CS3612 Project Report

Kaiwen Zhu 520030910178

I choose the tasks for enough computing resources.

1 Fashion-MNIST Clothing Classification

1.1 Model

The neural network designed by myself contains $4 + 4 \times$ `num_res_blocks` layers including $2 + 4 \times$ `num_res_blocks` convolutional layers and 2 fully connected layers. Here `num_res_blocks` is a hyper-parameter no less than 2.

In order to grasp the features of images, the main idea of my network is to learn *more* number of *relevant* features. To accomplish this, I use more convolution kernels to increase the number of feature maps, and max pooling to reduce the size of feature maps so that they are more compact and relevant.

I encapsulate such two functions into several layers called "ConvBlock". With parameters `in_channel`, `out_channel` and `s`, one ConvBlock takes a feature of shape (`in_channel`, `2s`, `2s`) as input and outputs a feature of shape (`out_channel`, `s`, `s`). Internally, a convolutional layer will be applied to the input to increase the number of channels (the number of feature maps) from `in_channel` to `out_channel`. Then a batch normalization layer is applied to avoid the vanishing gradient and exploding gradient problem. After a ReLU layer for activation, the feature will go through `num_res_blocks` ResBlocks (explained later) so that the features are extracted. At last, a maxpooling layer is applied to halve the size of feature maps, and a dropout layer is applied to avoid overfitting. All convolutional layers have kernel size of 3×3, padding of 1 and stride of 1 to preserve the size of features maps. The maxpooling layer has kernel size of 2×2, no padding and stride of 2. The dropout layer has the probability of 0.5. The architecture is summarized in Figure 1. The size of the feature map of each layer is indicated on the arrows between layers, and the text "same" means the size is the same as the last one.

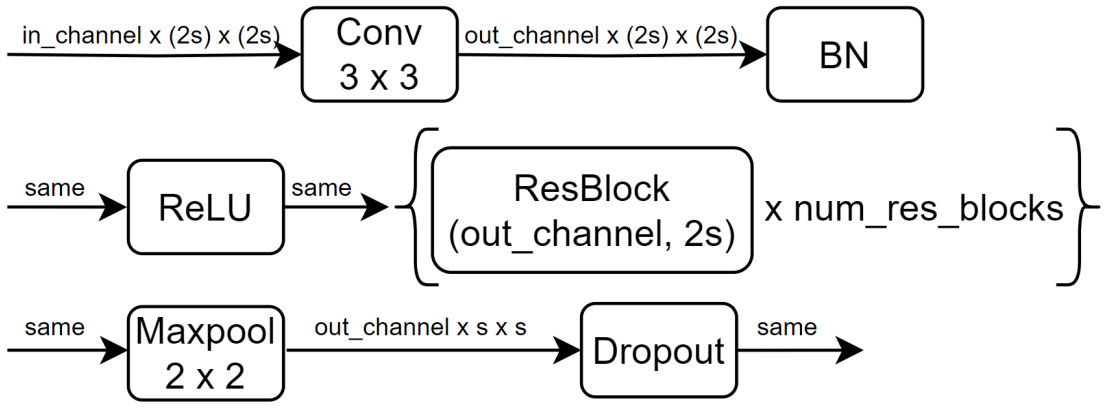


Figure 1: Architecture of a ConvBlock with parameter `in_channel`, `out_channel`,`s`

In ConvBlock, a module called "ResBlock" is used to extract features while not changing the size of inputs. It consists of two convolutional layers followed by batch normalization layers. Besides, it uses skip connection (adding the original input and the output of two layers of convolutional layers) in the hope of avoiding degradation in propagation, which is the key to the effectiveness of deep networks [1]. The architecture of ResBlock is shown in Figure 2.

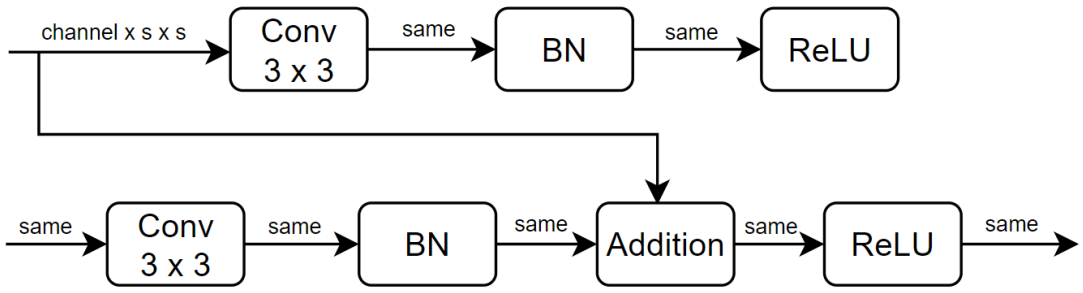


Figure 2: Architecture of a ResBlock with parameter `channel`, `s`

Considering the size of the input feature map is 32, it is reasonable to halve the size twice to obtain a moderate size of 8. Hence two ConvBlocks are needed. After flattening the output of two ConvBlocks, two fully connected layers are applied to reduce the dimension of the feature to 10, and at last a log softmax layer outputs the logarithm of the probability distribution over the 10 classes. This is the architecture of the neural network designed by myself, as is shown in Figure 3. Note there are four hyper-parameters to tune: the number of ResBlocks in a ConvBlock `num_res_blocks`, the number of channels of the output of the two ConvBlocks `num_channel_1` and `num_channel_2`, and the dimension of the output of the first fully connected layer `hidden_dim_fc`.

The loss function is chosen as the cross entropy of the output distribution (strictly, that's the exponent of the output) between the ground truth distribution (i.e., the distribution with probability 1 on the true label).

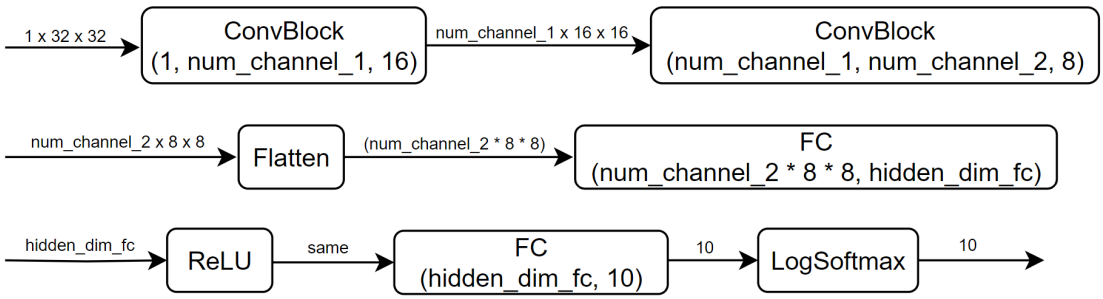


Figure 3: Architecture of my network

1.2 Result

I partitioned 20% of the 60,000 training images as the validation set. For some configuration, having trained the model for 100 epochs, I chose the model from the epoch with the lowest loss on the validation set and tested it on the testing set. After some trials, I found with the hyper-parameters in Table 1, the model from epoch 31 achieves satisfying performance, with a test accuracy of about 92.72% (as for hyper-parameters for training, the good batch size, learning rate and weight decay are 64, 0.001 and 0.0001 respectively).

Table 1: Hyper-parameters of the best model

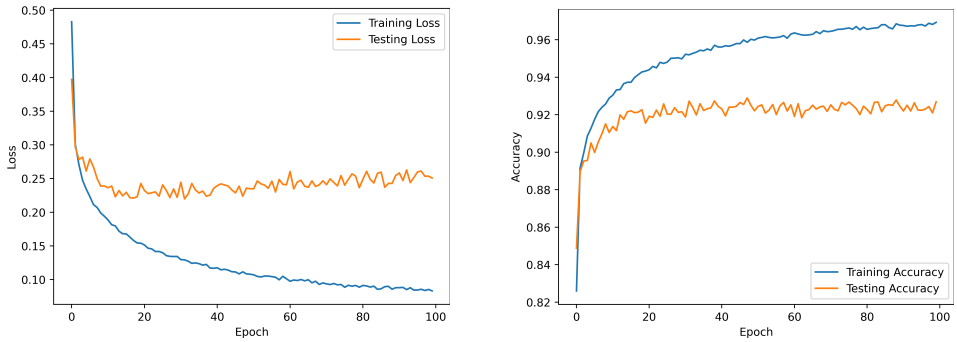
`num_res_blocks`	`num_channel_1`	`num_channel_2`	`hidden_dim_fc`
3	32	64	128

1.3 Visualization

Now we visualize the training process and features of the best model.

1.3.1 Loss and Accuracy

The loss and accuracy for the training set and testing set during the training process are plotted in Figure 4. It is clear that the loss/accuracy on the training set decreased/increased steadily and converged, indicating that the training was really effective; while the loss/accuracy on the testing set achieved their best much earlier (the loss even slightly increased later), and the curve is not so smooth, indicating the model failed to generalize to the testing set perfectly. This implies overfitting on the training set when the model was trained too much.



(a) Loss (b) Accuracy

Figure 4: Loss and accuracy on the training set and testing set during training

1.3.2 Intermediate Features

Consider the intermediate features of the network. I randomly selected 15 samples from each of the 10 classes and fed these 150 samples into the network. The output of the first convolutional layer, the first fully connected layer, and the final layer are visualized in Figure 5 using PCA (Principal Component Analysis) and t-SNE (t-Distributed Stochastic Neighbor Embedding). For my implementation of PCA and t-SNE see appendix A and appendix B respectively.

From Figure 5, we could find that as the samples propagate forward in the network, samples of the same class tend to cluster together. For example, features output by the first convolutional layer show almost no evidence of clustering, as is shown in 5b; while after the final layer, at least samples of class 1, 5, 7, 8, 9 are separated to some extent from other classes, as is shown in Figure 5f.

To better understand this, we quantify the quality of clustering. Good clustering should satisfy that samples of the same class are close to each other while samples of different classes are distant from each other. Calinski-Harabasz index is a metric considering these two factors [2]. It is the ratio of the between-cluster dispersion and within-cluster dispersion. Symbolically, suppose n_E samples are divided into k classes, c_E is the center of all samples, C_q is the set of the n_q samples in cluster q and c_q is the center of cluster q for $q = 1, 2, \cdots, k$, then the Calinski-Harabasz index is defined as

$$\frac{\mathrm{tr}(B_k)/(k-1)}{\mathrm{tr}(W_k)/(n_E-k)},$$

where

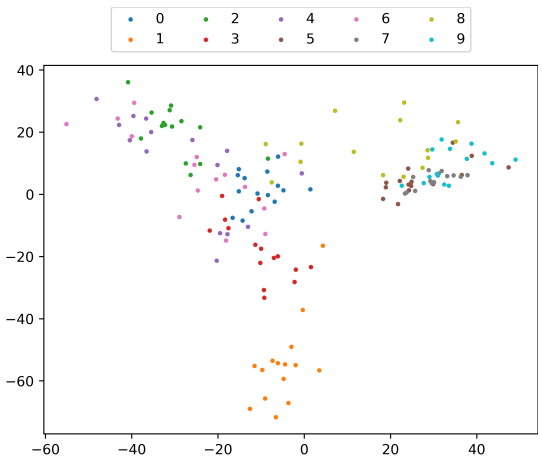
$$W_k = \sum_{q=1}^{k} \sum_{x \in C_q} (x - c_q)(x - c_q)^\top$$

and

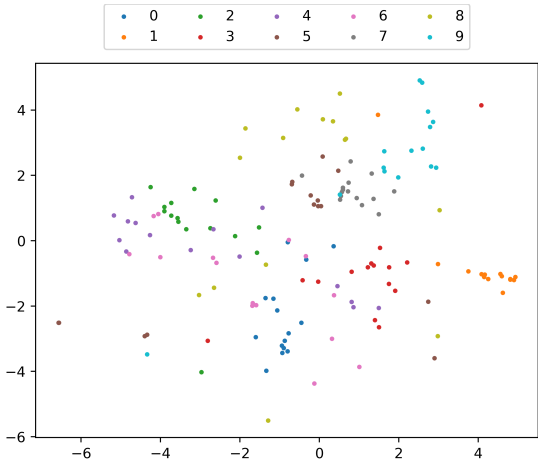
$$B_k = \sum_{q=1}^{k} n_q (c_q - c_E)(c_q - c_E)^\top.$$

The Calinski-Harabasz indices of the three sets of features are shown in Figure 6, where the labels on the horizontal axis from left to right correspond to features output by the first convolutional layer, the first fully connected layer, and the final layer, respectively. Figure 6 demonstrates that the quality of clustering increases a lot as samples propagate forward in the network.

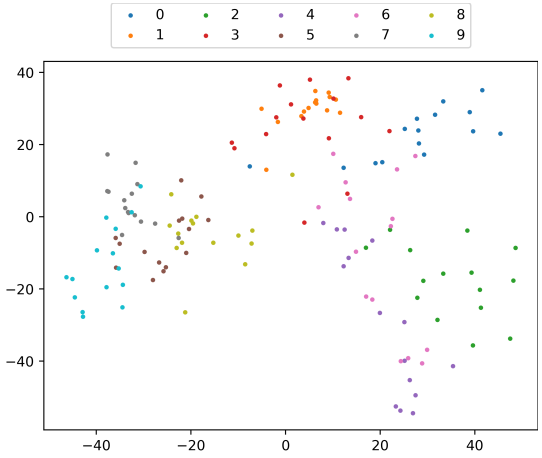
Besides, it is noteworthy that when performing PCA, the proportion of preserved variance increases as features propagate forward in the network. Figure 7 shows this tendency. This implies that during the forward propagation, the information of samples is prone to gather at fewer principle components, witnessing the effect of the network again.



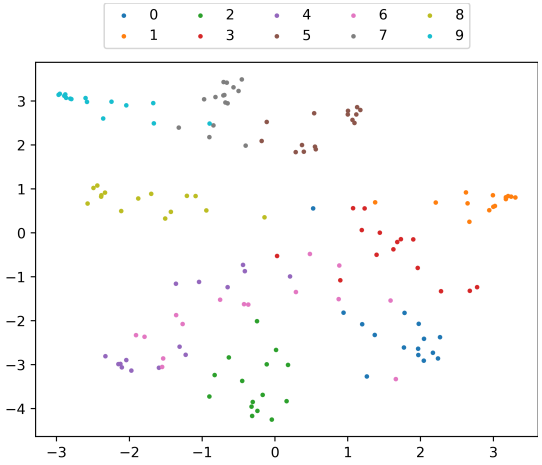
(a) Output of the first convolutional layer visualized by PCA



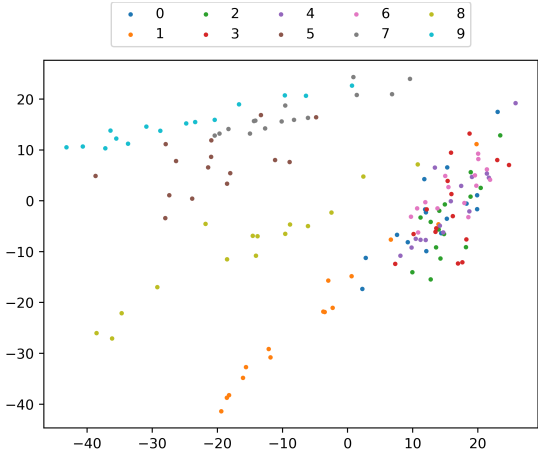
(b) Output of the first convolutional layer visualized by t-SNE



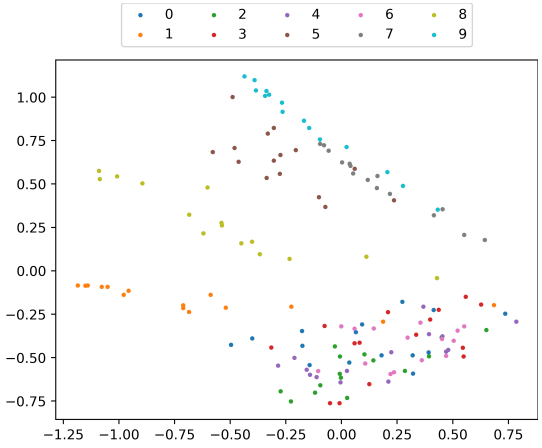
(c) Output of the first fully connected layer visualized by PCA



(d) Output of the first fully connected layer visualized by t-SNE



(e) Output of the final layer visualized by PCA



(f) Output of the final layer visualized by t-SNE

Figure 5: Visualization of intermediate features using PCA and t-SNE

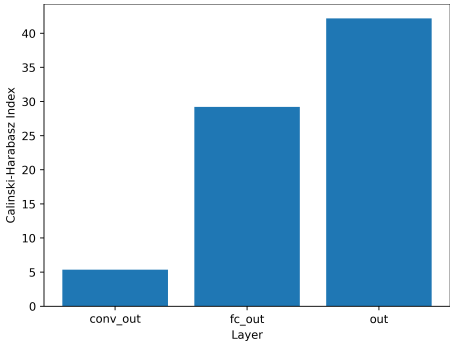


Figure 6: Calinski-Harabasz index of the 10 classes of features

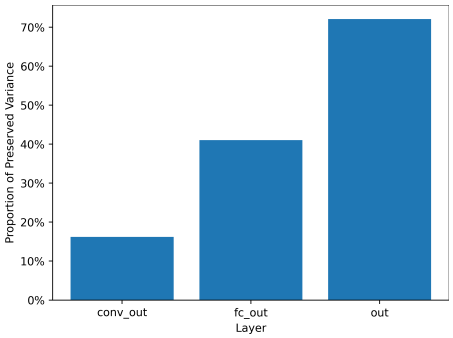


Figure 7: Proportion of preserved variance of PCA

2 Image Reconstruction

2.1 Model

VAE (Variational Auto-Encoder) consists of an encoder and a decoder. The encoder takes as input an image of shape (C, H, W), where C is the number of channels, H and W are height and width respectively, and outputs two vectors of `code_dim` dimensions `mean`, `log_var`. Then we sample a point $\mathbf{z}$ from $\mathcal{N}(\texttt{mean}, \exp(\texttt{log\_var}))$ as a code. The decoder takes $\mathbf{z}$ as input and outputs a feature of shape (C, H, W) as the generated image. This architecture is summarized in Figure 8.

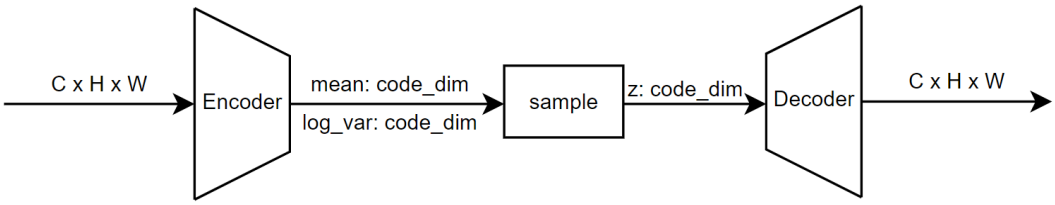


Figure 8: Overall architecture of my VAE

Encoder The encoder contains several convolutional layers. Let the number of channels of the input image be $c_0 = C$. For a sequence of numbers of channels $(c_1, c_2, \cdots, c_n)$, the i-th $(i = 1, 2, \cdots, n)$ convolutional layer takes as input a feature of shape $(c_{i-1}, H/2^{i-1}, W/2^{i-1})$ and outputs a feature of shape $(c_i, H/2^i, W/2^i)$. Each convolutional layer has kernel size of 3, padding of 1 and stride of 2 to halve the size of each feature map. Besides, there are a batch normalization layer and ReLU layer between each two adjacent convolutional layers. After those convolutional layers, the feature is flattened and fed into two fully connected layers respectively to obtain `mean` and `log_var`. This architecture of my encoder is shown in Figure 9.

Decoder The architecture of the decoder is roughly the reverse of that of the decoder. First, a fully connected layer converts the dimension of the input to that before the fully connected layers in the encoder, then the feature is reshaped to the shape before flattening in the encoder. After that, n layers of up-convolutional layers are applied to convert the shape of the feature to (c_0, H, W). Specifically, the i-th $(i = 1, 2, \cdots, n)$ up-convolutional layer takes as input a feature of shape $(c_{n+1-i}, H/2^{n+1-i}, W/2^{n+1-i})$ and outputs a feature of shape $(c_{n-i}, H/2^{n-i}, W/2^{n-i})$. In order for up-convolution to double the size of each feature map, we add zeros between elements in the feature map and use kernel size of 4, padding of 1 and stride of 1. Again, batch normalization layers and ReLU layers are added between convolutional layers. At last, a sigmoid layer outputs the normalized image. This architecture of my decoder is shown in Figure 10.

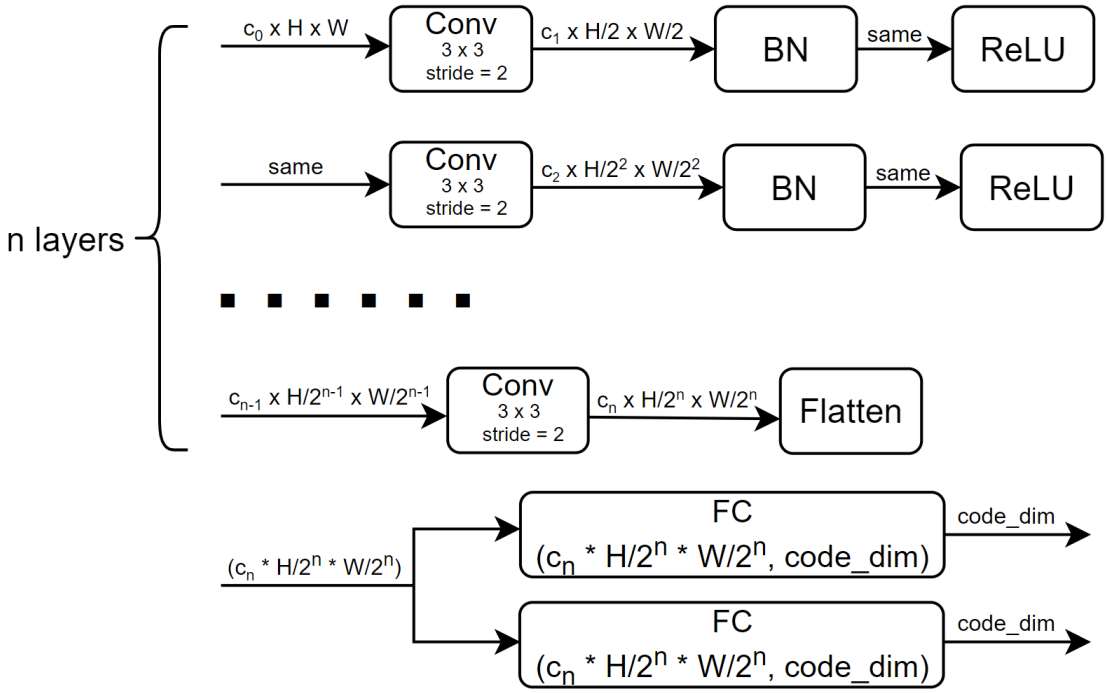


Figure 9: Architecture of my encoder

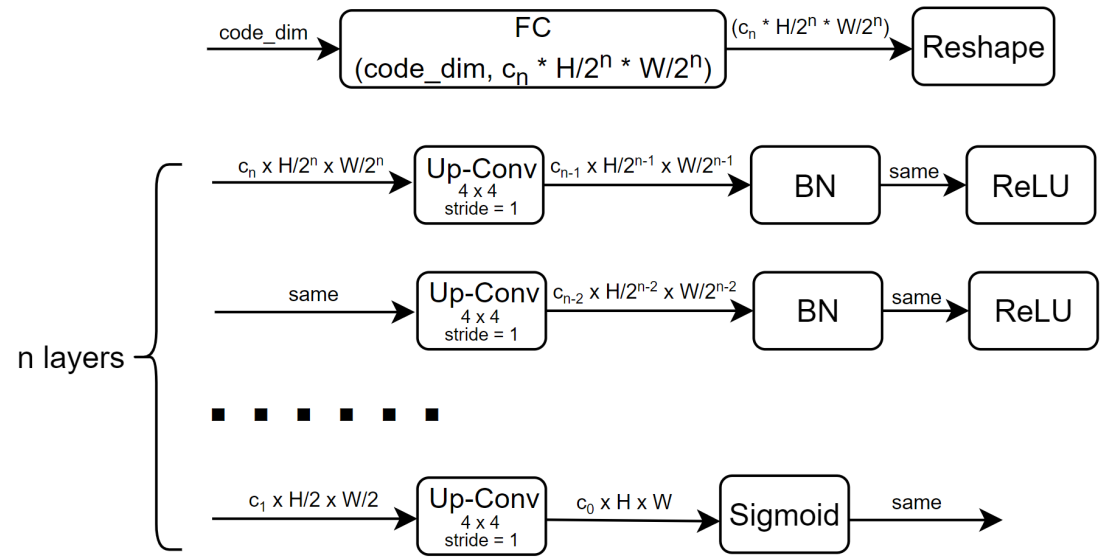


Figure 10: Architecture of my decoder

2.2 Implementation

Loss function Denote the distribution modeled by the decoder and encoder by $p_{\boldsymbol{\theta}}$ and $q_{\boldsymbol{\phi}}$ respectively. For an image $\mathbf{x}^{(i)}$, we hope to maximize its log likelihood $\log p_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})$. Considering the code (or latent variable) $\mathbf{z}$, the log likelihood could be rewritten as

$$\mathrm{KL}\left(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^{(i)}) \parallel p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}^{(i)})\right) + \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^{(i)})}\left[\log p_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}|\mathbf{z})\right] - \mathrm{KL}\left(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^{(i)}) \parallel p_{\boldsymbol{\theta}}(\mathbf{z})\right).$$

Because the first item is non-negative, the sum of the second and the third item is a lower bound of the log likelihood. Thereby we obtain the loss to be minimized [3]

$$\mathcal{L}\left(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}^{(i)}\right) = \mathrm{KL}\left(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^{(i)}) \parallel p_{\boldsymbol{\theta}}(\mathbf{z})\right) - \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^{(i)})}\left[\log p_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}|\mathbf{z})\right].$$

Assume the prior $p_{\boldsymbol{\theta}}(\mathbf{z}) = \mathcal{N}(0, \mathbf{I})$ and the posterior approximation $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^{(i)}) = \mathcal{N}(\boldsymbol{\mu}^{(i)}, \boldsymbol{\sigma}^{(i)}\mathbf{I})$, where $\boldsymbol{\mu}^{(i)}, \boldsymbol{\sigma}^{(i)} \in \mathbb{R}^J$ are calculated by the encoder fed with $\mathbf{x}^{(i)}$ (strictly, what the encoder outputs is $\log(\boldsymbol{\sigma}^{(i)})^2$). It follows that the first item of the loss is

$$\mathrm{KL}\left(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^{(i)}) \parallel p_{\boldsymbol{\theta}}(\mathbf{z})\right) = -\frac{1}{2}\sum_{j=1}^{J}\left[1 + \log\left(\sigma_j^{(i)}\right)^2 - \left(\mu_j^{(i)}\right)^2 - \left(\sigma_j^{(i)}\right)^2\right].$$

As for the second item, to obtain the expectation, we simply sample from $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}^{(i)}) = \mathcal{N}(\boldsymbol{\mu}^{(i)}, \boldsymbol{\sigma}^{(i)}\mathbf{I})$ and use the sampled $\mathbf{z}$ to compute $\log p_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}|\mathbf{z})$ in implementation. Again, we assume $p_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}|\mathbf{z})$ is a multivariate Gaussian with a diagonal covariance and a mean $g_{\boldsymbol{\theta}}(\mathbf{z})$ output by the decoder, then $\log p_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}|\mathbf{z}) \simeq \|\mathbf{x}^{(i)} - g_{\boldsymbol{\theta}}(\mathbf{z})\|^2$. Another point in implementation is that, in order to make computing $\mathbf{z}$ differentiable, we do not directly sample from $\mathcal{N}(\boldsymbol{\mu}^{(i)}, \boldsymbol{\sigma}^{(i)}\mathbf{I})$, but sample $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$ and compute $\mathbf{z} = \boldsymbol{\mu}^{(i)} + \boldsymbol{\sigma}^{(i)} \odot \boldsymbol{\epsilon}$, where $\odot$ signifies element-wise product.

Training Considering that the size of input images will be halved several times, I use interpolation to convert the size of training images to 256×256. I partitioned 20% of training images as the validation set and trained for 100 epochs, and the batch size, learning rate and weight decay are 64, 0.001 and 0.0001 respectively. The hyper-parameters of the VAE designed by myself are the dimension of code `code_dim` and numbers of channels `hidden_dims`. I found that with `code_dim`=256 and `hidden_dims` = (16, 32, 64, 64, 128), the model from epoch 45 achieves a low enough validation loss. Using these hyper-parameters, the training loss and validation loss during training are shown in Figure 11. We could see that the loss converged rapidly.

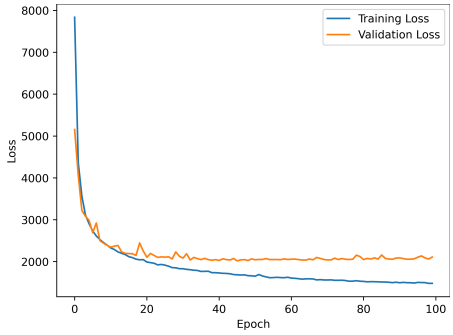


Figure 11: Loss on the training set and validation set during training

2.3 Reconstruction

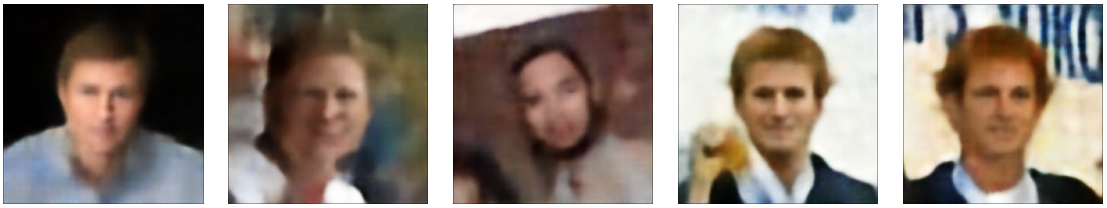
To measure the ability of the model to generate images, a basic way is to compare an image and the image generated from its code. That is, we hope to reconstruct true images. Select 5 images (shown in Figure 12a) from the training set, and the reconstruction results are shown in Figure 12b correspondingly. Although the reconstructed images are a little blurry, broadly they take on human shapes and are similar to the original images. This implies that the encoder and decoder collaborate well.

2.4 Image Generation by Interpolation

Now we try to use linear interpolation to generate images with specific properties. Choose two images and feed them into the encoder to get the codes $\mathbf{z}_1, \mathbf{z}_2$ of them. For some $\alpha \in (0, 1)$, interpolate them to obtain a new code $\mathbf{z} = \alpha\mathbf{z}_1 + (1-\alpha)\mathbf{z}_2$, and then use the decoder to generate a new image.

We experiment on the two images in Figure 13(white man and black man), the reconstruction of which is shown in Figure 14. Let α take $0.1, 0.2, \cdots, 0.9$, and we obtain the results in Figure 15. We could find that as the weight of the code of the black man $1-\alpha$ increases, the skin color of the men in the generated image becomes darker. This shows that our model does grasp the features of images and can use them to generate new images with desired properties. It is safe to come to the conclusion that the model has the capacity to compress the features into codes and generate images from codes of some features.

(a) Original images



(b) Reconstructed images

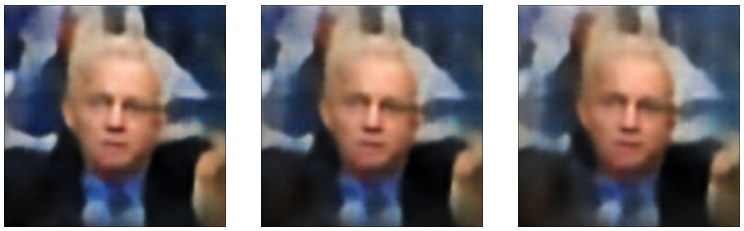
Figure 12: Original images and reconstructed images

(a) Image 1 (b) Image 2

Figure 13: The two images to be interpolated

(a) Image 1 (b) Image 2

Figure 14: Reconstruction of the two images to be interpolated



(a) $\alpha = 0.9$ (b) $\alpha = 0.8$ (c) $\alpha = 0.7$

(d) $\alpha = 0.6$ (e) $\alpha = 0.5$ (f) $\alpha = 0.4$

(g) $\alpha = 0.3$ (h) $\alpha = 0.2$ (i) $\alpha = 0.1$

Figure 15: Interpolation of the two images

A Implementation of PCA

Given a set of n samples with p features, my implementation of PCA is as follows:

1. Subtract the average over all n samples from each of the p features to get the normalized data $X \in \mathbb{R}^{n\times p}$.
2. Calculate the covariance matrix $\Sigma = \frac{1}{n-1}X^\top X$.
3. Calculate the eigenvalues and corresponding eigenvectors of Σ.
4. Select the two eigenvectors $w^{(1)}, w^{(2)} \in \mathbb{R}^p$ corresponding to the two largest eigenvalues.
5. Calculate $Y = XW \in \mathbb{R}^{n\times 2}$ where $W = [w^{(1)} \quad w^{(2)}] \in \mathbb{R}^{p\times 2}$. The rows of Y are the samples of reduced dimensions.

B Implementation of t-SNE

Given a set of n samples $x_i, i = 1, 2, \cdots, n$, my implementation of t-SNE is as follows [4]:

1. Compute
$$p_{j|i} = \begin{cases} \frac{\exp(-\|x_i - x_j\|^2/2\sigma_i^2)}{\sum_{k\neq i}\exp(-\|x_i - x_k\|^2/2\sigma_i^2)} & i \neq j \\ 0 & i = j \end{cases}$$
for all $1 \leq i, j \leq n$, where σ_i is set as 30 in my implementation.
2. Compute $p_{ij} = \frac{p_{i|j}+p_{j|i}}{2n}$ for all $1 \leq i, j \leq n$.
3. Initialize $y_i \in \mathbb{R}^2, i = 1, 2, \cdots, n$ randomly.
4. Compute
$$q_{ij} = \begin{cases} \frac{(1+\|y_i - y_j\|^2)^{-1}}{\sum_{k\neq l}(1+\|y_k - y_l\|^2)^{-1}} & i \neq j \\ 0 & i = j \end{cases} \tag{1}$$
for all $1 \leq i, j \leq n$.
5. Use gradient descent to optimize the loss
$$C = \mathrm{KL}(p \parallel q) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}}.$$
That is, subtract from y_i a learning rate multiplied by
$$\frac{\partial C}{\partial y_i} = 4\sum_j (p_{ij} - q_{ij})(1 + \|y_i - y_j\|^2)^{-1}(y_i - y_j)$$
for $i = 1, 2, \cdots, n$ iteratively until convergence. Note that q_{ij}'s should be recomputed by (1) in each iteration. Then $y_i, i = 1, 2, \cdots, n$ are the samples of reduced dimensions.

References

[1] A. Emin Orhan and Xaq Pitkow. Skip connections eliminate singularities, 2018.

[2] T. Caliński and J Harabasz. A dendrite method for cluster analysis. *Communications in Statistics*, 3(1):1–27, 1974.

[3] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[4] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.