

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.

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

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 500,000 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 notable, since other generative models were unable to learn this unlabeled 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. Furthermore, we were able to replicate their observation that β -VAE is able to learn unlabeled factors, like chair leg style.

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

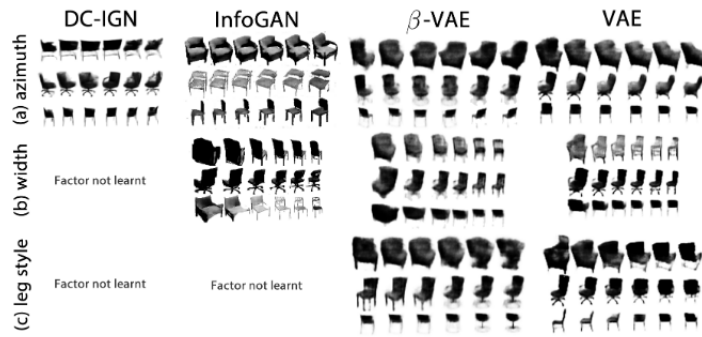


Figure 4: 3dChair results published by DeepMind in [1]

CelebA

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

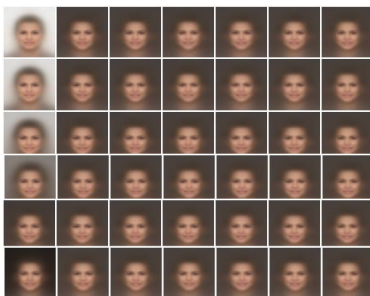


Figure 5: Poorly generated faces

We suspect that this occurred failure occurred 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 recommend adding an additional hyperparameter to regulate the loss further. Additionally, other implementations of β -VAE we found online were able to successfully generate results comparable 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 certainly play a factor in the quality of the results.

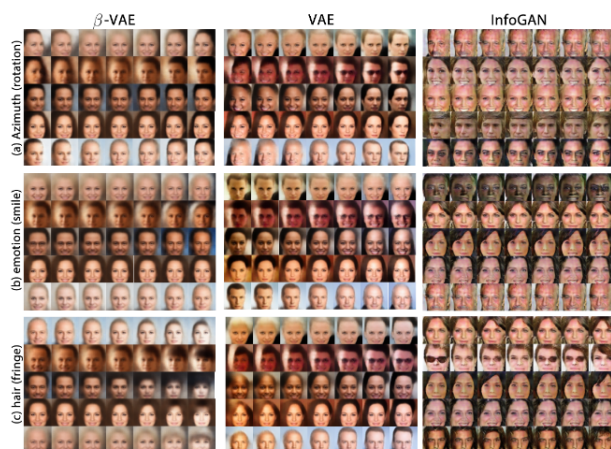


Figure 6: faces generated by DeepMind

Thus, following the papers recommendations 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].

```

1  """
2  model.py
3
4  contains network architecture for beta vae, as described in the appendix of [2]
5
6  """
7  import torch
8  import torch.nn as nn
9  from torch.autograd import Variable
10 import torch.nn.init as init
11
12
13 def reparam(mu, logvar):
14     """reparametization 'trick'
15
16     allows optimization through sampling process.
17     inputs: mean and variance
18     outputs: random var with perscribed mean and noisy variance terms
19
20     """
21
22     std = logvar.div(2).exp()
23     eps = Variable(std.data.new(std.size()).normal_())
24
25     return mu + std * eps
26
27
28 class View(nn.Module):
29     """View
30
31     acts like tf/np reshape
32
33     """
34
35     def __init__(self, size):
36         super(View, self).__init__()
37         self.size = size
38
39     def forward(self, tensor):
40         return tensor.view(self.size)
41

```

```

42
43 class betaVAE(nn.Module):
44     """betaVAE
45
46     class used to setup the betaVAE architecture
47
48     """
49
50     def __init__(self, z_dim=10, nchan=1):
51         super(betaVAE, self).__init__()
52         self.z_dim = z_dim
53
54         self.encoder = nn.Sequential(
55             nn.Conv2d(nchan, 32, 4, 2, 1),
56             nn.ReLU(True),
57             nn.Conv2d(32, 32, 4, 2, 1),
58             nn.ReLU(True),
59             nn.Conv2d(32, 32, 4, 2, 1),
60             nn.ReLU(True),
61             nn.Conv2d(32, 32, 4, 2, 1),
62             nn.ReLU(True),
63             View((-1, 32 * 4 * 4)),
64             nn.Linear(32 * 4 * 4, 256),
65             nn.ReLU(True),
66             nn.Linear(256, 256),
67             nn.ReLU(True),
68             nn.Linear(256, z_dim * 2),
69         )
70
71         self.decoder = nn.Sequential(
72             nn.Linear(z_dim, 256),
73             View((-1, 256, 1, 1)),
74             nn.ReLU(True),
75             nn.ConvTranspose2d(256, 64, 4),
76             nn.ReLU(True),
77             nn.ConvTranspose2d(64, 64, 4, 2, 1),
78             nn.ReLU(True),
79             nn.ConvTranspose2d(64, 32, 4, 2, 1),
80             nn.ReLU(True),
81             nn.ConvTranspose2d(32, 32, 4, 2, 1),
82             nn.ReLU(True),
83             nn.ConvTranspose2d(32, nchan, 4, 2, 1),
84         )
85

```

```

86     def forward(self, x):
87         """forward
88
89         propgates input through the network
90         inputs: sample input
91         output: reconstructed input, mu and var from the latent space
92
93         """
94         dist = self.encode(x)
95         mu = dist[:, : self.z_dim]
96         logvar = dist[:, self.z_dim :]
97         z = reparam(mu, logvar)
98         x_recon = self.decode(z)
99
100        return x_recon, mu, logvar
101
102    def encode(self, x):
103        return self.encoder(x)
104
105    def decode(self, z):
106        return self.decoder(z)
107
108
109 if __name__ == "__main__":
110     pass

```