Deep Convolutional Autoencoder Network for Breast Histopathology Image Clustering

(Category: Healthcare)

Bozhao Liu bozhao91@stanford.edu

Ju Zhang juzhang@stanford.edu

Abstract

Cancer subtyping enables oncologists to understand the heterogeneity within the same type of cancer, provide personalized treatment and improve the prognosis of cancer patients. To discover cancer subtypes using histopathology images, a common group of methods is unsupervised clustering, such as k-means clustering. However, the sole unsupervised learning methods often show inferior performance when applied on large-scale and high-dimensional image data due to the curse of dimensionality. While several dimension reduction method based on supervised deep learning has been proposed, including utilizing intermediate layers of a deep CNN model trained on labeled image data, these methods are subject to biases introduced from human labeling. In our study, we proposed and experimented on using deep autoencoder networks to extract image features, and feeding the dimension-reduced features to unsupervised clustering. We compared the performance of k-means clustering after combined with Deep Convolutional Autoencoder of different architecture, with two baselines of sole k-means clustering and supervised DenseNet classification, on a labeled breast histopathology image dataset. We demonstrated that our best model could achieve performance significantly higher than sole unsupervised clustering. Our proposed method is innovative as it is the first study that cross compare multiple autoencoder-based latent feature extraction methods on cancer histopathology images. Our work also demonstrates a high potential to help overcome the issues when subtyping cancer using high-dimensional images.

1. Introduction

In recent years, there is an increasing amount of research focusing on data-driven cancer diagnosis with machine learning methods and has achieved performance superior to humans. A less popular but equally important research field is identifying novel subtypes of cancers. The medical community has realized that cancer subtyping is a critical step towards personalized therapy and provides important biological insights into cancer heterogeneity. In the real-world clinical setting, cancer subtyping can assist doctors in providing precise treatment to patients, slowing cancer progression, and improving cancer prognosis.

Studies have shown that novel cancer subtype exploration can be done based on various types of data, such as multi-omics data, RNA sequence/micro-array data, clinical chemical data, and histopathological data. Among them, the histopathology (biopsy) images are one of the most common imagery for cancer screening and are usually collected for every patient at the early stage of the clinical journey, making it an ideal basis for subtype exploration.

A common way of determining cancer subtypes is by clustering individual cancer datum into clusters that shared underlining patterns and structures. Many studies have shown that both supervised and unsupervised machine learning can assist this clustering process. However, there are challenges for both types of methods. Unsupervised clustering such as k-means clustering often generates inconsistent results due to the high dimensionality of the data, whereas clustering based on supervised machine learning such as using features extracted from Deep Convolutional Neural Network (Deep CNN) can be heavily influenced by the biases in data. In this project, we aim to explore the applicability and evaluate the performance of a semi-supervise learning method that combines Convolutional Deep Autoencoder Network (CDAN) with an unsupervised method to cluster cancer histopathology images in comparison with the benchmarks of sole unsupervised learning and supervised learning methods. We believe CDAN can not only reduce the bias problem but also make the downstream unsupervised clustering result more consistent by dimensionality reduction [2].

Unsupervised learning algorithms are generally more common for cancer subtype exploration due to reasons such as limitations in computational resources, data types, and the availability of labels. However, it has been observed that while unsupervised clustering works reasonably well on medium-scale and low dimensional data, its performance degrades drastically on high-dimensional datasets with a massive number of samples [14]. Consequently, as one type of high-dimensional data, the histopathological images need to undergo dimension reduction before fed into unsupervised learning algorithms to achieve decent performance.

One of the methods of dimension reduction, when labels are available, is to train a Deep CNN model of classification, and extract the weights from one hidden layer as the input for unsupervised clustering [23]. However, the applicability of such a method is highly doubted among researchers regarding the potential biases of the extracted features since

the Deep CNN model is trained solely for classifying correctly.

Based on the above analysis, we conducted this study to solve the problem of the difficulty of high-dimensional histopathology image clustering. We combined autoencoder with unsupervised clustering to create a semi-supervised learning framework to derive novel subtypes of cancers. We reason that since autoencoders can non-linearly transform data into a latent space without looking at the labels, the dimension size can be reduced without introducing the biases resulted from supervised learning on prefixed labeling [14], which in turn improve the performance of the unsupervised learning algorithm. In our study, We successfully constructed multiple autoencoder models on a variety of architectures, including a fully-connected autoencoder, two Convolutional Autoencoders, an AlexNet Autoencoder, and a ResNet Autoencoder, then extracted the weights from the bottleneck layer as the dimension-reduced latent features, and finally apply unsupervised clustering on these features. We leveraged a dataset of labeled breast histopathology images to evaluate the performance of our method, by conducting a cross comparison of the impact of different AEs on clustering performance when in combination with the k-means clustering as well as a DenseNet based supervised classification model as the gold standard. We discovered that latent features extracted from Deep CNN autoencoders significantly improved the unsupervised F1 and accuracy scores of k-means clustering.

2. Related Work

In recent years, the deep autoencoder models have been shown the capability to extract latent features and reduce the dimension of images [8], and also achieved great accesses when used in clustering tasks[22] [24]. Since its inception, many studies have improved the autoencoder-based clustering method with different architectures and learning objectives and show the advantages of their work.

The autoencoder-based clustering can be viewed as two stages, i.e feature learning and clustering [26]. One strategy is to train the autoencoder first and extract the latent features from intermediate layers for the subsequent clustering algorithm. In [25], the authors implemented this method on MNIST dataset, first extract image features from the feature layer of a trained autoencoder, and then used that for the k-means clustering. [10] improve this framework by imposing a locality-persevering constraint on the learned representations to make the learned representations suitable for clustering.

Recently, more studies also try to adopt an alternative strategy and make the model perform the feature learning and clustering work jointly. For example, [11] added a self-expressive layer between the encoder and the decoder to allow the model to learn the affinity matrix and perform clustering directly. Both studies of [29] [7] used a deep convolutional embedded clustering algorithm that jointly learns the latent features representation with deep convolutional autoencoders and performs cluster assignment, by minimizing a loss function consists of both clustering loss and reconstruction loss. The authors of [27] proposed an improved architecture with an inception-like block with different types of convolution filters to preserve the local structure of convolution layers and enable clustering at different stages of feature extraction.

There are also an increasing amount of studies utilizing adversarial training and variational autoencoder in clustering tasks. Among them, [21] proposed ClusterGAN, which is an adversarial autoencoder framework to achieve clustering in latent space. by jointly training the GAN along with the inverse-mapping network with a clustering-specific loss. [5] proposed Dual Adversarial Auto-encoder, which simultaneously maximizes the likelihood function and mutual information between observed examples and a subset of latent variables, and showed its effectiveness in clustering tasks. [12] [17] proposed unsupervised generative clustering approaches which combine variational autoencoder model and Gaussian Mixture Model in data generative procedure to achieve clustering of images.

Some studies also proposed to create some ensembles of autoencoders to assist clustering. In [3], the authors combined three AE-based model convolutional autoencoder (CAE), adversarial autoencoder (AAE), and stacked autoencoder (SAE) to extract image features from MNIST and CIFAR-10 and achieve satisfying performance in the MNIST task. In [30], the authors adopt a collection of autoencoders or variational autoencoder where each autoencoder and the latent vectors from the autoencoders are concatenated to feed into a mixture assignment neural network to infers the distribution over clusters.

Besides the proposed improvement in the new architectures and framework, many studies have already implemented the deep learning model including autoencoders for feature extraction of medical images. In [23], the authors used supervised convolutional deep learning to learn about the image features during the classification task and used it for clustering. The authors of [18] used a Deep Wavelet Autoencoder-Based Deep Neural Network to extract the Bran MRI image features and use it for the subsequent DNN classification tasks. [20] propose Deep Clustering Convolutional Autoencoder model with a loss function composed of both clustering loss and reconstruction loss to cluster Cholangiocarcinoma images, and found that the derived clusters have significantly different hazard ratio by survival analysis. [1] integrated the adversarial training and autoencoder training to cluster patches from prostate cancer histopathology images. [15] proposed a convolutional autoencoder combined with k-means clustering to

classify duodenal biopsy images from subjects with Celiac Disease, Environmental Enteropathy, and healthy controls. In [4], the authors build a deep autoencoder model to learn compressed representations of scanpath images, and trained the clustering model to discovered clusters of eye-tracking scanpaths in Autism spectrum disorder. Recently [6] also proposed a framework to support unsupervised image features learning for lung nodule through lung CT image data, in which representations of the features are obtained from a Convolutional Autoencoder.

These studies lay out the groundwork for our research and also prove the innovativeness of our work, as no similar work has compared different autoencoder-based frameworks on breast cancer histopathology image clustering tasks.

3. Methods

3.1. Autoencoder-Clustering Framework

Our main proposed method to solve the histopathology image clustering problem is to combine a deep convolutional autoencoder with unsupervised clustering.

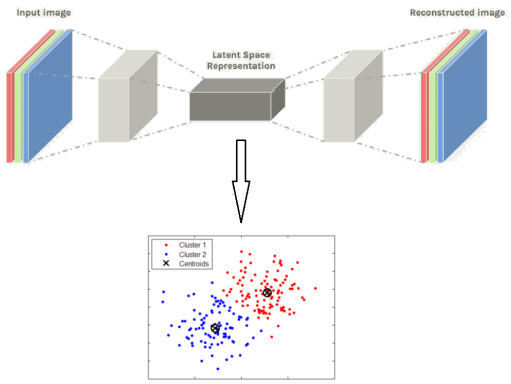

Figure 1: **General framework of our approach.** Diagram combined and adapted from [13] [15]

As Figure 1 shows, the autoencoder model is trained to reconstruct the original histopathology image. The bottleneck layer can be extracted as features of reduced dimension, which in turn being used for unsupervised clustering algorithms, such as k-means clustering

3.2. Fully-Connected and Convolutional Autoencoder

The deep convolutional autoencoder model takes images as inputs and tries to reconstruct the image when producing the output. The model is trained to minimize reconstruction loss such as square errors. We can represent the AE model as:

$$\phi : \chi \to F \tag{1}$$
$$\psi : F \to \chi \tag{2}$$
$$\phi, \psi = argmin_{\phi,\psi} ||X - (\phi \circ \psi)X'||^2 \tag{3}$$

In which ϕ denoted The encoder function that maps the original data X to a latent space F, which is present at the bottleneck. For fully-connected autoencoder, the feature mapping $\phi : \chi \to F$ can be represented as

$$h = f(x * W + b) \tag{4}$$

where W are weight matrix, b is the bias, and f is a non-linear activation function

For convolutional autoencoder, it is

$$h^j = f(x * W^j + b^j) \tag{5}$$

where W^j are weights of filters, b is the corresponding bias of the j-th feature map, and f is an activation function.

The decoder function ψ maps the latent space F at the bottleneck to the output. For fully-connected autoencoder, formula (4) still holds for $\psi : F \to \chi$. For convolutional autoencoder, to reconstruct the representative feature maps back to the original shape of the input image, a transposed convolutional operation can be used, which can be represented as

$$\hat{\chi} = f(\sum_{j \in H} h^j * \widetilde{W}^j + c) \tag{6}$$

$\widetilde{W}^j$ is the flip operator that transposes the weights, c is the corresponding bias, f is an activation function, and H indicates the group of feature maps.

As formula (3) indicated, we are trying to learn neural network-based functions to minimize the L2 loss between X and X', which are input and reconstructed output.

3.2.1 ResNet Convolutional Autoencoder

We implemented the ResNet autoencoder that adds residual connections to increase network capacity and prevent learned embedded-representations from deteriorating due to vanishing gradient. The diagram of a ResNet autoencoder is shown in 2.

Same as Deep Convolutional Autoencoder, ResNet Convolutional Autoencoder also consist of one encoder component and one decoder component. For the encoder, each hidden layer is composed by a non-linear mapping and a residual connection. The hidden representation in a hidden layer is compute follows:

$$h^{(l+1)} = r(h^{(l)}) + f(h^{(l)}) \tag{7}$$

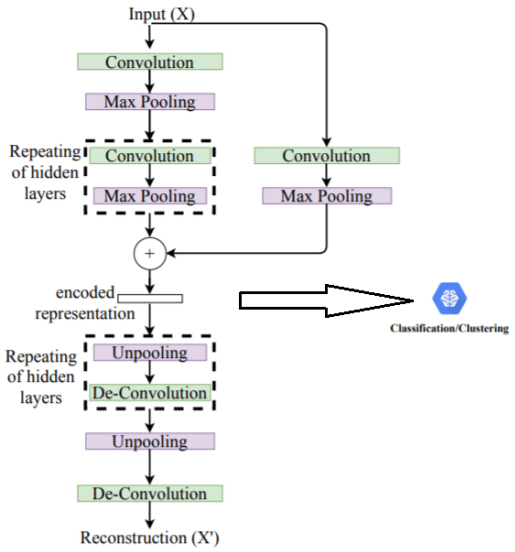

Figure 2: **ResNet Autoencoder architecture** [28]

Where the residual connection $r(h)$ is convolution operations and max pooling mapping that ensures the dimensions match the output of the function $f(h)$, which is also repeated layers of convolution and max pooling. The decoder component does not have residual connections. The training process is the same as Deep Convolutional Autoencoder, in which the reconstruction loss is being minimized. The encoded representation can later be extracted for the downstream unsupervised clustering task.

3.2.2 AlexNet Autoencoder

AlexNet was first proposed in [16]. It contain eight layers in which the first five were convolutional layers (with or without maxpooling) following three fully connected layers.

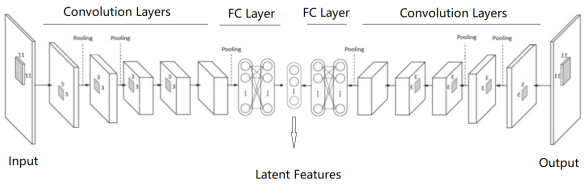

Figure 3: **Architecture of AlexNet-based Autoencoder**

It was originally proposed for the image classification task. In this study, We experimented and implemented an AlexNet-based autoencoder (Figure 3), in which the encoder and decoder each adopt two symmetrical AlexNets to allow the model to reconstruct the input and learn the latent features of the images.

3.3. k-means clustering

After the model is trained, we then extract the bottleneck layer as the latent features since the network has the most compressed dimension size here. As the dimensionality of the original data has been reduced, we can apply k-means clustering on the newly extract latent features as training data. To reduce the computational cost during clustering, We adopt an mini-match k-means clustering alogorithm:

1. Initialize k (k = 2 in this study) cluster centroids $u_1..u_k$ with x picked randomly from X;
2. Repeat until convergence:

$$x_1...x_m \longleftarrow \text{Pick m examples from } X \tag{8}$$

$$c^{(i)} := argmin_j ||x^{(i)}) - u_j||^2 \tag{9}$$

$$u_j := \frac{\sum_{i=1}^{m} 1\{c^{(i)} = j\} x^{(i)}}{\sum_{i=1}^{m} 1\{c^{(i)} = j\}} \tag{10}$$

Where m is minibatch size, $c^{(i)}$ is the cluster

3.4. Supervised DenseNet CNN

To better evaluate the performance of our semi-supervised autoencoder clustering approach, We implemented a DenseNet model to perform supervised learning on image classification. DenseNet is composed of many Dense Blocks. In one DenseNet block, direct connections from any layer to all subsequent layers were introduced. the layer receives the feature-maps of all preceding layers as:

$$h^{(l+1)} = r(h^{(0)}, h^{(1)}...h^{(l)}) + f(h^{(l)}) \tag{11}$$

Figure 4 shows the diagram architecture of the model. The input images are passed through multiple dense blocks to generate feature maps, and finally flattened and passed into fully connected layers for classification.

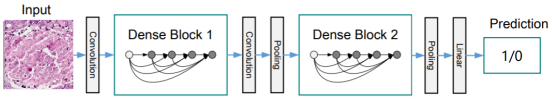

Figure 4: **Densely Connected Convolutional Neural Network for classification task**. Modified from [9]

Training of the DenseNet model is based on optimizing the softmax loss, which is calculated as the negative log likelihood of the softmax function

$$p(c_i|X) = \frac{\exp(z_i)}{\sum_{j=1}^{K} \exp(z_j)} \tag{12}$$

$$E = -\frac{1}{N} \sum_{n=1}^{N} log(p(c_i|X)) \tag{13}$$

Where $p(c_i|X)$ is the probability of class c,given input X, z is the score for ith class among total C classes.

3.5. Performance Evaluation

The performance of models are evaluated based on unsupervised F1 score and accuracy score. After clustering the images into groups of K, The labels for each cluster is inferred based on the labeling of the majority of data points in the cluster. The inferred labels can then be used in the downstream evaluation similar to supervised classification tasks.

The main evaluation metrics in this study are F1 score and accuracy, which are applicable for all methods proposed. The accuracy can be formulated as

$$ACC = \max_{m} \frac{\sum_{i=1}^{n} 1\{l_i = m(c_i)\}}{n} \tag{14}$$

where n denotes the number of data points, l_i denotes the ground-truth label, c_i is the cluster assignment produced by the clustering method, and m denotes all possible one-to-one mappings between clusters and labels [3].

The F1 score can be represented as

$$F1 = \frac{2 * precision * recall}{precision + recall} = \frac{TP}{TP + \frac{1}{2}(FP + FN)} \tag{15}$$

Where TP is the number of true positives, FP is the number of false positives and FN is the number of false negatives

4. Data

In this study, we conduct our experiments on the breast histopathology dataset which consists of 277,524 histopathology images of Invasive Ductal Carcinoma (IDC), the most common subtype of all breast cancers. This dataset is publically available on Kaggle.com [19]. This dataset has been generated from images of 162 whole mount slide images of Breast Cancer (BCa) specimens scanned at 40x. and the provider has conducted a certain level of preprocessing to generate a dataset of uniformly sized images. Each image is provided in *.png* format, with 50 pixels in width, 50 pixels in height, and 3 channels of RGB. Each pixel has a value between 0 and 255. Each image has been labeled Invasive Ductal Carcinoma (IDC) positive or negative, as shown in Figure 5.

We choose to use this labeled dataset as it provides the convenience of enabling us to evaluate the clustering algorithm performance with metrics usually associated with supervised learning such as F1 and accuracy. In addition, this enables us to compare the performance of our approaches to a supervised learning model, i.e. deep CNN classification model.

As we built and compared multiple models, in accordance with that, the breast histopathology data is processed differently. For instance, the input for the deep CNN model

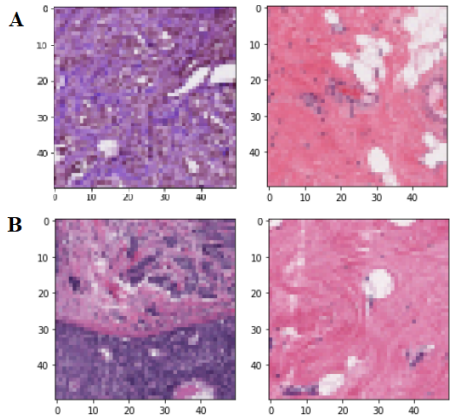

Figure 5: **Random data samples.** A: the upper panel shows images with positive labeling; B: the lower pannel shows cancer image with negative labeling

is normalized images with the same dimension as the original images. For k-means clustering, the images are reshaped into one-dimension vectors and then have the pixel value normalized to 0-1. For large autoencoder models, we resized the images to 250 pixels in width and 250 pixels in height before normalization. For small autoencoder models, we kept the images with 50 pixels in width 50 pixels in height and applied normalization directly.

We did not split the data into training and validation set in advance, since we chose to use cross-validation to generate the experimental results. As we used Adam optimizer in model training (see Experiment section), we chose a batch size of 300 for AlexNet Autoencoder, 160 for ResNet Autoencoder, and 1,000 for all other neural network models. For minibatch k-means clustering, we also used a batch size of 1,000.

5. Experiments

We conducted multiple experiments with different autoencoder architectures in addition to two baseline models of sole k-means clustering and supervised learning with DenseNet. We applied five-fold cross-validation to prove the consistency for all the deep learning models and take the mean of the results as our final result.

5.1. Experiment Setup

The experiments were done on a PC with an Intel I9 9900K (2.6GHz) with 8 cores CPU, 64GB memory, and one Nvidia RTX 3090 GPU. The operating system is Linux Ubuntu 20.04. All coding is performed in Python 3.8. We implemented our deep learning model in Pytorch 1.8.1 using the Compute Unified Device Architecture (CUDA) by Nvidia.

5.2. Model Implementation

5.2.1 Baseline

We first implemented the unsupervised k-means approach and supervised learning with DenseNet baseline approach to provide a general understanding of how well can the model perform. We reckon the performance using autoencoders should rest somewhere in between pure supervised and sole unsupervised approaches. For k-means clustering, we choose $K = 2$ which is equal to the number of labels of the dataset, and let the model run until convergence. For DenseNet, the model was built with five Dense blocks and finally connected with a fully-connected layer to perform classification. We training the model with Adam optimizer with a learning rate of $1e^{-4}$ for 30 epochs.

5.2.2 Fully-Connected Autoencoder

For autoencoders, we first experimented with a fully connected autoencoder to provide a comparison for the convolutional approaches.

We adopted 64 latent features and trained the network for 70 epochs with Adam optimizer and a learning rate of $1e^{-3}$. To find the final architecture, We completed an ablation study on the number of latent features. We discovered that increasing the latent feature layer size to more than 64 does not help improve the performance and a number lower than 64 reduced the performance. We also discovered that increasing the bandwidth of each layer does not help improve the performance but instead reduces the performance once the bandwidth exceeds a certain level. The final network architecture is shown in Appendix 8.1

5.2.3 Base Convolutional Autoencoder

We proceed to construct a base Convolutional Autoencoder with max-pooling layers. To reduce the dimension from 50 to 1, we choose an architecture with a five-layer encoder and a six-layer decoder. The network architecture is shown in Fig 9

We adopted latent features of size 256 and trained the network for 70 epochs with Adam optimizer and a learning rate of $1e^{-3}$. From the experiment, we found a further increase in performance compared to the Fully-Connected Autoencoder (see Result section) 5.2.2. After an ablation study on the filter sizes, we decided on the network architecture as in Appendix 8.1. We tested different choices for the sizes of the latest features and found out that 256 delivered the best performance. We also tried to increase the size of each channel of the convolutional layer but it does not result in improved performance.

5.2.4 Convolutional Autoencoder Without MaxPool

While tuning the decoder in Sec.5.2.3, we find completely reversing the encoder process by replacing the max pooling layer with upsize layer considerably affects the loss during the autoencoder training and reduced the performance of the clustering method on the extracted latent features. Therefore we deduced that the max pooling layer may influence the performance and conducted experiments on removing it from the autoencoder architecture. We constructed an alternative Convolutional Autoencoder which takes similar architecture as Sec.5.2.3 but with max pooling layers removed. The network architecture is shown in Appendix 8.1 and the model training is similar to the base Convolutional Autoencoder. We found further improvement in terms of accuracy and F1 score (see Result section).

5.2.5 AlexNet Based Autoencoder

Next, we experimented on whether adopting an AlexNet architecture in autoencoder for feature extraction may further improve the performance of the clustering. In our implementation, we used a pretrained Alexnet as the encoder and a reversed Alexnet with MaxPool layers removed as the decoder, and trained the model with Adam optimizer as well.

We found that the AlexNet Based Autoencoder did not produce expected performance improvement (see Result section). More concretely, the validation loss flattened early and the model in training quickly entered the phase of overfitting. We adopted the early stopping technique and conducted the hyperparameter searching for the optimal learning rate in the range between $1e^{-3}$ and $1e^{-4}$, but the model performance is still not satisfying. The network architecture is shown in Appendix 8.1.

5.2.6 Residual CNN Autoencoder

As the Alexnet-based Autoencoder 5.2.5 failed to further improve the performance of the clustering result, we hypothesized that adopting the residual connection between convolution blocks may potentially aid with the over-fitting problem. However, we observed a similar pattern of early flattening of validation loss, and using early stopping and lowering the learning rate from $1e^{-3}$ to $3e^{-5}$ contribute little to solving this issue and improving the clustering result. The network architecture is shown in Appendix 8.1.

6. Results

6.1. Visualization of Clustering Results

We performed t-distributed stochastic neighbor embedding (t-SNE), a technique popular for revealing local structure in high-dimensional data, to visualize the latent spaces from each autoencoder. The result is shown in 6. The class

separation in the visualization qualitatively agrees with the relative clustering performance. For example, both t-SNE visualizations of features from the ResNet AE and AlexNet AE have curly shapes, which may impact the performance of k-means clustering. In contrast, the t-SNE visualization of features from convolutional autoencoders looks more regular and easier to separate.

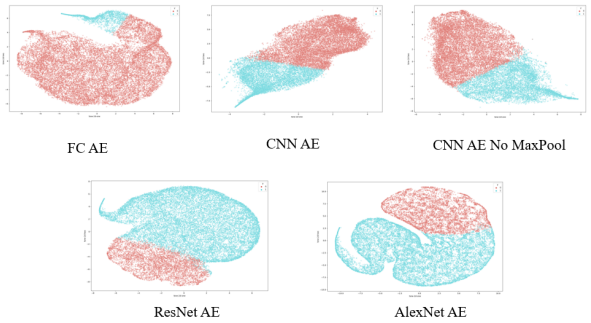

Figure 6: **t-SNE visualization of clustering results using different methods** . Different clusters are colored in red and blue respectively.

6.2. Clustering Performance

The clustering performance, represented by Accuracy and F1 score, have been summarized in Table 1

Models	F1	Accuracy
Sole k-means	0.593	0.714
FC AE + k-means	0.605	0.729
CNN AE + k-means	0.623	0.736
AlexNet AE + k-means	0.53	0.532
ResNet AE + k-means	0.515	0.50
CNN AE (No MaxPool)+k-means	**0.651**	**0.747**
Supervised with DenseNet	0.825	0.90

Table 1: **Performance of models on the classification of the Breast Histopathology data.** For the unsupervised methods, the F1 and Accuracy score were evaluated on inferred labels after clustering. Bold text indicates the best performance metric.

We can observe that using latent features extracted from the Fully-connected autoencoder, base CNN Autoencoder, and CNN Autoencoder without Max Pooling all improve the F1 and Accuracy metrics from sole k-means clustering. As we expected, the latent features extracted from the convolutional Autoencoder boost the k-means clustering performance better, since the convolutional layers enable the model to learn patterns in histopathology images on different levels. Interestingly, removing max pooling layers from CNN autoencoder boosts the k-means clustering. We hypothesized that, while max pooling operation benefits the model by introducing translation invariance and mitigating overfitting, too much max pooling non-linearity causes information loss when generating the latent features.

Our autoencoders built with AlexNet architecture and Residual blocks failed to improve the clustering result. A few reasons could cause this phenomenon. First, the network is too deep therefore can easily cause the latent feature extracted from the autoencoders to be biased towards the training data. Second, the deep structure of these two models introduces an excessive number of max pooling layers, thus causing a loss of information. Third, evidenced by our experiment in Sec.5.2.2, the bandwidth (i.e. number of neurons in the fully connected layer) being too wide in the fully connected layer may negatively impact the result.

7. Conclusion and Future Work

Our work demonstrated that by using latent features extracted from an appropriate autoencoder model, the performance of an unsupervised clustering algorithm on breast histopathology image data can be improved. Particularly, the convolutional autoencoder models can improve both the unsupervised accuracy and F1 score by a decent margin. We also discovered that in terms of the same network with the same depth, removing the pooling non-linearity has a positive effect on the latent feature extraction. In addition, we found that simply increasing the depth of the autoencoder or the bandwidth of the autoencoder does not necessarily help with improving the performance.

Due to the limitation of the course project scope, we have not tried all things of interest. To continue and improve from this study, in the future, the first thing we would like to further experiment on is to modify the AlexNet Autoencoder and ResNet Autoencoder with lower bandwidth. Next, we plan to experiment on building a Variational Autoencoder for feature extraction. Furthermore, we would like to validate our findings on a histopathology image dataset with more complexity, such as multiple class and variational image sizes.

7.1. Contribution and Acknowledgement

The work of this study is evenly distributed between Bozhao Liu and Ju Zhang. Bozhao has completed the work of building the fully-connected autoencoder, convolutional autoencoders, and AlexNet Autoencoder, Ju has completed the work of building k-means clustering, ResNet Autoencoder, and DenseNet supervised model. Our work is at https://github.com/Bozhao-Liu/Kaggle-breast-cancer-autoencoder. The writing work is also evenly distributed. We would like to acknowledge and give special thanks to the help from CS231N teaching team, including our project mentor Shao Lin, along the way.

References

[1] W. Bulten and G. Litjens. Unsupervised prostate cancer detection on he using convolutional adversarial autoencoders. *ArXiv*, abs/1804.07098, 2018. 2

[2] Yue Cao, Thomas Andrew Geddes, Jean Yee Hwa Yang, and Pengyi Yang. Ensemble deep learning in bioinformatics. *Nature Machine Intelligence*, 2(9):500–508, 2020. 1

[3] P. Chen and J. Huang. A hybrid autoencoder network for unsupervised image clustering. *Algorithms*, 12:122, 2019. 2, 5

[4] Mahmoud Elbattah, R. Carette, Gilles Dequen, Jean-Luc Guérin, and F. Cilia. Learning clusters in autism spectrum disorder: Image-based clustering of eye-tracking scanpaths with deep autoencoder. *2019 41st Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 1417–1420, 2019. 3

[5] Pengfei Ge, Chuan-Xian Ren, Jiashi Feng, and S. Yan. Dual adversarial auto-encoders for clustering. *ArXiv*, abs/2008.10038, 2020. 2

[6] Soumya Suvra Ghosal, Indranil Sarkar, and Issmail Elhallaoui. Lung nodule classification using convolutional autoencoder and clustering augmented learning method(calm). In *HSDM@WSDM*, 2020. 3

[7] Xifeng Guo, Xinwang Liu, En Zhu, and Jianping Yin. Deep clustering with convolutional autoencoders. In *ICONIP*, 2017. 2

[8] Geoffrey E. Hinton and R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313:504 – 507, 2006. 2

[9] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks, 2018. 4

[10] Peihao Huang, Yan Huang, Wei Wang, and Liang Wang. Deep embedding network for clustering. In *2014 22nd International Conference on Pattern Recognition*, pages 1532–1537, 2014. 2

[11] Pan Ji, T. Zhang, Hongdong Li, M. Salzmann, and I. Reid. Deep subspace clustering networks. *ArXiv*, abs/1709.02508, 2017. 2

[12] Zhuxi Jiang, Y. Zheng, Huachun Tan, Bangsheng Tang, and Hanning Zhou. Variational deep embedding: An unsupervised and generative approach to clustering. In *IJCAI*, 2017. 2

[13] Shin-Dong Kang. Graph convolutional neural networks. https://fr.slideshare.net/uspace/graph-convolutional-neural-networks/4. 3

[14] Md Rezaul Karim, Oya Beyan, Achille Zappa, Ivan G Costa, Dietrich Rebholz-Schuhmann, Michael Cochez, and Stefan Decker. Deep learning-based clustering approaches for bioinformatics. *Briefings in Bioinformatics*, 22(1):393–415, 02 2020. 1, 2

[15] Kamran Kowsari, R. Sali, M. Khan, W. Adorno, S. A. Ali, S. Moore, B. Amadi, Paul Kelly, Sana Syed, and D. Brown. Diagnosis of celiac disease and environmental enteropathy on biopsy images using color balancing on convolutional neural networks. *ArXiv*, abs/1904.05773, 2019. 2, 3

[16] A. Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60:84 – 90, 2012. 4

[17] Kart-Leong Lim, Xudong Jiang, and Chenyu Yi. Deep clustering with variational autoencoder. *IEEE Signal Processing Letters*, 27:231–235, 2020. 2

[18] Pradeep Kumar Mallick, S. Ryu, S. Satapathy, Shruti Mishra, G. Nguyen, and Prayag Tiwari. Brain mri image classification for cancer detection using deep wavelet autoencoder-based deep neural network. *IEEE Access*, 7:46278–46287, 2019. 2

[19] Paul Mooney. Breast histopathology images. https://www.kaggle.com/paultimothymooney/breast-histopathology-images. Accessed: 2021-04. 5

[20] Hassan Muhammad, C. Sigel, G. Campanella, T. Börner, Linda M. Pak, S. Büttner, J. Ijzermans, B. Koerkamp, M. Doukas, W. Jarnagin, A. Simpson, and Thomas J. Fuchs. Unsupervised subtyping of cholangiocarcinoma using a deep clustering convolutional autoencoder. In *MICCAI*, 2019. 2

[21] S. Mukherjee, Himanshu Asnani, Eugene Lin, and S. Kannan. Clustergan : Latent space clustering in generative adversarial networks. *ArXiv*, abs/1809.03627, 2019. 2

[22] Xi Peng, Shijie Xiao, Jiashi Feng, W. Yau, and Zhang Yi. Deep subspace clustering with sparsity prior. In *IJCAI*, 2016. 2

[23] Matthias Perkonigg, Daniel Sobotka, A. Ba-Ssalamah, and G. Langs. Unsupervised deep clustering for predictive texture pattern discovery in medical images. *ArXiv*, abs/2002.03721, 2020. 1, 2

[24] Yazhou Ren, N. Wang, Mingxia Li, and Zenglin Xu. Deep density-based image clustering. *Knowl. Based Syst.*, 197:105841, 2020. 2

[25] Chunfeng Song, Y. Huang, Feng Liu, Z. Wang, and Liang Wang. Deep auto-encoder based clustering. *Intell. Data Anal.*, 18:S65–S76, 2014. 2

[26] Fei Tian, Bin Gao, Qing Cui, E. Chen, and T. Liu. Learning deep representations for graph clustering. In *AAAI*, 2014. 2

[27] Q. Wang, Jiaqing Xu, R. Li, P. Qiao, K. Yang, S. Li, and Y. Dou. Deep image clustering using convolutional autoencoder embedding with inception-like block. *2018 25th IEEE International Conference on Image Processing (ICIP)*, pages 2356–2360, 2018. 2

[28] Chathurika S. Wickramasinghe, Daniel L. Marino, and M. Manic. Resnet autoencoders for unsupervised feature learning from high-dimensional data: Deep models resistant to performance degradation. *IEEE Access*, 9:40511–40520, 2021. 4

[29] Junyuan Xie, Ross B. Girshick, and Ali Farhadi. Unsupervised deep embedding for clustering analysis. *ArXiv*, abs/1511.06335, 2016. 2

[30] Dejiao Zhang, Yifan Sun, Brian Eriksson, and Laura Balzano. Deep unsupervised clustering using mixture of autoencoders, 2017. 2

8. Appendices

8.1. network architectures

Fully Connected AutoEncoders

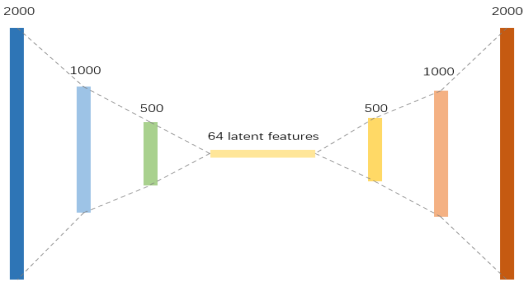

Figure 7: Fully Connect Autoencoder

(encoder): Sequential(
(0): Flatten(start_dim=1, end_dim=-1)
(1): Linear(in_features=7500, out_features=2000, bias=True)
(2): BatchNorm1d(2000, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(3): ReLU(inplace=True)
(4): Linear(in_features=2000, out_features=1000, bias=True)
(5): BatchNorm1d(1000, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(6): ReLU(inplace=True)
(7): Linear(in_features=1000, out_features=500, bias=True)
(8): BatchNorm1d(500, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(9): ReLU(inplace=True)
(10): Linear(in_features=500, out_features=64, bias=True)
(11): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(12): ReLU(inplace=True)
)
(decoder): Sequential(
(0): Linear(in_features=64, out_features=500, bias=True)
(1): BatchNorm1d(500, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(2): ReLU(inplace=True)
(3): Linear(in_features=500, out_features=1000, bias=True)
(4): BatchNorm1d(1000, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(5): ReLU(inplace=True)
(6): Linear(in_features=1000, out_features=2000, bias=True)
(7): BatchNorm1d(2000, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(8): ReLU(inplace=True)
(9): Linear(in_features=2000, out_features=7500, bias=True)
(10): Sigmoid()
)

Base CNN Autoencoders

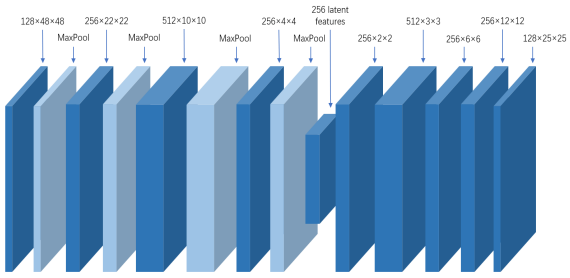

Figure 8: Arbitrary Convolutional Autoencoder

(encoder): Sequential(
(0): Conv2d(3, 128, kernel_size=(3, 3), stride=(1, 1))
(1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(2): ReLU(inplace=True)
(3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(4): Conv2d(128, 256, kernel_size=(5, 5), stride=(1, 1), padding=(1, 1))
(5): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(6): ReLU(inplace=True)
(7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(8): Conv2d(256, 512, kernel_size=(4, 4), stride=(1, 1), padding=(1, 1))
(9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(10): ReLU(inplace=True)
(11): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(12): Conv2d(512, 256, kernel_size=(3, 3), stride=(2, 2), padding=(2, 2))
(13): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(14): ReLU(inplace=True)
(15): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(16): Conv2d(256, 256, kernel_size=(2, 2), stride=(1, 1))
(17): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
)
(decoder): Sequential(

(0): ReLU(inplace=True)
(1): ConvTranspose2d(256, 256, kernel_size=(4, 4), stride=(1, 1), padding=(1, 1))
(2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(3): ReLU(inplace=True)
(4): ConvTranspose2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
(5): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(6): ReLU(inplace=True)
(7): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
(8): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(9): ReLU(inplace=True)
(10): ConvTranspose2d(256, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
(11): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(12): ReLU(inplace=True)
(13): ConvTranspose2d(256, 128, kernel_size=(5, 5), stride=(2, 2), padding=(1, 1))
(14): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(15): ReLU(inplace=True)
(16): ConvTranspose2d(128, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
(17): Sigmoid()
)

Base CNN Autoencoders (maxpool removed)

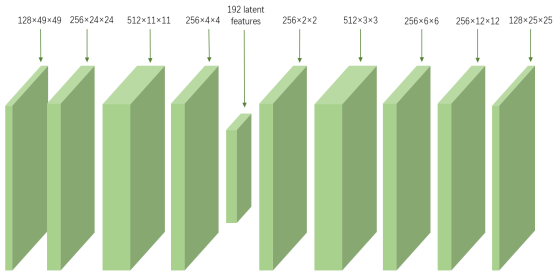

Figure 9: Convolutional Autoencoder(Maxpool Removed)

(encoder): Sequential(
(0): Conv2d(3, 128, kernel_size=(4, 4), stride=(1, 1), padding=(1, 1))
(1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(2): ReLU(inplace=True)
(3): Conv2d(128, 256, kernel_size=(5, 5), stride=(2, 2), padding=(1, 1))
(4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(5): ReLU(inplace=True)
(6): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2))
(7): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(8): ReLU(inplace=True)
(9): Conv2d(512, 256, kernel_size=(5, 5), stride=(2, 2))
(10): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(11): ReLU(inplace=True)
(12): Conv2d(256, 192, kernel_size=(4, 4), stride=(2, 2))
(13): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
)
(decoder): Sequential(
(0): ReLU(inplace=True)
(1): ConvTranspose2d(192, 256, kernel_size=(4, 4), stride=(1, 1), padding=(1, 1))
(2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(3): ReLU(inplace=True)
(4): ConvTranspose2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
(5): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(6): ReLU(inplace=True)
(7): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
(8): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(9): ReLU(inplace=True)
(10): ConvTranspose2d(256, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
(11): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(12): ReLU(inplace=True)
(13): ConvTranspose2d(256, 128, kernel_size=(5, 5), stride=(2, 2), padding=(1, 1))
(14): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(15): ReLU(inplace=True)
(16): ConvTranspose2d(128, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
(17): Sigmoid()
)

AlexNet Based Autoencoder

(encoder): AlexNet(
(features): Sequential(
(0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4),

padding=(2, 2))
(1): ReLU(inplace=True)
(2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
(3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
(4): ReLU(inplace=True)
(5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
(6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(7): ReLU(inplace=True)
(8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(9): ReLU(inplace=True)
(10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(11): ReLU(inplace=True)
(12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
)
(avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
(classifier): Sequential(
(0): Dropout(p=0.5, inplace=False)
(1): Linear(in_features=9216, out_features=4096, bias=True)
(2): ReLU(inplace=True)
(3): Dropout(p=0.5, inplace=False)
(4): Linear(in_features=4096, out_features=4096, bias=True)
(5): ReLU(inplace=True)
(6): Linear(in_features=4096, out_features=512, bias=True)
)
)
(decoder): RevertAlexNet(
(classifier): Sequential(
(0): ReLU(inplace=True)
(1): Dropout(p=0.5, inplace=False)
(2): Linear(in_features=512, out_features=4096, bias=True)
(3): ReLU(inplace=True)
(4): Dropout(p=0.5, inplace=False)
(5): Linear(in_features=4096, out_features=4096, bias=True)
(6): ReLU(inplace=True)
(7): Dropout(p=0.5, inplace=False)
(8): Linear(in_features=4096, out_features=9216, bias=True)
)
(features): Sequential(
(0): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(1): ReLU(inplace=True)
(2): ConvTranspose2d(256, 256, kernel_size=(5, 5), stride=(1, 1), padding=(1, 1))
(3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(4): ReLU(inplace=True)
(5): ConvTranspose2d(256, 384, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
(6): BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(7): ReLU(inplace=True)
(8): ConvTranspose2d(384, 192, kernel_size=(4, 4), stride=(2, 2), padding=(2, 2))
(9): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(10): ReLU(inplace=True)
(11): ConvTranspose2d(192, 192, kernel_size=(6, 6), stride=(2, 2), padding=(1, 1))
(12): BatchNorm2d(192, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(13): ReLU(inplace=True)
(14): ConvTranspose2d(192, 64, kernel_size=(6, 6), stride=(2, 2), padding=(1, 1))
(15): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(16): ReLU(inplace=True)
(17): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(2, 2))
(18): Sigmoid()
)

Residual CNN Autoencoder

(encoder): ResNet(
(features): Sequential(
(0): Conv2d(3, 64, kernel_size=(6, 6), stride=(2, 2), padding=(2, 2), bias=False)
(1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(2): ReLU(inplace=True)
(3): MaxPool2d(kernel_size=4, stride=2, padding=1, dilation=1, ceil_mode=False)
(4): ResBlock(
(Convpass): Sequential(
(0): Conv2d(64, 160, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
(1): BatchNorm2d(160, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(2): ReLU(inplace=True)
(3): Conv2d(160, 256, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
(4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(5): ReLU(inplace=True)
(6): Conv2d(256, 256, kernel_size=(4, 4), stride=(2, 2),

padding=(1, 1))
)
(identityPass): Conv2d(64, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
)
(5): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(6): ReLU(inplace=True)
(7): MaxPool2d(kernel_size=4, stride=2, padding=1, dilation=1, ceil_mode=False)
(8): ResBlock(
(Convpass): Sequential(
(0): Conv2d(256, 384, kernel_size=(7, 7), stride=(1, 1), padding=(3, 3))
(1): BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(2): ReLU(inplace=True)
(3): Conv2d(384, 512, kernel_size=(7, 7), stride=(1, 1), padding=(3, 3))
(4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(5): ReLU(inplace=True)
(6): Conv2d(512, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
)
(identityPass): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
)
(9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(10): ReLU(inplace=True)
(11): MaxPool2d(kernel_size=4, stride=2, padding=1, dilation=1, ceil_mode=False)
(12): Conv2d(512, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
(13): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(14): ReLU(inplace=True)
(15): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)
(classifier): Sequential(
(0): Dropout(p=0.5, inplace=False)
(1): Linear(in_features=512, out_features=256, bias=True)
(2): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(3): ReLU(inplace=True)
(4): Linear(in_features=256, out_features=256, bias=True)
)
)
(decoder): ReverseResNet(
(classifier): Sequential(
(0): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(1): ReLU(inplace=True)
(2): Linear(in_features=256, out_features=256, bias=True)
(3): ReLU(inplace=True)
(4): Dropout(p=0.5, inplace=False)
(5): Linear(in_features=256, out_features=4096, bias=True)
(6): BatchNorm1d(4096, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(7): ReLU(inplace=True)
(8): Dropout(p=0.5, inplace=False)
)
(features): Sequential(
(0): ConvTranspose2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
(1): ReLU(inplace=True)
(2): Dropout(p=0.5, inplace=False)
(3): ConvTranspose2d(512, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
(4): ReLU(inplace=True)
(5): Dropout(p=0.5, inplace=False)
(6): ReverseResBlock(
(Convpass): Sequential(
(0): ConvTranspose2d(512, 384, kernel_size=(7, 7), stride=(1, 1), padding=(3, 3))
(1): BatchNorm2d(384, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(2): ReLU(inplace=True)
(3): ConvTranspose2d(384, 256, kernel_size=(7, 7), stride=(1, 1), padding=(3, 3))
(4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(5): ReLU(inplace=True)
(6): ConvTranspose2d(256, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
)
(identityPass): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
)
(7): ReLU(inplace=True)
(8): Dropout(p=0.5, inplace=False)
(9): ConvTranspose2d(256, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
(10): ReLU(inplace=True)
(11): Dropout(p=0.5, inplace=False)
(12): ReverseResBlock(
(Convpass): Sequential(
(0): ConvTranspose2d(256, 160, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
(1): BatchNorm2d(160, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(2): ReLU(inplace=True)
(3): ConvTranspose2d(160, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))

(4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(5): ReLU(inplace=True)
(6): ConvTranspose2d(64, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
)
(identityPass): ConvTranspose2d(256, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
)
(13): ReLU(inplace=True)
(14): Dropout(p=0.5, inplace=False)
(15): ConvTranspose2d(64, 3, kernel_size=(6, 6), stride=(2, 2), padding=(2, 2))
(16): Sigmoid()
)