# Image Search Using Embeddings Learned from CNNs

Michael Aldridge
maldridge31@gatech.edu

Yufeng Wang
ywang3886@gatech.edu

Sylvia S. Tran
stran42@gatech.edu

Madhuri Jamma
mjamma@gatech.edu

August 4, 2021

## Abstract

Content based image retrieval (CBIR) helps to find similar images in a large dataset, given a query image. The similarity between the images is computed, which is then used to rank them and return the top matching images. Instead of relying on labels to conduct a search, we seek to use image features learned from deep models to retrieve image embeddings. Subsequently, the embeddings are searched using nearest neighbor algorithms. State of the art approaches augment deep models originally designed for image classification to learn embeddings that can be used to retrieve similar images. We measured the performance of such models on CBIR, using image labels to compute precision and recall as evaluation metrics. Among these approaches included ResNeXt-WSL, a model proposed by Facebook AI Research, and DenseNet. Our goal is to build a simplified model to generate image embeddings that perform comparably well with these state of the art approaches. To compare to the transfer learning approaches, an Auto Encoder model, a simplified DenseNet model, and a ResNet variant were built and trained on the COCO 2014 dataset. Overall, our results demonstrate that effective CBIR can be accomplished using a variety of image embedding techniques that leverage deep learning, including models that are less complex than state of the art computer vision models.

## 1 Introduction

In the last decade, the proliferation of image content has exploded. Technological developments such as camera phones and social media platforms heavily contributed to the increase in the number of images being generated by individuals. According to New York Times, as of 2015, 75% of all photos were taken by a camera phone [11]. Those trends continue to this day, creating a demand for services that enable individuals to quickly find photos from a query.

This need has been addressed with recent advances in research and applications on content-based image retrieval (CBIR). Despite existing research and applications, continued work remains relevant given the proliferation of image data. In the realm of image retrieval, there are two primary approaches: (i) text-based, and (ii) content-based [17]. The text-based approach has limitations due to the amount of labels required for a set volume of images. Although there are several widely used labeled datasets that are publicly available, none of them contain an exhaustive set of labels that could possibly be encountered in real life. Additionally, labeling new data can be costly and error-prone, so using a method that does not require labels could be more scalable in practice. Lastly, the quality of label-based searches are limited by the quality of the labels themselves. These issues can be alleviated using CBIR.

CBIR relies on features extracted from images rather than labels. Extracted features can include image shape, color, texture, edges, or combinations thereof. To the extent an architecture for image-to-vector can be identified and inexpensively trained, it can be readily used on a personalized library of photos. Therefore, this work focuses on image feature based CBIR, and not text-based search. The COCO dataset labels used to evaluate our experiments will not be used in the image search query [15]

Our objective is to discover embedding technique(s) that are effective for CBIR, and investigate whether less complex architectures than those found can perform comparably well. Several existing models are optimized for tasks such as object detection that have been used for CBIR. When augmented for CBIR, the final classification layer is removed from the architecture, and the output of the reduced model is used as an image embedding. Drawing inspiration from these approaches, we train ResNet-18 and DenseNet-inspired architectures as classification models on the COCO dataset before removing the final fully connected layers, and allow this reduced model to produce image embeddings. Additionally, an Auto Encoder has been trained on the same dataset to investigate the effectiveness of a fully unsupervised approach.

### 1.1 Dataset

The COCO dataset [15] released by Microsoft, was used for this project. COCO (Common Objects in Context) is a large-scale object detection, segmentation, key-point detection, and captioning dataset. Specifically, the 2014 version of the dataset was used, which contains 164K images split into 82K training, 41K validation, and 41K test images. The images have varying shapes which is not desirable for convolution-based embedding approaches. Therefore, images were resized and padded to 224 x 224 x 3, and then normalized to have a mean of [0.485, 0.456, 0.406] and standard deviations of [0.229, 0.224, 0.225] for each of the respective color channels (red, green, blue).

The "people" class is over-represented in the training data with over $\tilde{4}$5K samples, while the remaining classes have less than 10K on average. To balance the training data, images from three super-categories were selected: Animal, Sports, and Food, which have 10 subcategories each. This reduced the training set to 43K images comprised of 16K animal, 16K sports, and 11K food images.

## 2 Approach

The images from the reduced dataset were preprocessed, then passed through deep learning models to generate image embeddings. These image embeddings were in turn passed as inputs to a nearest-neighbor algorithm to search for similar embeddings in the feature space. When image search is typically performed, the query image is first run through a model architecture to generate the embedding and this embedding is then compared to our embedding database using a similarity metric. The top $k$ images are then returned from nearest neighbors. However, for purposes of this project's scope, an abridged CBIR was constructed by selecting query images from within the embedding database, one from each of the selected super-categories.

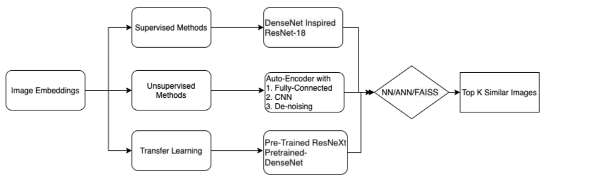


Image Embeddings
Supervised Methods
Unsupervised Methods
Transfer Learning
DenseNet Inspired ResNet-18
Auto-Encoder with 1. Fully-Connected 2. CNN 3. De-noising
Pre-Trained ResNeXt Pretrained-DenseNet
NN/ANN/FAISS
Top K Similar Images

Figure 1: Approach Overview: Supervised learning, Unsupervised learning and transfer learning for Image Embedding Retrieval.

Three types of embedding techniques were explored and compared, inspired by the following model architectures: (1) AutoEncoder, (2) DenseNet, and (3) Residual networks (i.e. ResNet and ResNeXt). In the case of DenseNet and residual networks, pre-trained models were compared to simpler home-grown models trained from scratch. For the residual networks, embeddings were generated using the Facebook AI ResNeXt101-32x8d architecture [9]. The home-grown residual network was a simpler ResNet-18 architecture and trained from scratch on the COCO 2014 training dataset [8].

### 2.1 AutoEncoder

AutoEncoders, as an unsupervised learning technique, are usually used on unlabeled data. They are also useful for dimensionality reduction. In this project, feature extraction, its third advantage, has been utilized.

Under the AutoEncoder family, three architectures were explored in order to obtain the hidden features. First, as a baseline, a dense fully-connected layer was used in both Encoder and Decoder, called VanillaAE. Second, the VanillaAE was improved to be more generalizable by including convolutions (ConvAE). Finally, inspired by [18], the denoising AutoEncoder (DenoisingAE), is implemented to enhance the model's ability to generalize on noisy images.

Encoders either compress or downsample the input images into smaller hidden feature vectors. For the VanillaAE, a fully-connected linear layer is used to flatten the final dimension of the images. For ConvAE, convolutional layers were applied to generate the feature map, followed by max pooling and activation functions like ReLu and Sigmoid. For DenoisingAE, corrupting the data on purpose by randomly turning some of the input values to zero helps when when there are more nodes in the hidden layer than there are inputs and the network is risking to learn the so-called "Identity Function" [21][2]

The purpose of Decoder is to reconstruct original images from the hidden features from the Encoder, and update the parameters through the pixel-level Mean Square Errors. The hidden features are upsampled to the size of the original inputs with torch.nn.ConvTranspose2d, which applies a 2D transposed convolution operator over an input image composed of several input planes.

Here are some implementation details: As the training objective is to reconstruct the original images, the MSE Loss is used to minimize the pixel-level difference. The training optimizer, Adam, helps the AutoEncoder to converge to the minima of the loss function with learning rate as 0.001 and batch size as 64 in the data loader. After 50 epochs, the curve of the loss function becomes flat, which means the model is fully trained from the images. One example with the training embeddings for image similarity can be found in 2

Figure 2: One Example from AutoEncoder's Image Similarity. Query image is the giraffe on the far left, subsequent images are most similar images retrieved, only one of which has a giraffe.

## 2.2 DenseNet

DenseNet is a dense convolutional neural network architecture that connects each layer in a feed-forward fashion. This architecture improves information flow by carrying information from previous layers to subsequent layers [12].

Because the DenseNet architecture is successful at retaining information from hidden layers, a DenseNet inspired (mini-DenseNet) network was built and used to generate embeddings for CBIR. The DenseNet bottleneck layers were primarily used, which consists of a sequence of three consecutive operations: batch normalization, a rectified linear unit, followed by a 3 x 3 convolutional layer. The mini-DenseNet is comprised of a 7 x 7 convolutional layer, batch normalization, rectified linear unit, 3 x 3 max pool layer, followed by two DenseNet bottleneck blocks. The key difference between the two architectures is DenseNet is a deeper architecture, and is comprised of four dense blocks, sandwiched by the initial feature extraction and a fully connected layer, whereas the mini-DenseNet is shallow, and does not contain dense blocks. Furthermore, parameters such as kernel size, stride, and padding for convolutional layers, and max pooling differ.

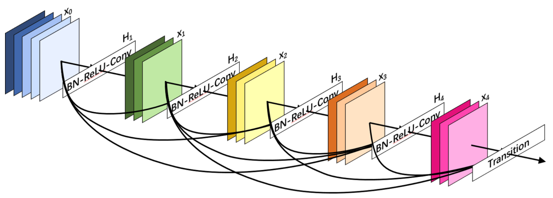


Figure 3: DenseNet Bottleneck Layers between four DenseNet blocks [12].

To compare results, embeddings were generated using both architectures. The mini-DenseNet was used to train a classifier over 25 epochs on the training and the validation data splits. The optimizer used was Adam to ensure a more stable training loss during training using a learning rate of 0.001, $\beta$ values of 0.9 and 0.999, $\epsilon$ of $1x10^{-5}$ and weight decay of 0.01 [14] [16]. To generate embeddings, the final fully-connected layers were excluded from both architectures, which resulted in embedding vectors of length 50,176, and 1,024 for DenseNet and the mini-DenseNet, respectively.

Both the pretrained DenseNet and mini-DenseNet architectures are computationally expensive. Both models were trained on an Amazon EC2 instance with 32 CPU cores and 2 GPUs. Additionally, the DenseNet and mini-Densenet architectures each had 25.6M and 102M parameters, respectively. It was difficult to arrive to a lighter architecture with fewer parameters that was more shallow while drawing from the integral building blocks of DenseNet. The increased number of parameters was largely attributed to the difference in kernel size and padding compared to DenseNet. Despite the increased number of parameters in the mini-DenseNet, the time taken to generate the embeddings on the shallow was far less than the DenseNet architecture because the embedding size was significantly smaller (1,024). During training, prior to removing the final fully-connected layer of the mini-Densenet, the model was over fit after epoch 11 (out of 25). Therefore, the pretrained mini-DenseNet used to generate embeddings was model state at epoch 11.

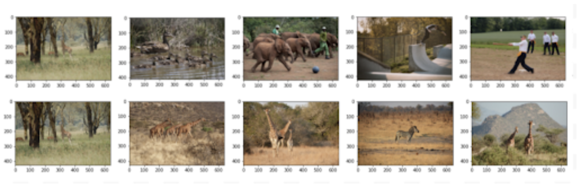


Figure 4: Top: mini-DenseNet retrieved images; Bottom: DenseNet retrieved images; Both: Far left column is the query image of a giraffe, subsequent four are the most similar images retrieved by ANNOY.

## 2.3 Residual Networks

Facebook AI's ResNeXt-WSL model was leveraged to generate image embeddings and perform image search, among other tasks, on the COCO 2017 validation set, inspired by the approach taken in [20][3]. ResNeXt is an evolution of ResNet that introduces the dimension of cardinality (the size of the set of transformations in the residual block). Blocks from both the ResNet and ResNeXt architecture with cardinality of 32 are compared in Figure 5. In the approach described in [20], ResNeXt-32x8d was used, which was pretrained on nearly 1Bn images from Instagram in an image classifier. ResNeXt-32x8d is the smallest of the models proposed in [9], with  85M trainable parameters. By introducing cardinality, researchers at Facebook AI found that they could achieve a superior top-1 classification error on the ImageNet-1K dataset with a ResNeXt model versus a ResNet model with similar complexity.

To generate the image embeddings, the last fully-connected layer that enables classification was removed. This generated an embedding space with 2,048 dimensions, which in turn was used to retrieve similar images. This technique was applied to this project's selected COCO 2014 validation set and used to benchmark performance.

To test whether a simpler model could perform on par with a complex model such as ResNeXt-32x8d, a ResNet-18 model as described in Deep Residual Learning for Image Recognition (Kaiming He, et. al.) was constructed and trained on the COCO 2014 dataset[8][7]. This model has trainable parameters totaling $\tilde{5}$M compared to ResNeXt's $\tilde{8}$5M, and also lacks the cardinality that is employed in ResNeXt. Similar to the approach in [20], the ResNet-18 model was first trained as a classifier. Model weights were initialized with uniform Xavier initialization, with Adam as the chosen optimizer (learning rate of 0.001, $\beta$ values of 0.9 and 0.999, $\epsilon$ of $1x10^{-5}$ and weight decay of 0.01). The loss function used to train the model was Cross Entropy Loss. [14] [16].

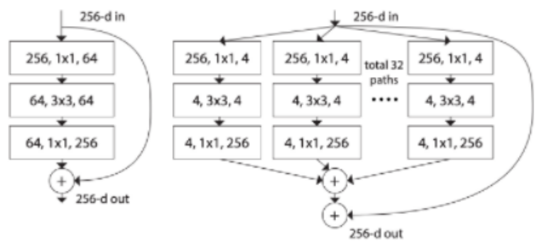


Figure 5: **Left**: ResNet Block. **Right**: A block of ResNeXt with cardinality=32 [9]

During training, challenges were encountered with the model weights converging to zero, resulting in trivial embeddings that were the same regardless of input image. The resolution to this issue was two-fold: (1) the weights were initialized using uniform Xavier initialization, (2) the learning rate was tuned to resolve the vanishing gradient. A smaller learning rate of 0.001 resulted in meaningful embeddings that were more adept for CBIR. Challenges related to compute and storage were addressed by executing the training and validation process in a Google Colab environment.

After training, the ResNet decoder, which mapped the learned features to classes, was removed. The reduced model produced embedding vectors of length 25,088. While the ResNet-18 model proved capable of retrieving some relevant images, its performance fell short of that exhibited by the pre-trained ResNeXt model. The performance metrics are detailed and discussed in Section 3.

Below is a comparison of the search results for a sample query image of a giraffe for both the ResNet-18 model (top) and the ResNeXt model (bottom). While the ResNet-18 embeddings did not return a giraffe, the top four results do bear resemblance to the query image. While this is only a sample image, these results suggest that the simpler ResNet-18 model is capable of returning similar images based on some latent image features learned during training.

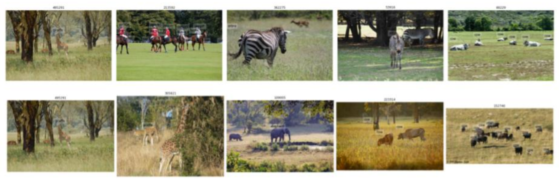


Figure 6: Top: ResNet-18 retrieved images; Bottom: ResNeXt retrieved images; Both: Far left column is the query image of a giraffe, subsequent four are the most similar images retrieved by KNN.

## 2.4 Image Retrieval

Nearest neighbors (NN) [6] algorithm is an unsupervised algorithm which works by calculating the distance of a query observation from all training observations to retrieve the top matches. Different distance metrics such as "cosine", "minkowski", "euclidean" can be specified as a tunable parameter. [5] Facebook AI Similarity Search (FAISS), is a python library [10] based on a NN implementation for billion-scale dataset that retrieves similar documents. During the FAISS search, a query vector is provided, and vectors with the nearest (Euclidean) distance to the query vector that are returned. Several techniques are utilized to make the retrieval faster such as exploiting multiple cores, machine SIMD vectorization, and BLAS libraries for efficient distance computations and retrieval [10]. Code tutorial used from [4].

Approximate Nearest Neighbors Oh Yeah (ANNOY), is a python library [1] that expeditiously executes NN queries in high dimensional spaces. ANNOY builds a forest of binary tree where each binary tree is created by recursively splitting the hyperplane equidistant between two random points. Points that are close to each other in the space are likely to be close to each other in the tree [1].

# 3 Experiments and Results

## 3.1 Performance Metrics

Success for this CBIR project was measured by calculating both precision and recall at $k$, where $k$ represents the number of top items returned from the search algorithms given a query image[**?** ]. While $k$ is arbitrarily set to 4, the team recognizes that because over 20K embeddings were generated for each of the respective techniques, going much deeper than $k = 4$ could be computationally expensive given the various embedding sizes.

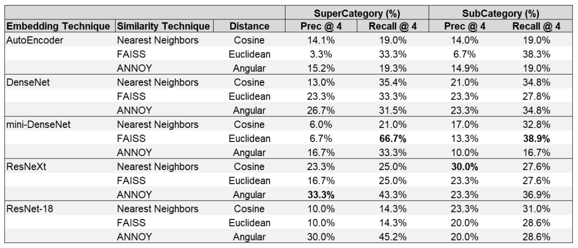
The formulas for precision and recall at k were derived by calculating the following:

$$\text{precision @ k} = \frac{\text{\# of relevant items @ k}}{\text{\# of recommended items}}$$
$$\text{recall @ k} = \frac{\text{\# of relevant items @ k}}{\text{total \# of relevant items}}$$

For each of the respective embedding approaches explored: (i) AutoEncoder, (ii) DenseNet, and (iii) Residual Learning, three search algorithms were used to retrieve similar images to a given query image. Each of the search algorithms have nearest-neighbors underpinnings: (i) Nearest Neighbors, (ii) FAISS, and (iii) ANNOY [19] [13] [1]. In each method, the ten nearest neighbors were considered to be the scope of the search result (i.e., number of recommended items). In the case of recall, total number of relevant items is only considered from the search result. Therefore, it should be noted that recall can at maximum reach 40% with these definitions. Despite the nearest-neighbors underpinnings, each of the respective search algorithms rely on different distance metrics: (i) cosine, (ii) Euclidean, and (iii) angular, respectively. By testing the home-grown embeddings against pretrained embeddings across multiple search algorithms, and benchmarking the ResNet-18 vs. ResNeXt, and mini-DenseNet vs. DenseNet, we can by proxy measure the effectiveness of each of the embedding approaches in mapping the input images to representative feature spaces through precision and recall metrics.

The results in the table below compare the precision and recall metrics at $k = 4$ and $n_neighbors = 10$ for the transfer learning approaches (DenseNet and ResNeXt) and their trained-from-scratch counterparts (mini-DenseNet and ResNet-18), along with the AutoEncoder.

Between both metrics, recall @ $k$ is considered to be the superior metric over precision @ $k$ because the denominator in the recall @ $k$ metric is comprised of those retrieved images whose category matches the query image category. This is a truer measure of both the embeddings' and search algorithms' ability to successfully create a representative feature map and select relevant images, respec-



| | | | SuperCategory (%) | | SubCategory (%) | |
|---|---|---|---|---|---|---|
| **Embedding Technique** | **Similarity Technique** | **Distance** | **Prec @ 4** | **Recall @ 4** | **Prec @ 4** | **Recall @ 4** |
| AutoEncoder | Nearest Neighbors | Cosine | 14.1% | 19.0% | 14.0% | 19.0% |
| | FAISS | Euclidean | 3.3% | 33.3% | 6.7% | 38.3% |
| | ANNOY | Angular | 15.2% | 19.3% | 14.9% | 19.0% |
| DenseNet | Nearest Neighbors | Cosine | 13.0% | 35.4% | 21.0% | 34.8% |
| | FAISS | Euclidean | 23.3% | 33.3% | 23.3% | 27.8% |
| | ANNOY | Angular | 26.7% | 31.5% | 23.3% | 34.8% |
| mini-DenseNet | Nearest Neighbors | Cosine | 6.0% | 21.0% | 17.0% | 32.8% |
| | FAISS | Euclidean | 6.7% | **66.7%** | 13.3% | **38.9%** |
| | ANNOY | Angular | 16.7% | 33.3% | 10.0% | 16.7% |
| ResNeXt | Nearest Neighbors | Cosine | 23.3% | 25.0% | **30.0%** | 27.6% |
| | FAISS | Euclidean | 16.7% | 25.0% | 23.3% | 27.6% |
| | ANNOY | Angular | **33.3%** | 43.3% | 23.3% | 36.9% |
| ResNet-18 | Nearest Neighbors | Cosine | 10.0% | 14.3% | 23.3% | 31.0% |
| | FAISS | Euclidean | 10.0% | 14.3% | 20.0% | 28.6% |
| | ANNOY | Angular | 30.0% | 45.2% | 20.0% | 28.6% |

Figure 7: Precision  Recall at 4 for each of the embedding techniques and search algorithms applied.

tively.

The mini-DenseNet embeddings are weaker than DenseNet at capturing representative features in the input images, evidenced by the consistently lower precision and recall metrics at $k = 4$. The only exception is in the case when the embeddings from both models are passed through the FAISS search algorithm. Here, mini-DenseNet precision and recall @ 4 exceed those of DenseNet, likely more so attributed to the search algorithm than the underlying embeddings. The penultimate layer of the DenseNet architecture is a convolutional layer, whereas the penultimate layer of the mini-DenseNet was a fully-connected linear layer. The DenseNet flattened convolutional layer resulted in better images retrieved than the mini-DenseNet.

In the case of the residual models, the transfer learning approach taken with ResNeXt generally seems to outperform that of the ResNet-18 model. That said, when using ANNOY as the search algorithm, ResNet-18 is not a far cry from ResNeXt from a quantitative metric perspective. Additionally, the query results compared between the two models in Section 2.3 appeared to generate reasonable results. While the pre-trained ResNeXt ($\tilde{8}$5M trainable parameters) significantly outperformed ResNet-18 ($\tilde{5}$M trainable parameters) for the Nearest Neighbors and FAISS search methods, the results seem to suggest that ResNet could perhaps perform on par with a far more complex model if the more time could be invested in hyperparameter tuning.

## 4 Conclusion

Unsupervised learning approaches can retrieve hidden features from images even without labeled data. AutoEncoder with its encoder-decoder architecture extracted features and optimized the training process through the mean square error on the original images and reconstructed images. With the supervised learning methods, the annotations and categorical information are utilized as training objectives, and the models can capture meaningful signals from the labels. For transfer learning with pretrained models, the knowledge from larger image datasets are leveraged and shared to fine tune for our tasks, which has promising results.

After generating the embeddings from each of the three architectures, ANNOY, FAISS, and Nearest Neighbor were used to retrieve images. Based on precision and recall @ 4, ANNOY, which uses forests (an ensemble of trees) to enable image search in the embedding space, performed better than the other two search methods. One reason is that the number of trees as a parameter can be tuned and better performance can be achieved through experiments. It is also considerably faster when dealing with larger datasets with high dimensionality.

While the pretrained ResNeXt generally demonstrated the best performance from a precision and recall perspective, qualitative inspection of the search results would suggest that these metrics don't do any of the search methods justice. Despite having relatively lower precision and recall, the other embedding techniques yielded similar looking images. This shortcoming on precision and recall stems from multiple factors. Firstly, when the images were labeled, only the most frequently occurring object was chosen as the label, so naturally information was lost in this process. Additionally, even the comprehensive labels did not capture things like saturation, texture, or color schemes, which seemed to be captured to an extent by the embedding techniques and reflected in the search results. While in the end the simpler models didn't score as well on precision and recall as the transfer learning approaches, the results do suggest that simpler models could be tuned to achieve comparable results given more time and resources. Overall, these exercises reaffirmed the notion that CBIR can be achieved using a variety of image embedding techniques and without searching a database of labels.

## References

[1] ANNOY library. https://github.com/spotify/annoy. Accessed: 2017-08-01. 5

[2] Denoising autoencoders explained. https://towardsdatascience.com/denoising-autoencoders-explained-dbb82467fc2. 3

[3] Docem github repository. https://github.com/gm-spacagna/docem. 4

[4] Faiss github tutorial. https://github.com/facebookresearch/faiss/wiki/Getting-started. 5

[5] Most popular distance metrics used in knn and when to use them. https://www.kdnuggets.com/2020/11/most-popular-distance-metrics-knn.html. 5

[6] Nearest neighbors classification. https://scikit-learn.org/stable/modules/neighbors.html. 5

[7] Resnet implementation with pytorch from scratch. https://niko-gamulin.medium.com/resnet-implementation-with-pytorch-from-scratch-23cf3047cb93. 4

[8] Kaiming He et. al. Deep residual learning for image recognition. 2015. 2, 4

[9] Saining Xie et. al. Aggregated residual transformations for deep neural networks. 2017. 2, 4

[10] Jeff Johnson Hervé Jegou, Matthijs Douze. Faiss: A library for efficient similarity search. https://engineering.fb.com/2017/03/29/data-infrastructure/faiss-a-library-for-efficient-similarity-search/, 2017. 5

[11] Stephen Heyman. Photos, photos everywhere. https://www.nytimes.com/2015/07/23/arts/international/photos-photos-everywhere.html, 2015. 1

[12] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks, 2018. 3

[13] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *arXiv preprint arXiv:1702.08734*. 5

[14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. 3, 4

[15] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014. 1, 2

[16] Loris Nanni, Gianluca Maguolo, and Alessandra Lumini. Exploiting adam-like optimization algorithms to improve the performance of convolutional neural networks, 2021. 3, 4

[17] D Pandey and S Kushwah. A review on cbir with its advantages and disadvantages for low-level features. *Int. J. Comput. Sci. Eng.*, 4(7):161–167, 2016. 1

[18] Yoshua Bengio Pierre-Antoine Manzagol Pascal Vincent, Hugo Larochelle. Extracting and composing robust features with denoising autoencoders. 2

[19] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. 5

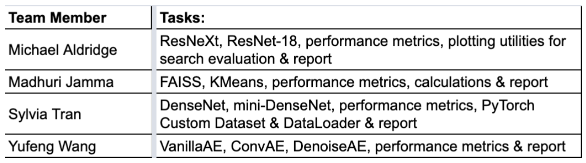
[20] Gianmario Spacagna. Extracting rich embedding features from coco pictures using pytorch and resnext-wsl. https://towardsdatascience.com/extracting-rich-embedding-features-from-coco-pictures-using-pytorch-and-resnext-wsl-vademecum-of-6fdbdbe876a8, 2020. 4

[21] R. Kala Varun Kumar, G. Nandi. Denoising autoencoder. 3

## 5 Project repository

Project Code Github repo.
URL: https://github.com/rage-against-the-machine-learning/cs7643-DL-Group-Project

## 6 Work division



| Team Member | Tasks: |
|---|---|
| Michael Aldridge | ResNeXt, ResNet-18, performance metrics, plotting utilities for search evaluation & report |
| Madhuri Jamma | FAISS, KMeans, performance metrics, calculations & report |
| Sylvia Tran | DenseNet, mini-DenseNet, performance metrics, PyTorch Custom Dataset & DataLoader & report |
| Yufeng Wang | VanillaAE, ConvAE, DenoiseAE, performance metrics & report |

Figure 8: Division of Work Summary.

**Michael Aldridge** Lead team member for residual learning models; Integrated ResNeXt transfer learning approach with custom dataset and DataLoader; Conducted performance benchmarking using the ResNeXt model; Trained ResNet-18 model from scratch; Implemented Precision and Recall functions for KNN search method; Build plotting utilities for search evaluation and results; Met with team on a weekly basis; Report contributions included Residual Learning section, Abstract, Introduction, Results, and Conclusion.

**Madhuri Jamma** Worked on the Retrieval part of the project. Academic survey of CBIR embedding methods. Performed literature survey for different clustering techniques, and implemented FAISS as a retrieval method. Rewrote baseline plotting code for precision/recall procedures for FAISS. Tried different clustering techniques such as KMeans, hierarchical clustering. Helped with writing the report. Met with the team on a weekly basis. Helped with Latex formatting. Code contribution: Image search using FAISS; Recall @ K, Precision @ K for Densenet, Densenet Inspired; AutoEncoder; ResNext, and ResNet-18 Embeddings. Report contribution: Abstract, CBIR pipeline explanation, Dataset explanation, Introduction, Image retrieval.

**Sylvia Tran**: Setup and hosted AWS S3, and team GitHub repository for team to use. Coded AWS helper functions to interact with S3. Coded PyTorch DataSet DataLoader class objects, created joint documents in a shared Google Drive to facilitate centralizing information to build a cohesive report. Met with the team on a weekly basis. Additional code contributions include: Recall @ $k$ for KNN, Precision Recall @ $k$ for ANNOY. Calculated precision recall for (i) DenseNet and (ii) DenseNet-Inspired for each (a) ANNOY, (b) Nearest Neighbors.

Report contributions include: Introduction, Densenet; Experiments Results; Precision Recall discussion and aggregation of scores across all permutations of embedding techniques and search algorithms attempted.

**Yufeng Wang** Propose this project; Read academic surveys and papers on this topic and list few options for solution; Build Colab Pro and AWS helping function to transfer data from AWS S3; Implement AutoEncoder and its variants in Pytorch and generate embeddings for image datasets; Create precision and recall evaluation for AutoEncoder models; Retrieve similar images with Annoy and Nearest Neighbor; Met with the team on a weekly basis.

Code Contribution: AutoEncoder (Linear; CNN; DeNoising); Annoy Indexing; Nearest Neighbor Search; Evaluation on Precision and Recall;

Report contributions: Plots for Approach Overview, AutoEncoders; Experiments and Results; Conclusions and future works.

## 7 Appendix

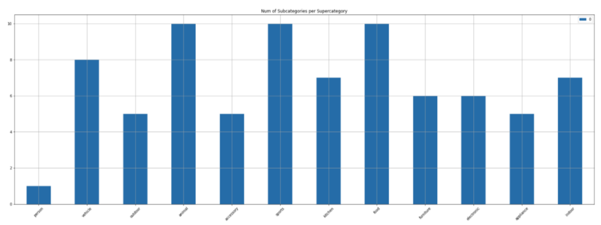


Figure 9: Number of subcategories per COCO super-category