative results examining the disentangling performance of β-VAE. Here, we see the VAE is able to disentangle chair leg style. This is noteable, since other generative models were unable to learn this unlabled factor.

Comparing the visual results that we generated to those produced by DeepMind, we see that we were able to successfully reproduce their results. We observe that the β-VAE is able to produce disentangled representations. Furthurmore, we were able to replicate their observation that β-VAE is able to learn unlabled factors, like chair leg style.

For reference, the results from [1] are depicted below.

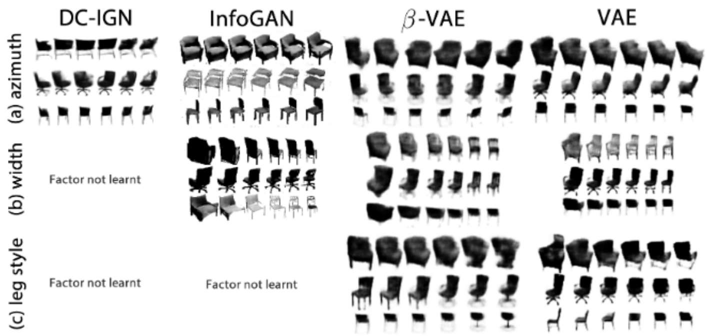

Figure 4: 3dChair results published by DeepMind in [1]

CelebA

We were not able to successfully reproduce the results for the CelebA dataset. Our attempt is depicted below in fig. 5

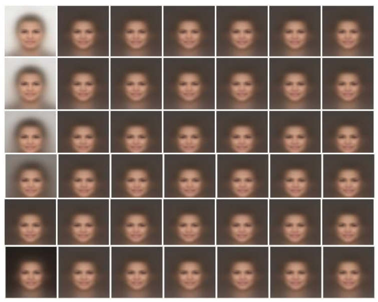

Figure 5: Poorly generated faces

We suspect that this occured failure occured primarily because of the choice of β. In [1], DeepMind reports that the faces in Fig. 6 were generated using $\beta = 250$, substantially more than the β used in the successful 3dChairs experiment ($\beta = 4$). In [3], the follow-up paper, they explain that setting beta too high can have adverse effects, and recomend adding an additional hyperparameter to regulate the loss furthur. Additionally, other implementaitons of β-VAE we found online were able to successfully generate results comprable to Fig. 6 using small β values, such as $\beta = 4$ or 5. We also don't know under what training conditions DeepMind was able to generate the faces in 6, which can certianly play a factor in the quality of the results.

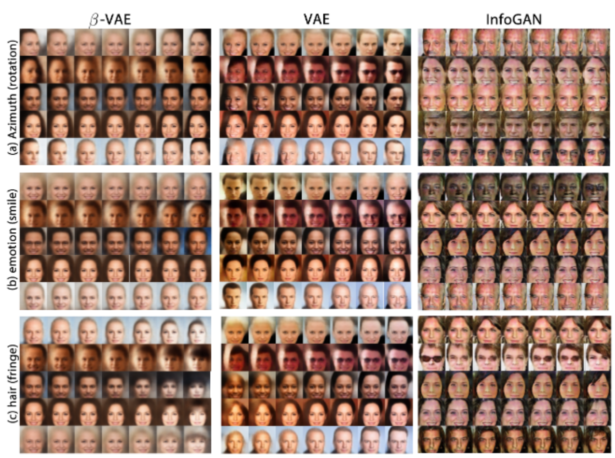

Figure 6: faces generated by DeepMind

Thus, following the papers recomendations and architecture choices, we were unable to replicate the CelebA results.

Conclusion

We were able to successfully reproduce Fig. 2 of [1] for the β-VAE in order to demonstrate that our implementation faithfully replicates their results. However, we were unable to replicate Fig. 1 of [1] using the information provided in the literature.

References

[1] I. Higgins, L. Matthey, A. Pal, C. Burgess, X. Glorot, M. M. Botvinick, S. Mohamed, and A. Lerchner, "beta-vae: Learning basic visual concepts with a constrained variational framework," in *ICLR*, 2017.

[2] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *CoRR*, vol. abs/1312.6114, 2013.

[3] C. P. Burgess, I. Higgins, A. Pal, L. Matthey, N. Watters, G. Desjardins, and A. Lerchner, "Understanding disentangling in beta-vae," *ArXiv*, vol. abs/1804.03599, 2018.

Credits

Ali, for his friendship and gpu :)

Appendix A: Code

Here is a code snippet showing the model architecture we replicated from the appendix of [1].

```
"""
model.py

contains network architecture for beta vae, as described in the appendix of [2]

"""
import torch
import torch.nn as nn
from torch.autograd import Variable
import torch.nn.init as init


def reparam(mu, logvar):
    """reparametization 'trick'

    allows optimization through sampling process.
    inputs: mean and variance
    outputs: random var with perscribed mean and noisy variance terms

    """

    std = logvar.div(2).exp()
    eps = Variable(std.data.new(std.size()).normal_())

    return mu + std * eps


class View(nn.Module):
    """View

    acts like tf/np reshape

    """

    def __init__(self, size):
        super(View, self).__init__()
        self.size = size

    def forward(self, tensor):
        return tensor.view(self.size)

```

```
class betaVAE(nn.Module):
    """betaVAE

    class used to setup the betaVAE architecture

    """

    def __init__(self, z_dim=10, nchan=1):
        super(betaVAE, self).__init__()
        self.z_dim = z_dim

        self.encoder = nn.Sequential(
            nn.Conv2d(nchan, 32, 4, 2, 1),
            nn.ReLU(True),
            nn.Conv2d(32, 32, 4, 2, 1),
            nn.ReLU(True),
            nn.Conv2d(32, 32, 4, 2, 1),
            nn.ReLU(True),
            nn.Conv2d(32, 32, 4, 2, 1),
            nn.ReLU(True),
            View((-1, 32 * 4 * 4)),
            nn.Linear(32 * 4 * 4, 256),
            nn.ReLU(True),
            nn.Linear(256, 256),
            nn.ReLU(True),
            nn.Linear(256, z_dim * 2),
        )

        self.decoder = nn.Sequential(
            nn.Linear(z_dim, 256),
            View((-1, 256, 1, 1)),
            nn.ReLU(True),
            nn.ConvTranspose2d(256, 64, 4),
            nn.ReLU(True),
            nn.ConvTranspose2d(64, 64, 4, 2, 1),
            nn.ReLU(True),
            nn.ConvTranspose2d(64, 32, 4, 2, 1),
            nn.ReLU(True),
            nn.ConvTranspose2d(32, 32, 4, 2, 1),
            nn.ReLU(True),
            nn.ConvTranspose2d(32, nchan, 4, 2, 1),
        )
```

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

        propgates input through the network
        inputs: sample input
        output: reconstructed input, mu and var from the latent space

        """
        dist = self.encode(x)
        mu = dist[:, : self.z_dim]
        logvar = dist[:, self.z_dim :]
        z = reparam(mu, logvar)
        x_recon = self.decode(z)

        return x_recon, mu, logvar

    def encode(self, x):
        return self.encoder(x)

    def decode(self, z):
        return self.decoder(z)


if __name__ == "__main__":
    pass
```

```
"""
losses.py
methods used to form the objective function described in [1]
"""
import torch

import torch.optim as optim
import torch.nn.functional as F
from torch.autograd import Variable
from torchvision.utils import make_grid, save_image


def r_loss(x, x_recon):
    """r_loss
    computes reconstruction loss described in [1]
    inputs: x, x_recon
    outputs: loss
    """

    batch_size = x.size(0)
    x_recon = torch.sigmoid(x_recon)
    recon_loss = F.mse_loss(x_recon, x, size_average=False).div(batch_size)

    return recon_loss


def kl_div(mu, logvar):
    """kl_div
    computes the kullback leibler divergence as part of the loss fcn
    inputs: mean and variance
    outputs: kld
    """

    if mu.data.ndimension() == 4:
        mu = mu.view(mu.size(0), mu.size(1))
    if logvar.data.ndimension() == 4:
        logvar = logvar.view(logvar.size(0), logvar.size(1))

    klds = -0.5 * (1 + logvar - mu.pow(2) - logvar.exp())
    total_kld = klds.sum(1).mean(0, True)

    return total_kld
```