Abstract Image Similarity with Pre-Trained Image Embeddings

Bachelor's Thesis

Degree Programme in Computer Applications

Spring 2025

Ron Kosova

HAMK
Häme University
of Applied Sciences

Abstract

DP	Computer Applications
Author	Ron Kosova
Subject	Abstract Image Similarity with Pre-Trained Image Embeddings
Supervisors	Mazhar Mohsin

Year 2025

This thesis aims to show that systems that understand images abstractly are easily achievable using modern advancements in computer vision, specifically utilizing the findings of neural style transfer research, as some of the first and most successful approaches to decoupling and representing style and content separately. It also attempts to compare this to more modern, natural language guided systems of image understanding.

The structure of the thesis is as follows: firstly, the theoretical background is described with special focus being placed on the concepts and models that make up the backbone of the systems presented in this thesis. Secondly, the tools and methodologies used for both the theory-based information gathering and the practical implementations and evaluations are listed and described. Thirdly, the implementation details are outlined in which the theoretical knowledge outline in earlier sections is put to practice. Finally, the results with respect to the evaluations are shown and possible explanations for the behaviours of the systems are discussed along with possible future research that may improve this behaviour.

Overall, this thesis designs and presents a number of systems built upon these theoretical ideas that, without any training beyond their pretext tasks, achieve good results in abstract image representation. Beyond this, the thesis also presents some valuable insights into the way the models forming the backbone of these systems “see” and “understand” images by analysing the results of the qualitative evaluation. Finally, the work presents a few solutions and extensions that build upon its findings.

Keywords	machine learning, abstract image representation, clip, vgg, resnet
Pages	48 pages and appendices 7 pages

Table of Contents

Figures

Tables

Equations

Pseudocode

Appendices

1 Introduction

Images are very semantically complex data. At a low level they are represented as a multi-dimensional array of pixel values, but at a high level they can encode complex abstract information. This generally includes objects, scenes, textures, and the way all of these components interact with each other. This thesis shows that this plethora of information is enough to compute the similarity of images. However, by the nature of abstract information, accessing it necessitates a system that “understands” images. This is where machine learning provides a breakthrough.

It has been shown, via research concerning style transfer, that CNNs trained on object recognition learn hierarchical representations of images that are increasingly more representative of object information (Gatys et al., 2015a). This means that CNNs trained on object recognition provide a way to extract content information from images. Fortunately, object recognition is a cornerstone problem of computer vision, producing many strong contenders. Many of these models are products of the ImageNet Large Scale Visual Recognition challenge, a benchmark dataset (Russakovsky et al., 2015). Products of this challenge include the VGG family of models (Simonyan & Zisserman, 2015), and the ResNet family of models (He et al., 2015), which are considered as possible feature extractors in this thesis. Along with object information, research has shown that feature maps from CNNs trained on object recognition can be used to extract texture information from images, which can be used to represent the “style” of images (Gatys et al., 2015a, 2015b).

Using only pre-trained CNNs the systems presented in this thesis can very reliably conduct image similarity computations using only the aforementioned representations of “content” and “style” in an image. These systems are compared directly with rich state-of-the-art, natural-language supervised image embeddings. Finally, the thesis also provides a simple proof-of-concept implementation of an end-to-end image searching system that focuses on both efficiency and effectiveness.

This work sets out to, and answers, these 3 research questions:

1. Can style transfer-based representations be used to compare images?
2. Can style transfer-based representations be used to formulate coherent and viable image recommendation/retrieval systems?

3. How do CNN-based systems compare to natural language-supervised systems like CLIP?

2 Machine Learning

I. Goodfellow et al (2016, p. 96) defines Machine Learning (hereafter ML) as “a form of applied statistics… with an emphasis on the use of computers to estimate complicated functions…”. This definition can be expanded with the definition of “learning” put forth by Mitchell (1997, p. 2):

“A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks T, as measured by P, improves with experience E.”

Connecting these definitions, one can define ML as the utilization of applied statistical methods and the efficiency of computers to learn a complex solution (function) to a problem such that the performance of a solution rises with experience. Even so, this definitions is quite broad. This is evident from the wide variety of ML algorithms that vary both in methodology and complexity. However, with the advancement of modern computational methods, Artificial Neural Networks, specifically Deep Neural Networks, have arisen as the foremost solution to complex problems. (IBM, n.d.)

Mohri et al. (2018, p. 1) defines the purpose of ML as that of “…designing efficient and accurate prediction algorithms”. This implies that ML can be applied to any problem that can be reformulated as a “prediction problem”. ML owes its practicality to the fact that many problems can be reformulated as such. Some examples include:

- **Language Modelling** – the main goal of which is to learn the probability distribution of sequences, however this problem then gets broken down into the chain rule which can be interpreted as *predicting* the next token in a sequence given the previous ones.
- **Object Recognition** – which can be formulated as learning the conditional probability of a class label given an image. This can be thought of as predicting the class of the object in the image.
- **Game Playing** – which can be thought of as the prediction of the next move or action in a game given the game state.

The reformulation can and is applied to many more different applications (Mohri et al., 2018).

ML approaches can be broken down into many different classes based on the problem they attempt to solve, the way they learn, the data they process, their architecture, etc. (Mohri et al., 2018). One of the main distinctions between different ML approaches is the paradigm used in their training (Mohri et al., 2018). The systems proposed in this thesis use models that have been (generally) trained in one of two training paradigms: supervised and unsupervised.

Simply put, *supervised learning* is the paradigm of learning by example. What this means is that in such a paradigm, the learner is given *labeled data* to learn from. Labeled data consists of a set of predictors/features/independent variables (terminology differs from use-case to use-case) and an associated set of labels/outputs/responses. The goal of supervised learning is for the model to then learn the relationship between these two sets such that, in the future, it can make predictions about the unlabeled sets of predictors (Hastie et al., 2009, p. 9; James et al., 2023, p. 25; Mohri et al., 2018, p. 6). This is generally implemented as an optimization process where the model being trained uses the predictors to make a prediction, this prediction is compared to the associated ground truth labels of the predictors via an appropriate measure, called the error/cost/loss, and the model is updated in an attempt to minimize this error (Hastie et al., 2009, p. 485). Generally, this optimization process is gradient based.

Unsupervised learning is the paradigm of learning from unlabeled data. This in itself is a more difficult problem as there is no guidance for the model since it is difficult to evaluate its performance. The goal of unsupervised models is, in general, to learn *about* the data. I.e., its properties, the properties of different observations and different variables, the relationships between observations, the relationships between variables, etc.. This information is then used differently based on different tasks (Hastie et al., 2009, p. 486; James et al., 2023, p. 25; Mohri et al., 2018, p. 6). Some prominent examples of such tasks are dimensionality reduction and clustering which learn smaller representations of observations and group “similar” (based on some metric appropriate to the problem at hand) observations respectively without any supervision (Hastie et al., 2009, p. 486; James et al., 2023, p. 25).

3 Convolutional Computer Vision and Image Understanding

Many modern problems can be modelled around the context of vision and imagery and understanding the "meaning" of that imagery. Problems such as autonomous cars and automated medical image processing have led computer vision to become one of the cornerstones of ML. This can be seen by the number of proposed solutions to popular computer vision challenges such as the ILSVRC (Russakovsky et al., 2015).

3.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a type of Deep Neural Networks that are used to model grid-like data (I. Goodfellow et al., 2016, p. 326). This directly implies that CNNs are well suited for processing images, which can be thought of as multi-dimensional grids of values, with each cell representing (a component of) a pixel of the image.

3.1.1 Convolution

I. Goodfellow et al. (2016, p. 326) defines CNNs as neural networks that use convolution in at least one layer. Convolution is a mathematical operation with a special definition in ML:

Equation 1. Convolution

$$F(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$

… where $I \in \mathbb{R}^{h_I \times w_I}$ is the input and $K \in \mathbb{R}^{h_K \times w_k}$ is the kernel (both are two dimensional arrays in this case, which is usually the case in single-channel images such as grayscale images). However, for convenience, it is usually implemented in an alternative form:

Equation 2. Cross-correlation

$$F(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n)$$

… called *cross-correlation* (I. Goodfellow et al., 2016, p. 329).

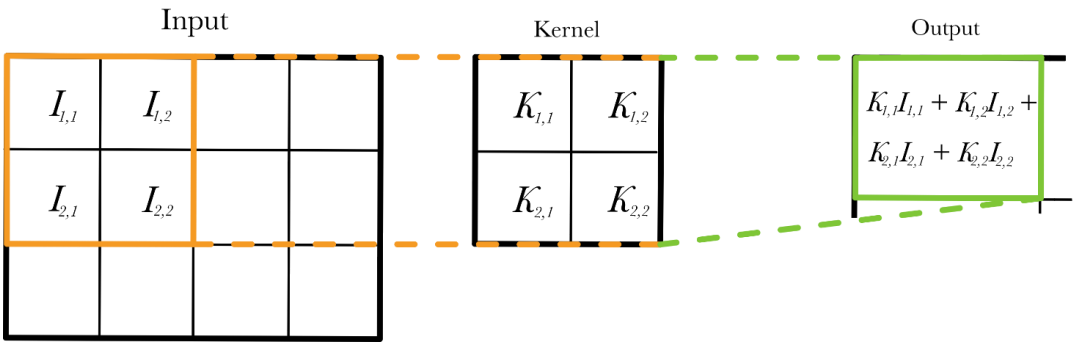
3.1.2 Kernels

In convolutional layers, the kernels and the biases make up the learnable parameters, they are responsible for feature extraction from the input data (LeCun et al., 1989). Kernels can be thought of as learnable filters, allowing CNNs to learn to feature select. Furthermore, by applying the same learned kernels over different positions of the image, CNNs take advantage of parameter sharing. Parameter sharing is important both in effectiveness and efficiency; it greatly reduces the number of free parameters without reducing the model's effectiveness as it incentivizes learning generalizable features (LeCun et al., 1989).

3.1.3 Feature Maps

The definition presented by Equation 2 can be thought of as defining a matrix $F \in \mathbb{R}^{h_F \times w_F}$ which is indexed by $i, j \in \mathbb{N}$. These indices represent the position of a unit on the output F of the convolutional operation, the *feature map*. The feature map represents the features extracted by the K kernel as it is applied over an image (LeCun et al., 1989). The process by which a feature map is computed via convolving a kernel over an image can be seen in Figure 1.

Figure 1. Application of Convolution



As per Equation 2 and Figure 1, each cell/unit (i, j) on the feature map is calculated by taking the sum of the element-wise product of each pixel on the input with its respective kernel value. The area over the input image where the kernel is applied is called the *receptive field*. Each cell of the feature map represents an activation of the feature map's kernel(s) over the unit's respective receptive field on the input image.

In simple cases, the size of the feature map is dependent on the *stride* of the convolution and the *size* of the kernel. The stride represents the distance between sequential applications of the convolution over the original image. As per PyTorch documentation (PyTorch, n.d.), the size of a feature map given an input I, kernel K, and stride S (all two dimensional in this case) is:

Equation 3. Feature map height

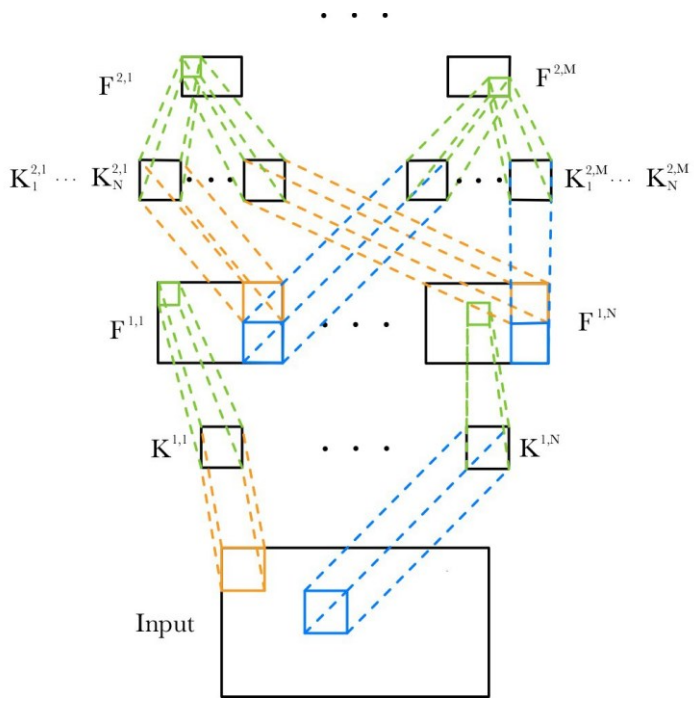
$$h_F = \left\lfloor \frac{h_I - (h_K - 1) - 1}{h_S} + 1 \right\rfloor$$

Equation 4. Feature map width

$$w_F = \left\lfloor \frac{w_I - (w_K - 1) - 1}{w_S} + 1 \right\rfloor$$

… where h_S and w_S are the stride length along the height and width respectively. A simplified example of a CNN can be seen in Figure 2.

Figure 2. Example of two convolutional layers



Due to the fact that feature map units usually have receptive fields larger than 1 unit/pixel and that strides may be higher than 1 some localization information is lost when applying a kernel over an input. However, approximate localization is generally sufficient (LeCun et al., 1989).

3.2 Convolution-based Extractors

As mentioned in the Introduction, this thesis will analyze the expressive abilities of different convolution-based architectures in the context of the proposed similarity computation methodology. The first is a simple but effective approach in the VGG family of networks (Simonyan & Zisserman, 2015) and the second is a more complex but more efficient architecture in the ResNet family of networks (He et al., 2015). Both models have a history of good performance on the ILSVRC in their respective submission years.

3.2.1 VGG

The VGG family of architectures is designed to take advantage of very deep networks. These architectures achieve this by utilizing smaller kernels in all layers (Simonyan & Zisserman, 2015).

The members of the VGG family of architectures all follow a similar configuration in their convolution layers inspired by the benchmark models of the time, such as earlier shallow CNNs (Ciresan et al., 2011; Krizhevsky et al., 2012) with some important differences revolving around the depth of the networks (Simonyan & Zisserman, 2015). The best performing models of the time were shallower networks with comparatively large receptive fields (of sizes such as 11×11) (Krizhevsky et al., 2012). This is because even though it has been shown that depth correlates to performance (I. J. Goodfellow et al., 2014), this also implies an increase in computational cost as the number of parameters increases rapidly. VGG networks, however, are efficient despite their increased depth. And this is due to their usage of smaller receptive fields (as small as 3×3). This is because larger receptive fields can be achieved by stacking smaller receptive fields on top of each other. E.g., a receptive field of size 5×5 can be achieved by layering two kernels with receptive fields of size 3×3. The benefit of this layering is that, when parametrized, the size of the set of parameters of the two smaller kernels is smaller than that of the single, larger kernel. Assuming the number of channels C remains the same after convolution, two 3×3 kernels

have $2(3^2C^2) = 18C^2$ parameters whereas a single 5×5 kernel has $5^2C^2 = 25C^2$ (Simonyan & Zisserman, 2015).

Other than in depth, VGG networks do not differ much between themselves: they all use 3×3 kernels with stride 1 and padding 1 exclusively (which ensures that the dimensions of the spatial dimensions of the output match those of the input). The convolution layers are separated into five groups, delimited by max-pooling layers with receptive fields of size 2×2 and stride 2 which effectively downsamples the input by a factor of 2. Each group doubles the number of channels from the previous group. This keeps the amount of information between groups consistent (Simonyan & Zisserman, 2015). Given some input $F_n \in \mathbb{R}^{C \times h \times w}$, any of these downsampling groups would produce and output of $F_{n+1} \in \mathbb{R}^{2C \times \frac{h}{2} \times \frac{w}{2}}$. VGG models compensate the loss of information from the halving of the spatial dimensions of downsampled feature maps by doubling the number of channels, and thus the number of kernels/learnable features per feature map.

This thesis focuses only on the ImageNet pre-trained version of VGG19 – the 19-layered version of VGG, with 16 convolutional layers, trained on the ImageNet dataset (Russakovsky et al., 2015). These convolutional layers will act as the feature extractors of the VGG-based solutions presented in this thesis. VGG19 is chosen because of its balance of expressive ability and efficient design.

3.2.2 ResNet

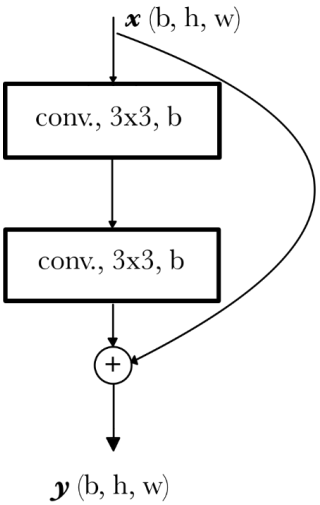
The ResNet family of architectures is based upon the concept of residual learning (He et al., 2015). Residual learning reformulates the learning of a desired underlying mapping $H(x)$ by a combination of non-linear layers to the form of the residual function $R(x) = H(x) - x$. The underlying mapping then becomes $H(x) = R(x) + x$. In this formulation, the layers are made to learn the residual function $R(x)$ by adding the input x to their output $R(x)$ via *skip connections* instead of directly learning $H(x)$ (He et al., 2015).

This reformulation is motivated by the so-called “degradation problem”; the accuracy of a network will increase with depth until it reaches a saturation point, past which it will actually start to degrade (He & Sun, 2014). This is counterintuitive considering that for every shallow network F with performance P there exists a deeper network G with performance P. This is because any deeper network G can be composed of any of its shallower networks F such that $G(x) = I(F(x))$ where I is an identity mapping. As much, for any shallow network,

every deeper network theoretically has a performance floor equal to the performance of the shallow network (He et al., 2015).

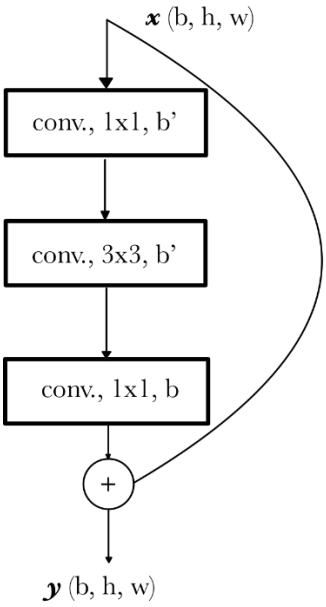
The ResNet architecture is made up of building blocks, which have themselves varying architectures. The shallower networks such as ResNet34 use building blocks made up of 2 stacked non-linear layers with a skip connection around them as described in Figure 3 (He et al., 2015).

Figure 3. ResNet Building Block



Whereas the deeper, better-performing networks use a practical compromise that keeps the same time complexity while making training less technically expensive; the “bottleneck” building block. So called due to its structure of (generally) 3 stacked non-linear layers in an arrangement such that the first layer down-samples the input from its original dimensionality b to a smaller dimensionality b' (usually as a 1×1 convolution), the second layer then performs feature extraction on the down-sampled input, and the third layer projects the input back to its original dimensionality, as depicted in Figure 4 (He et al., 2015).

Figure 4. ResNet "Bottleneck" Building Block



The building blocks (for both shallow and deep networks) are arranged sequentially. Much like VGG, the components of ResNet are delimited into different groups by down-sampling layers, however instead of pooling layers, convolutional layers are used directly to down-sample (with stride 2×2). Each group is made up of multiple building blocks that retain the same size and number of feature maps. That is, except for the first layer of every group, which down samples the input by half while doubling the number of feature maps. This also necessitates a special skip-connection which matches the dimensionality of the input to that of the output. This is usually implemented through a learned linear projection of the input (usually as a 1×1, stride 2 convolution which also doubles the feature maps on the input)(He et al., 2015).

3.3 Content and Style

This thesis bases its approach to image content and style expression on the findings of style transfer research. Research shows that convolutional layers of a CNN trained on image recognition encode information about the content of the image. Along the hierarchy of convolutional layers this information is increasingly less explicit and more abstract, focusing less on the original pixel values and more on the content of the image (Gatys et al., 2015a). For style, this thesis uses two different approaches of representing style using statistical information from the activations of different convolutional layers.

The first, and more descriptive approach, is presented in (Gatys et al., 2015b). Here, style is represented as the texture of the image, computed as the coactivation rate of the input's feature maps at the output of each down-sampling layer of the extractor network. This is computed as the Gram matrix of the flattened feature maps of each style layer.

Equation 5. Gram matrix of layer l

$$G^l = F^l {F^l}^\top$$

… where $G^l \in \mathbb{R}^{n_l \times n_l}$ is the Gram matrix of layer l, $F^l \in \mathbb{R}^{n_l \times d_l}$ is the matrix of the flattened feature maps of layer l, n_l is the number of feature maps at layer l and d_l is their flattened size. While descriptive, this representation of style is very space-inefficient. This is due to the fact that, as per Equation 5, the size of the Gram matrix is dependent on the number of feature maps. As noted in the section on Convolution-based Extractors, both models considered here will double the number of feature maps at each down-sampling layer. This means that the Gram matrices of later layers will reach very large sizes, increasing quadratically, becoming prohibitive to store and compute similarity with efficiently.

It is for this reason that this thesis explores another representation of style inspired by the findings of channel-wise feature map distribution mapping in style transfer (Huang & Belongie, 2017; Li et al., 2017). These works find that, besides using Gram matrices, channel-wise distribution statistics such as the mean and standard deviation of feature maps work well as a more compact representation of style.

Equation 6. Mean and Std. Deviation of ith feature map of layer l

$$\mu^i_{F^l} = \frac{1}{d_l}\sum_{k=1}^{d_l} F^l_{ik}, \quad \sigma^i_{F^l} = \sqrt{\frac{1}{d_l}\sum_{k=1}^{d_l}\left(F^l_{ik} - \mu^i_{F^l}\right)^2}$$

4 Transformers and CLIP

As mentioned in the Introduction, this thesis also includes a simple similarity system implemented using CLIP, a natural-language supervised visual model implemented using transformers (Radford et al., 2021). The point of such a system is to provide a state-of-the-art solution for comparative analysis.

CLIP (Contrastive Language-Image Pretraining) bridges the gap between vision and language, and as such, requires architectures that are performant in both mediums. It achieves this by using the transformer architecture, for both vision (Visual Transformer) and language.

4.1 Transformers

Transformers are models based on self-attention, devised with the intent of providing a solution for sequential modelling problems. The main difference between transformers and other sequential modelling techniques is that the transformer architecture is not recurrent. Since transformers are not recurrent they are able to be parallelized, providing a much more performant solution to sequence modelling. (Vaswani et al., 2017)

4.1.1 Self-Attention

The way that the transformer avoids recurrency is through self-attention (Vaswani et al., 2017). Self-attention is a special case of attention. Attention as an idea in machine learning has a long history, but the attention mechanism used by transformers originates from NLP (Bahdanau et al., 2016; Brauwers & Frasincar, 2023). This type of attention mechanism, initially devised for encoder-decoder RNN models, bridges the gap between the encoder and the decoder, allowing the decoder to learn to selectively attend to the encoded source for context when generating a token instead of relying on the encoder to encode all of the information (Bahdanau et al., 2016). Self-attention, on the other hand, allows each token in the sequence to attend to every other token in the sequence (Vaswani et al., 2017).

Like RNN-based models, Transformers are also made up of encoders and decoders, both of which make use of self-attention. The encoder uses self-attention to simultaneously generate representations of the input sequence, where each token can attend to every other token. This allows the model to efficiently capture dependencies across the entire

sequence without the need for sequential processing of the tokens (Vaswani et al., 2017). The decoder uses a modified version of self-attention, called *masked self-attention*. In masked self-attention, the attention weights of a token at position i are adjusted so that it cannot attend to tokens beyond position i. Since the decoder is given the true output sequence during training, this masking is crucial during training because it allows the decoder to predict the entire output sequence in parallel while ensuring that he model retains its autoregressive nature during inference. That is, the masking prevents the decoder from "looking ahead" by attending to future tokens when predicting a token during training, thereby maintaining the integrity of the sequence generation process during inference. (Vaswani et al., 2017)

4.1.2 Scaled Dot-Product Attention

The general attention model can be represented in terms of operations on three entities: the query matrix Q, the key matrix K, and the value matrix V. These three are then used for *scoring*, *alignment*, and finally for computing the *context vector*. The values of the query matrix depends on the use case, whereas the key and value matrices are usually linear projections of the input of the attention model:

Equation 7. Key and Value matrices

$$K = W_K \times X$$
$$V = W_V \times X$$

… where $W_K \in \mathbb{R}^{d_K \times n_X}$, $W_V \in \mathbb{R}^{d_V \times n_X}$, $X \in \mathbb{R}^{d_X \times n_X}$. (Brauwers & Frasincar, 2023) An example of such a system is the already mentioned Bahdanau (Additive) attention:

Equation 8. QKV Bahdanau Attention

$$c_i = V\alpha_{:i}^{\top}$$
$$\alpha_{ij} = \frac{e^{E_{ij}}}{\sum_{k=1}^{n_K} e^{E_{ik}}}$$
$$E_{ij} = w^{\top}\sigma(W \times q_i + U \times k_j + b)$$

… where $q_i \in \mathbb{R}^{d_Q}$, $k_j \in \mathbb{R}^{d_K}$, and $v_j \in \mathbb{R}^{d_V}$ are the query, key, and value column vectors in matrices $Q = \left[q_1, q_2, \ldots, q_{n_Q}\right] \in \mathbb{R}^{d_Q \times n_Q}$, $K = \left[k_1, k_2, \ldots, k_{n_K}\right] \in \mathbb{R}^{d_K \times n_K}$, and $V =$

$[v_1, v_2, \dots, v_{n_V}] \in \mathbb{R}^{d_V \times n_V}$, where $n_K = n_V$. $W \in \mathbb{R}^{d_e \times d_Q}$, $U \in \mathbb{R}^{d_e \times d_K}$, $w \in \mathbb{R}^{d_e}$, and $b \in \mathbb{R}^{d_e}$ are the weights and biases of the *scoring model* and σ represents a non-linear activation function. $E_{ij} \in E$ where $E \in \mathbb{R}^{n_Q \times n_K}$, represents the score of query vector i and key vector j, $\alpha_{ij} \in \alpha$, where $\alpha \in \mathbb{R}^{n_Q \times n_K}$, represents the aligned scores of query vector i with key vector j via the Softmax function, and $c_i \in C$, where $C = [c_1, c_2, \dots, c_{T_d}] \in \mathbb{R}^{d_V \times n_Q}$, represents the context vector of query i. (Bahdanau et al., 2016; Brauwers & Frasincar, 2023) In the context of sequence-to-sequence modelling for example, the key and value matrices are the encoder annotations $H = [h_1, h_2, \dots, h_{T_e}] \in \mathbb{R}^{d_h \times T_e}$. In this case, W_K and W_V can be thought of as identity matrices. The query matrix is the matrix of decoder states $S = [s_1, s_2, \dots, s_{T_d}] \in \mathbb{R}^{d_s \times T_d}$. (Bahdanau et al., 2016)

Transformers, however, use a different attention mechanism, called scaled multiplicative attention or *scaled dot product attention*, named after its use of the dot product for its scoring method (Brauwers & Frasincar, 2023; Vaswani et al., 2017). In QKV format:

Equation 9. QKV Scaled Dot Product Attention

$$C = VA$$
$$A = \text{softmax}\left(\frac{S}{\sqrt{d_e}}\right)$$
$$S = K^\top Q$$

…where:

Equation 10. QKV Matrices in Transformers

$$Q = W_Q X$$
$$K = W_K Z$$
$$V = W_V Z$$

… where $X \in \mathbb{R}^{d_X \times n_X}$ and $Z \in \mathbb{R}^{d_Z \times n_X}$, which are the primary and context sequences respectively. $W_Q \in \mathbb{R}^{d_e \times d_X}$, $W_K \in \mathbb{R}^{d_e \times d_Z}$, and $W_V \in \mathbb{R}^{d_O \times d_Z}$, where d_e is the attention dimension size and d_O is the attention output dimension size, form the weights that are used to linearly transform X and Z into Q, K, V matrices (Phuong & Hutter, 2022; Vaswani et al., 2017). As mentioned, the difference between Bahdanau attention and scaled dot-product attention is that the latter uses the dot-product of the keys of the context tokens

and the queries of the primary tokens to represent the attention scores $S \in \mathbb{R}^{n_z \times n_x}$. Thus, S_{ij} represents the dot product (score) of the key of context token k_i and query of context token q_j. The score is then scaled by the root of the attention dimension size for numerical stability. The benefit of scaled dot-product attention is that it uses optimized vector operations thus it is more computationally efficient. As such, it is also scalable to larger sequence lengths. (Brauwers & Frasincar, 2023; Vaswani et al., 2017)

Transformers use scaled dot-product attention for both cross-attention and self-attention. In the case of cross attention, much like with Bahdanau attention in sequence-to-sequence problems, the output of the encoder is used as the context sequence Z, whereas the output of the previous module in the decoder is used as the primary sequence X. In the case of self-attention, the output of the previous module is used as both the primary and context sequence, thus $X = Z$. (Vaswani et al., 2017)

Transformers also make use of multiple attention heads in parallel. This means that for any input to the attention mechanism, transformers will apply attention h times, where h is the number of heads, and combine the outputs. This allows the transformers to learn and attend to information from different representations of the data. (Phuong & Hutter, 2022; Vaswani et al., 2017)

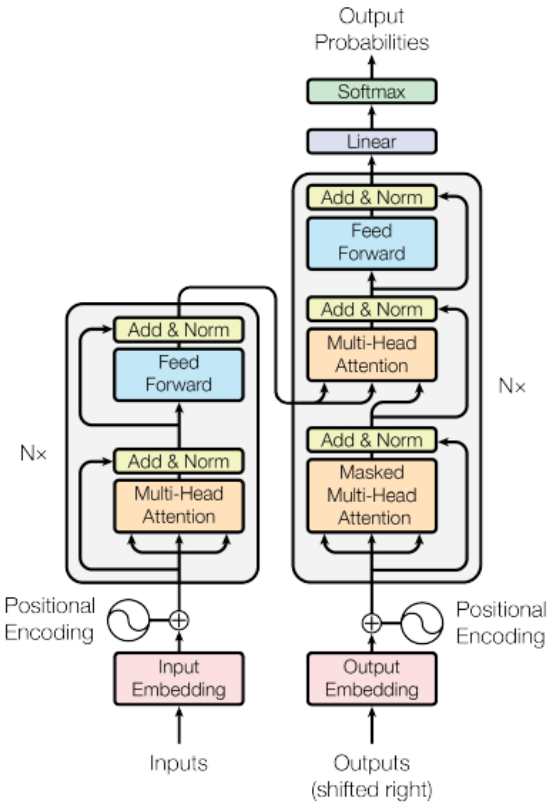
4.1.3 Structure

As mentioned above, Transformers use the encoder-decoder paradigm in sequence-to-sequence modelling problems (Vaswani et al., 2017). Both can be described as the product of discrete blocks and the operations between them, as per Figure 5. The encoder block is less complex, being made up of less blocks and operations. It is made up of two sub-blocks: a multi-head self-attention layer and a feed-forward network. A residual connection is implemented around both sub-blocks such that the input of a block is added to the output of the block (necessitating that the input and output are of the same dimensions). Layer normalization (Ba et al., 2016) is then applied to each residual connection. The encoder module is made up of N encoder blocks stacked sequentially. (Vaswani et al., 2017)

The decoder block is very similar to the encoder block with a few notable differences. Firstly, it uses a masked multi-head self-attention sub-block to enforce the auto-regressive nature of the decoder module. This masked self-attention block is immediately followed by a multi-head cross-attention block. This is the block that facilitates the connection between

the encoder and the decoder. Finally, the decoder implements a simple feedforward network. Similarly to the encoder blocks, every sub-block in the decoder implements a residual connection followed by layer normalization. The decoder module is also made up of N decoder blocks stacked sequentially, with the output of the final decoder block used to predict the next token in the sequence, which is appended to the decoder input and fed back to the decoder. (Vaswani et al., 2017)

Figure 5. Transformer diagram (Vaswani et al., 2017)



Both modules implement layer normalization, which makes training of complex models like the Transformer quicker. Layer normalization at a layer l is the normalization of the summed a^l inputs of the neurons at layer l:

Equation 11. Layer Normalization

$$\bar{a}_i^l = \frac{g_i^l}{\sigma^l}(a_i^l - \mu^l)$$

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \,, \quad \sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left(a_i^l - \mu^l\right)^2}$$

$$a^l = Wh^l \,, \quad h^l = f(a^{l-1} + b^{l-1})$$

…where $\bar{a}_i^l$ is the vector of normalized inputs to the neurons in layer l, g^l is a vector of learnable gain parameters that are used to scale the normalized inputs, H is the number of neurons in the layer l, W is a learnable linear transformation which is applied to h^l, the vector of the outputs of the previous layer of neurons. (Ba et al., 2016) As it can be seen from Equation 11, the normalization for each summed input to the neurons is performed using the layer statistics μ^l and σ^l which are scalars that represent the mean and standard deviation of the summed inputs of layer l respectively.

Due to its structure, self-attention lacks the inductive bias for token order. Inductive biases refer to the set of assumptions that a machine learning model makes about the data it is designed to model (Mitchell, 1980). This means that self-attention does not inherently assume that token order in a sequence is important (Vaswani et al., 2017). To address this, Transformers implement positional encodings in the sequence inputs for both the encoder and decoder module. These ensure that the Transformer inherently takes into account the order of tokens when learning the relationships between them. Typically, positional embeddings are implemented via periodic functions; in the original Transformer model, the positional encodings are implemented using sine and cosine functions. (Vaswani et al., 2017)

These functions are used to generate separate values for each dimension of the token embedding based on the position of the token in the sequence and the embedding dimension. These value are then added to the original sequence embeddings, thereby incorporating positional information. (Vaswani et al., 2017)

4.2 Vision Transformers

Vision Transformers (ViT) are an adaptation of the Transformer architecture so that It can model image data (Dosovitskiy et al., 2021). While it may sound unintuitive, using the same model for both sequence-based learning and image-based learning is not unheard of, with CNNs being used to model sequential data (Gehring et al., 2017; I. Goodfellow et al., 2016,

p. 368). However, in this case, it is the sequential model being adopted for image modelling.

The intent behind using the Transformer architecture in ViTs is to leverage its inherent scalability and efficiency. However, due to its design, the Transformer architecture does not have the same inductive biases that make CNNs so powerful when it comes to modelling image data (Dosovitskiy et al., 2021). In CNNs, key inductive biases include *translation invariance* – the assumption of the model that a feature detected in one part of the image will be relevant in other parts of the image – and *locality*, which assumes that features spatially near each-other form more important relationships than those further apart.

Locality is especially a problem when it comes to self-attention based models such as the Transformer, and by extension ViT, as self-attention inherently assumes global relationships are just as important as local ones. Despite these challenges, the efficiency of the Transformer architecture compensates for the lack of these inductive biases. ViTs can be trained on much larger datasets than is typically feasible with CNNs, which helps overcome the disadvantage of the missing inductive biases. (Dosovitskiy et al., 2021)

ViTs are designed so that they can use the Transformer architecture with little to no modifications. Specifically, ViTs make use of the encoder-only Transformer architecture. The input of the ViT is a sequence of patches $P \in \mathbb{R}^{d_P \times n_P}$ where $d_p = p^2 C$ where (p, p) is the shape of each patch and C is the number of channels. These patches are then embedded via a learned linear transformation and an extra learnable embedding $x_{\text{class}} \in \mathbb{R}^{d_e}$ is prepended to the resulting embedding sequence $X \in \mathbb{R}^{d_e \times n_X}$:

Equation 12. Patch sequence pre-processing.

$$X = \left[x_{\text{class}}, z_1, z_2, \dots, z_{n_p}\right] \in \mathbb{R}^{d_e \times n_X}$$
$$Z = \left[z_1, z_2, \dots, z_{n_p}\right] = WP \in \mathbb{R}^{d_e \times n_P}$$

… where $W \in \mathbb{R}^{d_e \times n_P}$ and $n_X = n_e + 1$. The state of the prepended token at the output y_{class} of the encoder is then used as a representation for the image. As this token is meant to present a global representation of the patch sequence, it is what is pushed through the prediction head. (Dosovitskiy et al., 2021)

4.3 CLIP (Contrastive Language-Image Pretraining)

CLIP is a multi-modal contrastive approach to learning image representations using natural language supervision. This involves training an image and text encoder on image and text pairs such that the embeddings of the components of a similar pair are closer in the embedding space than those of dissimilar pairs. (Radford et al., 2021)

In the original CLIP implementation, the encoders are both Transformer based and the "closeness" of the embeddings in the embedding space is computed via their cosine similarity (Radford et al., 2021). Cosine similarity measures the cosine of the angle between two vectors. This can be used to represent the "similarity" of two vectors as their angles; if the cosine similarity is near its lower bound (-1), the two vectors are pointing in near opposite directions, if the cosine similarity is near its upper bound (1), the two vectors are pointing in the same direction. The cosine similarity of two vectors v and u can be directly derived from the definition of their dot product. (Han & Kamber, 2012, p. 77)

Thus, CLIP training with N similar pairs consists of training the image and text encoders such that their embeddings project to a learned space in which the cosine similarity of the embeddings of similar pairs is higher than that of dissimilar pairs. This is done by maximizing the cosine similarity of the N similar embedding pairs and minimizing the cosine similarity of the $N^2 - N$ dissimilar pairs. (Radford et al., 2021)

This form of learning, where the goal is to learn a representation of space where similar data points are "closer" than dissimilar ones is called contrastive representation learning. It is a form of representation learning where the data representations are learned by "comparing" data points. As such, all that is needed to train a contrastive model is some definition of similarity. Similarity can be defined directly from data, thereby removing the need for human labeled data (Le-Khac et al., 2020). CLIP is trained by contrasting text and images, and as such similarity must be defined between them. This similarity is implicitly defined by the image-text pairs that make up the training data, as the components of each pair are considered to be similar and any pairs not found in the dataset are dissimilar. (Radford et al., 2021)

CLIP is made up of two modules – a *text encoder* and an *image encoder*. Both are trained to learn a joint embedding space where "similar" image and text embeddings are closer to each other. (Radford et al., 2021)

Equation 13. CLIP Logits

$$Z = e^{\tau}\left(\overline{P_e}^{\top}\overline{T_e}\right)$$

$$\overline{p_{e_i}} = \frac{p_{e_i}}{\|p_{e_i}\|}, \qquad \overline{t_{e_i}} = \frac{t_{e_i}}{\|t_{e_i}\|}$$

$$P_e = W_P P_f, \qquad T_e = W_T T_f$$

Equation 13 shows the logits computations needed to train CLIP with a mini-batch of n image-text pairs (Radford et al., 2021). $P_f \in \mathbb{R}^{d_P \times n}$ and $T_f \in \mathbb{R}^{d_T \times n}$ represents the features extracted from the mini-batch of images and text respectively by the learned encoders of CLIP. $W_p \in \mathbb{R}^{d_e \times d_P}$ and $W_T \in \mathbb{R}^{d_e \times d_T}$ represent the learned linear projection weights used to project the extracted features into the embedding space. $P_e \in \mathbb{R}^{d_e \times n}$ and $T_e \in \mathbb{R}^{d_e \times n}$ are the resulting embeddings from the projection of the features. $\overline{P_e} \in \mathbb{R}^{d_e \times n}$ and $\overline{T_e} \in \mathbb{R}^{d_e \times n}$ are the l_2-normalized embedding matrices such that $\|\overline{P_e}\| = 1 = \|\overline{T_e}\|$. This step is important as it simplifies the computation of the logit matrix which is computed as the scaled pairwise cosine similarity of the embeddings. As per Equation 15**,** the computation of the cosine similarity between two vectors is the quotient of their dot product and the product of their Euclidean (l_2) norms. The l_2-normalization of the embedding matrices simplifies the cosine similarity computation to just the dot product of two embedding vectors. As such, the logit matrix $Z \in \mathbb{R}^{n \times n}$, where Z_{ij} is the scaled cosine similarity of the i-th image embedding vector $\overline{p_{e_i}}$ and the j-th text embedding vector $\overline{t_{e_j}}$, can directly be computed as the product of the scaling factor e^{τ} and the matrix product $\overline{P_e}^{\top}\overline{T_e}$. Here, τ represents the temperature parameter which is also directly learned during training. (Radford et al., 2021)

During inference, for each sample, the image is pushed through the image encoder to get the image features $p_f \in \mathbb{R}^{d_p}$ and the set of possible class labels is augmented in the form of descriptive texts such as: “An image of a {class name}”. These are then pushed through the text encoder to get the text features $T_f \in \mathbb{R}^{d_t \times n}$ where n is the number of classes. These are then projected into the joint embedding space and then used to compute the logits vector $z \in \mathbb{R}^{1 \times n}$. The class is then predicted by applying softmax normalization to the logits which gives a probability distribution over the possible classes. (Radford et al., 2021)

Radford et al. (2021) tests 8 combinations of different image ecoders and a Transformer based text encoder. The findings show that the best performing CLIP models are those that use a ViT image encoder and a Transformer encoder-based text encoder. More

specifically, the findings show that the ViT encoder designated as ViT-L/14@336px is the best performing. This denotes a “large” ViT which is made up of 24 layers, uses an attention dimension of 1024 with 16 heads, and an MLP of size 4096 (Dosovitskiy et al., 2021). Slight adjustments are made to the overall architecture but in general it remains the same (Radford et al., 2021).

The feature representations P_f and T_f are derived from the final layers of these encoders. In the case of the ViT-based image encoder, the output at the (prepended) [CLS] token of the last encoder sub-module is used as the image representation (Dosovitskiy et al., 2021; Radford et al., 2021). The Transformer based text embedder follows the GPT-2 architecture in which the original Transformer decoder module is modified such that the cross-attention sub-blocks within the decoder sub-modules are removed (owing to GPT-2’s decoder-only architecture), the layer normalizations are performed ahead of each sub-block, and an additional layer normalization is performed after the final masked self-attention block. (Radford et al., 2019, 2021) CLIP retains these masked self-attention blocks in its text encoder implementations, thereby making it more similar to a Transformer decoder than encoder, despite the task of encoding text. This is to accommodate the usage of pre-trained, auto-regressive language models as text encoders. CLIP uses the representation of the [EOS] (end-of-sequence) token at the output of the final text encoder sub-module as the feature representation of text. (Radford et al., 2021) This is presumably because, in lieu of a pre-pended [CLS] token like in cases of bidirectional encoders with unrestricted self-attention, in the CLIP encoder with masked self-attention the [EOS] token is the only one to be able to attend to every other token in the sequence as it is the final token in the sequence.

5 Methods and Tools

This section describes the methodology used throughout this thesis to facilitate the theoretical research and the practical implementations. Along with this, this section also outlines the tools/technologies used in the completion of the proof-of-concept implementation described in the practical sections.

5.1 Journal as a Research Method

This thesis and the experiments within were planned using a research journal/diary. In this context, a research journal refers to a document or a set of documents and programs that is used to keep an organized and systematic approach to research and ideation (Miller, 2007). This thesis uses two main programs to facilitate such a journal: Obsidian and Zotero.

Obsidian is a Markdown based note-taking and personal knowledge base app that provides a complex and flexible knowledge management system (Obsidian, n.d.). The main draw of Obsidian when planning and writing this thesis was its “Graph view” which allows the visualization of connections between different notes within the same “Vault”. These connections are facilitated via links within the connected notes. This allows for a more holistic view of the relationship between concepts and ideas presented within the knowledge base itself (Pyne & Stewart, 2022).

Zotero is a resource management system, used in this thesis for the collection and management of all bibliographic data. This includes all research materials that are directly cited in this thesis along with some that are not. The main benefit of Zotero is its direct integration with Word, the text processing software used to write this thesis, and browsers like Chrome and Firefox, which allow it to directly collect resources from the web along with available metadata (Zotero, n.d.).

5.2 Proof-of-Concept

The practical implementations of the system presented in this thesis are implemented through proofs-of-concept. A proof-of-concept (POC) is an implementation of an idea such that its viability/potential is tested based on some objective (TechTarget, n.d.). The POC presented in this thesis are meant to show the technical feasibility of the proposed systems

in practical applications. The POCs presented here are intended to represent rudimentary ranking/retrieval systems. They are presented “as is”, meaning their performance is meant to be directly indicative of the underlying systems, as such no extra optimizations are applied. Instead of breaking the flow of the document with large chunks of language specific code, the POC implementations are shown using language agnostic and mathematical algorithms written in pseudo-code. This keeps the implementations language agnostic and provides a concise but technical and informative overview. Full code can be found in the code blocks listed in appendix 2.

5.3 Tools

This section outlines the tools and technologies used to implement and test the proposed systems and POC implementations. Generally, the main tool behind the implementation of the POCs is Python, more specifically, through the PyTorch library (Paszke et al., 2019). This is because PyTorch, among other benefits, provides direct access to the required CNN based models (VGG and ResNet) through TorchVision. Since these provided models are implemented using PyTorch they are also easily customizable. OpenAI also provides an open-source implementation of CLIP.

Other libraries are then used to facilitate different parts of this thesis, not all related to the POCs. One important case of this is the evaluation of models. This is mainly done through the use of the Python implementation of the OpenCV library (Bradski, 2008) and the SentenceTransformers (Reimers & Gurevych, 2019). OpenCV is an open source computer vision library offering many efficient implementations of functions used in real-time computer vision. In this thesis, it is mainly used to compute statistical features of images for the evaluations described in the coming section. SentenceTransformers is a Python library used for training and inference with advanced, Transformer-based text embedding models. In this thesis, its implementation of **all-mpnet-base-v2** is used to embed image captions for similarity comparisons during evaluation.

5.4 Evaluation

As mentioned in the Introduction, the problem of semantically understanding images is a difficult one, owing to the complexity of the encoding of semantic information within images. In this thesis, the semantic information is represented by using object information derived from object detection embeddings and texture information as defined by the stated works in

neural style representation. Even with this abstraction of semantic information, the number of possible combinations and their complexities are staggering. As such, it is difficult to design rigorous and inclusive datasets and experiments for evaluating systems operating on such data. To compensate for this complexity, this thesis employs two different evaluation methods, one quantitative and the other qualitative.

5.4.1 Quantitative Evaluation

The *quantitative system* is applied over 500 random images from the ArtCap dataset (Lu et al., 2024). This is a dataset of fine art paintings paired with descriptive captions. It was chosen because paintings inherently are very diverse when it comes to both texture and content. They can also be easily classified in terms of their content and texture (i.e., in terms of genre and style) which implies a well divided similarity space along these two features. The provided captions are also very rich sources of information, especially due to the recent advancements in natural language understanding.

However, even with a dataset with such desirable features, similarity evaluation still remains a difficult problem. Indeed, if there was an objective metric to evaluate how similar two images are there would be no need for any further systems. Even so, the evaluating system does not have to be perfect as long as it is consistent with the intuition of the task at hand. The quantitative evaluation system presented in this thesis aims to achieve such consistency by being intuitive and simple. It makes use of three statistics meant to represent the content and texture similarity between the target image and its most similar image according to each system. These are *caption embedding cosine similarity*, *histogram-of-gradients distance*, and *color histogram correlation*.

5.4.2 Caption embedding Cosine Similarity

Caption embedding cosine similarity is computed by taking the cosine similarity of the embedding of the caption of the target image and the embedding of the caption of the candidate image. These embeddings are generated using SentenceTransformers' *all-mpnet-base-v2* model, the best performing sentence embedder. The intuition of this statistic is to utilize the rich semantic information encoded in the captions to represent the similarity in the scene and content represented in these paintings. This can be demonstrated from a few examples of captions found in this dataset (Lu et al., 2024):

- “A woman in a long skirt in a flower garden on a spring or summer day.”
- “A large group of people are gathering outside a huge building.”
- “A man is sitting at a table and a woman standing in front of the man.”

5.4.3 HOG Distance

Histogram-of-Oriented-Gradients distance is chosen to represent one part of the similarity of the texture of the image. The histogram of oriented gradients is computed by dividing the given image in cells, where for each cell a histogram of orientations is computed. The bins of the histogram represent an equidistant distribution of angles, usually 9 of them ($0 - 180°$ for unsigned angles and $0 - 360°$ for signed angles). Using the finite difference kernels $k_x = [-1,0,1]$ and $k_y = [-1,0,1]^\mathsf{T}$ which are convolved over the cell at a stride length of 1, the magnitude and orientation of the gradient at that pixel are computed. They are then used to cast weighted votes into one or two of the nearest orientation bins. Each cell thus ends up with a histogram of oriented gradients. The results of cells are combined in overlapping blocks with neighboring cells and normalized. The block-normalized histograms are then concatenated and the resulting vector is used as a feature vector. (Dalal & Triggs, 2005) This thesis implements this using the HOG implementation of OpenCV with 9 orientations, which is based on Dalal & Triggs (2005).

These feature vectors, due to implicitly containing localized oriented gradient information, are very good at describing edges and their directions within an image. As such, in images such as paintings, where a large portion of the texture is imparted via brush-strokes and pencil lines which are present as strong edges, it represents a very strong texture descriptor. In the presented evaluations, the distance between two such vectors is interpreted as a measure of texture similarity.

5.4.4 Color Histogram Correlation

Color is also a very important component of texture in images and a good differentiator. That is, if two images have similar subjects and edges but not similar colors one cannot say that they are completely similar. To represent this similarity in the evaluations, color histogram correlation is used. A color histogram represents the frequency distribution of each color channel (RGB in this case). In this evaluation the color histogram is counted in 256 bins per channel, thus each possible value is counted in its own bin. The histograms are then concatenated and normalized to form a 768 dimensional feature vector. The

feature vectors are then compared using their sample Pearson correlation coefficient (OpenCV, n.d.):

Equation 14. Histogram Correlation

$$r(H_1, H_2) = \frac{\sum_{c \in C}[(H_1(c) - \overline{H_1})(H_2(c) - \overline{H_2})]}{\sqrt{\sum_{c \in C}[\, H_1(c) - \overline{H_1}\,]^2 \sum_{c \in C}[\, H_2(c) - \overline{H_2}\,]^2}}$$

… where H_1, H_2 are the normalized histogram vectors.

5.4.5 Qualitative Evaluation

As mentioned previously, there is difficulty in objectively evaluating abstract similarity in image. This is partly owing to the lack of a standard evaluation of similarity between images (if such a system existed, there would be no need for works such as this thesis), and partly, and arguably a cause for the first issue, the fact that image comparison is inherently not objective. This is why this thesis presents a simple quantitative evaluation as an intuitive way to compare models with a texture- and content-rich evaluation space, and a qualitative evaluation with a more diverse evaluation space. The qualitative evaluation is meant to present a more subjective evaluation of the performance of the models. To provide a more diverse set of possible candidates, the dataset used will be comprised of roughly 500 curated images from Pexels.com, sourced through its API[1]. While a few of these evaluations are shown later on in this thesis, more uncurated examples can be found in appendix 3.

[1] Pexels.com API: https://www.pexels.com/api/documentation/#photos-curated

6 Proofs-of-concept

This section outlines and describes the design and implementation of the practical section of this thesis. In it, the proposed systems are designed using the theoretical framework and implemented using the tools and methodologies described in previous sections.

6.1 Similarity Score

To facilitate ranking, each system needs to define a way to score the similarity between two images. Indeed, this functionality is the backbone of any ranking system. In the case of this thesis, the ranking systems are defined as functions $R: \mathbb{R}^{h \times w} \times \mathbb{R}^{h \times w} \to \mathbb{R}$ which take two images and outputs a real-valued similarity score based on some similarity function. The way the similarity backbone function is defined varies based on the general implementation of the solution, i.e., Gatys-based (named after the first author of the neural style transfer paper the CNN approach is based on, includes all feature map statistics-based solutions) and CLIP-based. In both cases, an important part is the cosine similarity. Cosine similarity is defined as the cosine of the angle between two vectors (Han & Kamber, 2012, p. 77).

Equation 15. Cosine Similarity

$$S_{AB} = \cos(\theta) = \frac{A \cdot B}{\|A\|_2 \, \|B\|_2}$$

Here, for vectors $A, B \in \mathbb{R}^n$, θ represents the angle between the vectors, $A \cdot B \in \mathbb{R}$ represents the dot product, and $\|A\|_2 \|B\|_2$ represents the product of the l_2-norms of the vectors. The range of cosine similarity is within the range $[-1, 1]$ where 1 means the vectors are pointing in the exact same direction ($\theta = 0$) and -1 means that the vectors are pointing in opposite directions ($\theta = \pi$). Cosine similarity is then used to define the similarity scoring functions for the systems presented.

In the case of CLIP-based similarity, cosine similarity is used directly. This is because the CLIP embeddings are 512 dimensional vectors and can be directly compared with cosine similarity. It is also well aligned with the way CLIP embedding space is constructed during training; CLIP is contrastively trained using cosine similarity as the distance metric (Radford et al., 2021).

Equation 16. CLIP-based Similarity

$$S_{XY}^{\text{CLIP}} = S_{XY}$$

… where $X, Y \in \mathbb{R}^d$ are vector embeddings of size $d = 512$.

In the case of Gatys-based similarity, hereafter Gatys similarity, where Gram matrices are used in conjunction with vector embeddings to compute the similarity of two images, the similarity is composed of two parts: the content similarity S^{G_C} and style similarity S^{G_S}. The content similarity score is the same as CLIP-based similarity, i.e., it is defined as the cosine similarity of the vector embeddings (which in the cases of VGG19 and ResNet50 is a flattened tensor of feature maps). The style similarity function is dependent on the style representation, of which this thesis explores two: Gram matrices per layer and feature map activation statistics (mean, std) per layer.

In the case of *Gram matrices*, the style similarity system used compares the flattened Gram matrices of each style layer with weighted cosine similarity. That is, the corresponding Gram matrices of each style layer are compared using cosine similarity, that similarity is then multiplied by a weight assigned to that layer and then added to an overall style similarity score.

Equation 17. Gram-based style similarity

$$S_{XY}^{\text{Gram}} = \sum_{i=1}^{L} w_i \left[\frac{1}{N_i} \sum_{j=1}^{N_i} S_{X_{j:}^i Y_{j:}^i} \right]$$

… where X, Y are lists of size L where X^i and Y^i are the Gram matrices of X and Y at layer i respectively. According to this equation, the style similarity of two images then is the sum of the weighted, row-wise averaged cosine similarity of the Gram matrices of the images.

In the case of *feature map activation statistics*, the similarity system used compares the monolithic channel-wise (feature-map-wise) mean and standard deviation vectors for each layer's feature maps as defined in Equation 6. That is, this system uses a style representation where each style layer's feature map activation distribution statistics (mean, standard deviation) are concatenated into a flat vector of size $2N_l$, where N_l is the number of feature maps in layer l. These vectors are then concatenated into a monolithic

representation vector and compared using cosine similarity to represent style similarity. This monolithic approach is possible in the case of activation statistics as the size of the individual layer vectors grows linearly with the number of feature maps ($2N_l$), compared to Gram matrices which grow quadratically with the number of feature maps (N_l^2).

Equation 18. Feature map-wise statistics similarity

$$S_{XY}^{\text{Stat}} = S_{XY}$$

… where X, Y are the flattened and concatenated feature map activation statistics vectors.

The content score for both approaches is the cosine similarity of the flattened embeddings representing the content (output of the adaptive pooling in VGG19 and the output of the last block in ResNet50).

Equation 19. Content Similarity

$$S_{XY}^{\text{Cont}} = S_{XY}$$

… where X, Y are the flattened content representations.

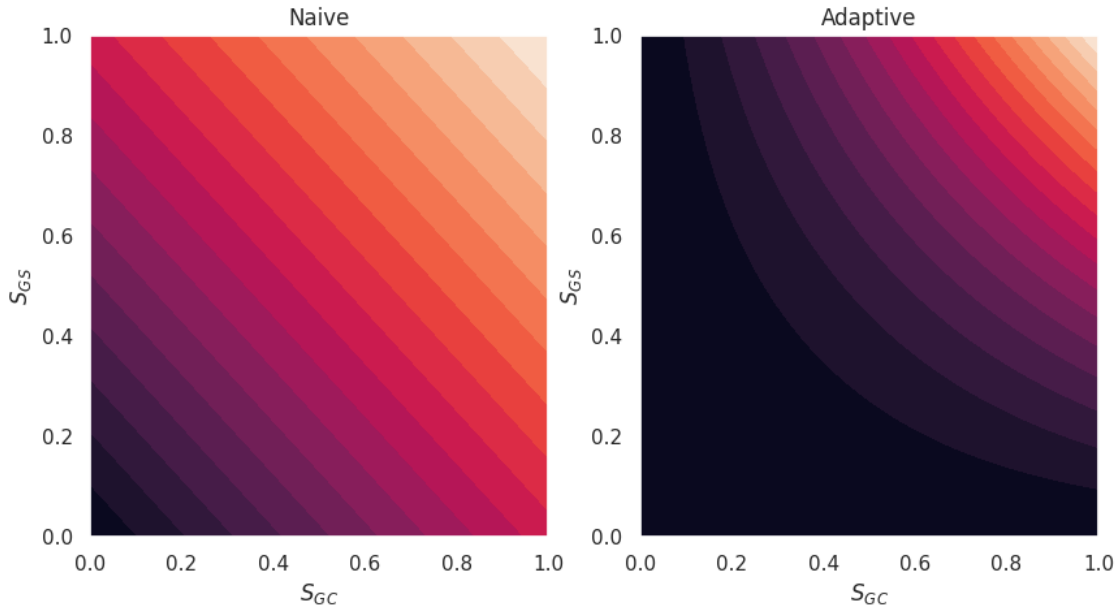
The final similarity score in the Gatys systems is given by the adaptive weighted similarity formulation:

Equation 20. Adaptive Weight Gatys similarity

$$S_{XY}^{\text{Gatys}} = (S_{XY}^{\text{Style}} S_{XY}^{\text{Cont}})(S_{XY}^{\text{Style}} + S_{XY}^{\text{Cont}})$$

… where S_{XY}^{Style} depends on the system used. The reason for this formulation can be seen in Figure 6; a naïve sum of the two similarity components leads to a linear scoring space.

Figure 6. Naive Gatys vs. Adaptive Gatys



Why this is undesirable can be seen in the levels graph (Figure 6); the overall naïve similarity can be easily biased by a large component value, leading to recommendations that are biased in one component or the other. The adaptive approach mitigates this with its scoring space, as can be seen in the levels graph; the score levels are much more tightly grouped around the theoretical max and much wider in the lower levels. This means that recommendations are less biased by component similarity. Another detail of note from Figure 6 is the truncated range of the scoring systems. Although theoretically both systems should have a range of $[-2,2]$, in practice this range is limited to $[0,1]$. This is due to the CNN systems' use of the ReLU activation function which clamps its input to 0 and above. This means that all the extracted embeddings are strictly positive. The dot product and product of norms of any two such vectors will always be positive, thus their cosine similarity will also always be positive, which leaves us with a component similarity range of $[0,1]$.

6.2 Implementation

This section of the thesis presents the pseudocode/algorithmic implementations of all the necessary classes and utility functions to construct and use simple image recommendation systems based upon the theoretical framework presented in the previous sections. As mentioned in section 5.2, pseudocode is used to provide a concise, language-agnostic

overview of the proposed systems without getting bogged down with language- or library-specific notation that may be unintuitive to some. However, Python implementations of each component presented can be found in appendix 2. The first component to be defined is also the one that is re-used the most and forms the functional backbone of all of the implementations: cosine similarity, which can be seen in Pseudocode 1.

Pseudocode 1. Cosine Similarity

Function $S_{\cos}(X, Y)$
  Parameters: X, Y
  Ensure that $X \in \mathbb{R}^{d_X}$, $Y \in \mathbb{R}^{d_Y}$ such that $d_X = d_Y$
  Compute cosine similarity $S \leftarrow \frac{X \cdot Y}{\|X\|_2 \|Y\|_2}$
  Return S

As per the above implementation, any safe implementation of cosine similarity needs to ensure that the compared vectors are of the same dimension.

6.2.1 CLIP Based Implementation

The first highlighted implementation is that of the CLIP-based embedder, which can be seen in Pseudocode 2. As CLIP serves as a sort of state-of-the-art comparison, its system is the most simplistic one. Using only the pre-trained vision transformer to embed the images in the CLIP space.

Pseudocode 2. CLIP Embedder

Class CLIPEmbedder
  Initialize:
    Load pre-trained CLIP image model M_θ
    Define preprocessing pipeline P
  Call:
    Parameters: I
    Preprocess image $I \leftarrow P(I)$
    Extract embedding $C \leftarrow M_\theta(I) \in \mathbb{R}^{d_c}$
    Return C

Since the CLIP-based system uses only the ViT vector embeddings, its similarity functionality is also very simplistic, using only the cosine similarity of two given embeddings to represent their abstract similarity. An implementation of this similarity is seen in

Pseudocode 3. Also note that this definition uses the cosine similarity function as defined by **Error! Reference source not found.**; the pseudocode implementations presented in this thesis can easily and directly access functions and objects defined in the other pseudocode blocks. This is another benefit of using pseudocode; it allows for looser reusability rules.

Pseudocode 3. CLIP Similarity

Function $S_{\text{CLIP}}(X, Y)$
  Parameters: X, Y
  Compute cosine similarity of embeddings $S \leftarrow S_{\cos}(X, Y)$
  Return S

The embedding class and function are then used to define a recommendation function, which takes in a target image T, and a collection/dataset of candidate images C_D to return a collection of the sorted indices of the candidate dataset. This can be seen in Pseudocode 4.

Pseudocode 4. CLIP Recommendation Function

Function $R_{\text{CLIP}}(T, C_D)$
  Parameters: T, C_D
  Load CLIPEmbedder $E \leftarrow$ CLIPEmbedder
  Embed target image $T_E \leftarrow E(T)$
  Initialize list of similarities $S_D \leftarrow [\cdot]$
  for each C in C_D do:
    Embed candidate image $C_E \leftarrow E(C)$
    Compute CLIP similarity of embeddings $S \leftarrow S_{\text{CLIP}}(T_E, C_E)$
    Append S to S_D
  end for
  Arg. Sort similarity list (in descending order) $I_D \leftarrow \text{argsort}(S_D)$
  Return I_D, S_D

These are sorted such that the index in the first position points to the candidate most similar to the target and the last index points to the candidate least similar to the target.

6.2.2 Gatys Based Implementation

The second implementation highlighted is that of the Gatys models; that is, the models that make use of the VGG19 and ResNet50 convolutional feature extractors to represent content and texture/style, named so after the first author of the original neural style transfer paper. This necessitates the definition of both the VGG19 and ResNet50 based embedders, which can be seen in Pseudocode 5 and

Pseudocode 6.

Pseudocode 5. VGG19 Embedder

Class VGG19Embedder
  Initialize:
    Load pre-trained VGG19 model M_θ
    Define content embedding layer c
    Define layers for feature extraction: $L = \{l_1, l_2, l_3, l_4, l_5\}$
    Define preprocessing pipeline P
  Call:
    Parameters: $I, stats$
    Preprocess image $I \leftarrow P(I)$
    Extract content embedding $C \leftarrow M_{\theta c}(I) \in \mathbb{R}^{n \times d_c}$
    for each $l \in L$ **do**:
      Extract layer activations $F_l \leftarrow M_{\theta l}(I) \in \mathbb{R}^{n \times d_l}$
      if $stats$ is "Grams":
        Compute Gram matrix $S_l \leftarrow F_l F_l^\top \in \mathbb{R}^{n \times n}$
      if $stats$ is "mean-std":
        Compute vector $S_l \leftarrow [\mu(F_{l1:}), \sigma(F_{l1:}), ..., \mu(F_{ln:}), \sigma(F_{ln:})]^\top \in \mathbb{R}^{2n}$
      end if
    end for
    Return C, S_L

From the algorithms, it is obvious that the main difference between the implementations is the chosen model and the number of layers. However, this is only the case in the pseudocode abstractions, as the Python implementations of each differ quite a bit due to

architectural differences. Thus, to match the Python implementations and to avoid cluttered control-flow blocks as much as possible, they are shown as two different pseudocodes.

Pseudocode 6. ResNet50 Embedder

Class ResNet50Embedder
  Initialize:
    Load pre-trained ResNet50 model M_θ
    Define content embedding layer c
    Define layers for feature extraction: $L = \{l_1, l_2, l_3, l_4\}$
    Define preprocessing pipeline P
  Call:
    Parameters: $I, stats$
    Preprocess image $I \leftarrow P(I)$
    Extract content embedding $C \leftarrow M_{\theta c}(I) \in \mathbb{R}^{n \times d_c}$
    for each $l \in L$ **do**:
      Extract layer activations $F_l \leftarrow M_{\theta l}(I) \in \mathbb{R}^{n \times d_l}$
      if $stats$ is "Grams":
        Compute Gram matrix $S_l \leftarrow F_l F_l^\top \in \mathbb{R}^{n \times n}$
      if $stats$ is "mean-std":
        Compute vector $S_l \leftarrow [\mu(F_{l1:}), \sigma(F_{l1:}), ..., \mu(F_{ln:}), \sigma(F_{ln:})]^\top \in \mathbb{R}^{2n}$
      end if
    end for
    Return C, S_L

As mentioned in the Content and Style section, two representations of style are considered: Gram matrices and channel-wise activation statistics. These are presented in the pseudocode by the options “Grams” and "mean-std". As the shape of the representation differs in each case, two differing similarity and recommendation functions are designed. In the case of Gram matrix representations of style, style similarity is computed as the layer-wise weighted average cosine similarity of the corresponding, flattened Gram matrices. In this configuration, the set of weights of the style layers is a hyperparameter that may be tuned based on the context of the image space. Whereas, in the case of activation statistics, the flat vectors of feature map activation statistics of every layer are concatenated into one vector, which are then compared using cosine similarity. In both cases, content similarity between two images is just a simple case of the cosine similarity of their flattened content representations.

These component similarity scores are then combined using the function defined by Equation 20. These implementations can be seen in

Pseudocode 7 and

Pseudocode 8.

Pseudocode 7. Mean-STD Similarity

Function $S_{\text{Mean-STD}}(X, S_X, Y, S_Y)$
  Parameters: X, S_X, Y, S_Y
  Define flattening function $F : \mathbb{R}^{n \times d_c} \rightarrow \mathbb{R}^{nd_c}$
  Flatten X content embedding $X \leftarrow F(X)$
  Flatten Y content embedding $Y \leftarrow F(Y)$
  Compute content similarity $S_C \leftarrow S_{\cos}(X, Y)$
  Concatenate all X style vectors $S_X \leftarrow \text{concat}(S_X)$
  Concatenate all Y style vectors $S_Y \leftarrow \text{concat}(S_Y)$
  Compute style similarity $S_S \leftarrow S_{\cos}(S_X, S_Y)$
  Compute similarity score $S \leftarrow (S_C S_S)(S_C + S_S)$
  Return S

Unlike the Grams based similarity function, the Mean-STD similarity function provides no weights to tune. This is due to the fact that the style representation is a singular vector instead of it being divided between layers.

Pseudocode 8. Grams Similarity

Function $S_{\text{Grams}}(X, S_X, Y, S_Y, W)$
  Parameters: X, S_X, Y, S_Y, W
  Define number of style layers $n_L \leftarrow |W|$
  Define layer index set $L = \{1, ...n_L\}$
  Define flattening function $F : \mathbb{R}^{n \times d_c} \rightarrow \mathbb{R}^{nd_c}$
  Flatten X content embedding $X \leftarrow F(X)$
  Flatten Y content embedding $Y \leftarrow F(Y)$
  Compute content similarity $S_C \leftarrow S_{\cos}(X, Y)$
  Define initial style similarity $S_S \leftarrow 0$
  for each l in L:
    Define l-layer Gram flattening function $K : \mathbb{R}^{n_l \times n_l} \rightarrow \mathbb{R}^{n_l^2}$
    Flatten S_{Xl} Gram $G_{Xl} \leftarrow K(S_{Xl})$
    Flatten S_{Yl} Gram $G_{Yl} \leftarrow K(S_{Yl})$
    Compute layer similarity $S_{Sl} \leftarrow S_{\cos}(G_{Xl}, G_{Yl})$
    Weigh S_{Sl} and update overall style similarity $S_S \leftarrow S_S + W_l(S_{Sl})$
  end for
  Compute similarity score $S \leftarrow (S_C S_S)(S_C + S_S)$
  Return S

This, in turn, affords less flexibility when it comes to tuning the performance of the system, but it also ensures much smaller style representations, cutting back on the time and space costs. Using these components, two recommendation functions are created: one for Gram-

based recommendation and one for Mean-STD-based recommendation. These can be seen in Pseudocode 9 and Pseudocode 10.

Pseudocode 9. Grams Recommendation

Function $R_{\text{Grams}}(T, C_D, W)$
  Parameters: T, C_D, W
  Load Gatys embedding model $E \leftarrow$ GatysEmbedder
  Embed target image $T_E, T_S \leftarrow E(T, \text{"Grams"})$
  Initialize list of similarities $S_D \leftarrow [\cdot]$
  for each C in C_D do:
    Embed candidate image $C_E, C_S \leftarrow E(C, \text{"Grams"})$
    Compute Grams similarity $S \leftarrow S_{\text{Grams}}(T_E, T_S, C_E, C_S, W)$
    Append S to S_D
  end for
  Arg. Sort similarity list (in descending order) $I_D \leftarrow \text{argsort}(S_D)$
  Return I_D, S_D

These recommendation functions are implemented similarly to the CLIP recommendation function with a few noticeable differences. Namely, the embedding model being loaded and the use of content and style embeddings.

Pseudocode 10. Mean-STD Recommendation

Function $R_{\text{Mean-STD}}(T, C_D)$
  Parameters: T, C_D
  Load Gatys embedding model $E \leftarrow$ GatysEmbedder
  Embed target image $T_E, T_S \leftarrow E(T, \text{"mean-std"})$
  Initialize list of similarities $S_D \leftarrow [\cdot]$
  for each C in C_D do:
    Embed candidate image $C_E, C_S \leftarrow E(C, \text{"mean-std"})$
    Compute MSTD similarity $S \leftarrow S_{\text{Mean-STD}}(T_E, T_S, C_E, C_S)$
    Append S to S_D
  end for
  Arg. Sort similarity list (in descending order) $I_D \leftarrow \text{argsort}(S_D)$
  Return I_D, S_D

In the pseudocode, GatysEmbedder is just an alias for either of the CNN-based models defined earlier. This was done in an effort to again reduce the number of control-flow statements in the pseudocode.

7 Results

This section goes over the results of the evaluations described in section 5.4 on the systems described in the previous section. After, it presents some highlighted results from the top performing CNN based model, both using its Grams configuration and its Mean-STD configuration which is meant to represent a more space efficient approach. These evaluations are then interpreted and expanded with contextual information. Finally, a few suggestions for future improvements and deeper research are presented.

7.1 Evaluation Results

The first evaluation case to be considered is the quantitative evaluation, as defined in section 5.4.1. The results of this evaluation can be seen in Table 1.

Table 1. Evaluation Results

500-subset of ArtCap	Avg. Caption Cos. Sim. @5*	Avg. HOG Dist. @5**	Avg. Color Hist. Corr. @5*	**Rank**
vgg19-e	0.16	21.35	0.22	5
resnet50-e	0.14	21.28	0.25	6
vgg19-eg-eq	0.16	20.97	0.35	2
resnet50-eg-eq	0.15	21.05	0.3	4
vgg19-emstd	0.15	20.61	**0.37**	2
clip	**0.24**	**20.51**	0.32	**1**

The following sections go through the results presented in Table 1 and interpret them with respect to the performance of the presented models while providing likely explanations for some noted behaviors.

7.1.1 Quantitative Results

In this table, the results of the evaluation for 6 systems are shown. The first two systems, **vgg19-e** and **resnet50-e** represent systems which use only the use of the content embeddings for similarity. These are used as base cases to prove that the addition of textural information increases the overall performance of recommendation systems that use these pre-trained embeddings. Indeed, this performance increase can be seen in the cases of **vgg19-eg-eq** and **resnet50-eg-eq**, which are the systems that utilize the Gatys similarity and Gram textural representations. The **-eq** flag indicates that the layer weights in

this example were left as equal values summing up to 1, essentially computing the average cosine similarity of Grams. From both sets of cases, it can be seen that, in this suite of tasks, the VGG19 systems outperform both of their corresponding ResNet50 systems. With this in mind, the next evaluation is that of the **vgg19-emstd** system, which uses the VGG19 Mean-STD textural representations for its Gatys similarity. Only the VGG19-backed version is tested as VGG19 seems to be a better performer in these tasks and also has the added benefit of producing more efficient embeddings. This system, surprisingly, outperforms its Gram counterpart and even achieves the best performance when it comes to color histogram correlation. Finally, **clip** scores show that it outperforms all other systems in caption similarity and HOG distance, however it is ranked 3rd when it comes to color histogram correlation. Potential reasons for these scores are discussed further below.

7.1.2 Qualitative Results

Figure 7 presents 3 highlighted qualitative examples of the performance of the top 3 models as per their quantitative evaluation.

Figure 7. Highlighted Qualitative Examples

The first thing to note is that in all of the examples, VGG19 w/ Grams and VGG19 w/ MSTD seem to be in agreement, at least for the top result. The quantitative results then imply that

they diverge in their rankings beyond the most similar image. However, this section is only focused on their picks for most similar.

The first row shows an interesting disparity in “definition” of similarity between the VGG19-based system and the CLIP based system, namely in that the former focuses more on the scenic composition and the latter focuses more on the content of the image. That is, VGG19 features seemingly show a similarity in the existence and relationship of objects within an image (here “person under a gateway”) whereas CLIP is more focused on matching the content and its description (here “woman in flowing dress posing with head to the side”).

The second row shows a similar disparity in a more extreme manner. In this case, it is evident that the VGG-based systems have focused more on the texture/distribution of color more than the content of the image, whereas CLIP has focused solely on the objective content of the image.

Finally, the third row shows all three systems being in agreement and finding an image with both similar texture and content to the target. These 3 examples were hand-picked to show the 3 extremes of these systems, for an uncurated list of examples see appendix 3.

7.1.3 Discussion on Results

This section presents a few probable explanations for the results seen in both the quantitative and qualitative evaluation results presented above. Firstly, it should be noted that neither evaluation is meant to be exhaustive. This is both due to time constraints but also because of complexity constraints. For example, the performance of the Gram based systems will most likely fluctuate intensely with changes to the layer weights. Prioritizing later weights will most likely increase their performance on content matching but may hurt their performance on texture matching, and vice-versa. However, the number of possible combinations of such weights is too large to be tested within the scope of this thesis. Another consideration is that of the qualitative approach; the performance in this task is severely dependent on the distribution of “similar” images within the candidate pool. That is to say, if there is no image that is “similar” to the target within the candidate pool then the system simply cannot present any similar images. This is another constraint that is beyond the scope of this thesis.

As for interpreting the results, both evaluations, imperfect as they may be, present a good look into the strengths and weaknesses of the highlighted systems as well as insight into the way these systems “understand” images and how this is influenced by their training and architecture. Firstly, it makes complete sense for CLIP to outperform all other models when it comes to matching captions, as this is somewhat similar to how CLIP was trained. Indeed, one can argue that this proficiency with coupling the mediums of image and text is also the reason as to why, in the visual examples, CLIP seems to mainly match images by “content description”. This is in line with how images are usually described by their captions; instead of focusing on the scenic composition, captions generally tend to describe the content of the image. Seeing as CLIP was trained on such image-caption pairs, this behavior is most likely a property of its multi-modal embedding space. The inverse can be said for CNN based models which, in extreme cases, seem to be heavily biased by texture.

Another thing to consider when it comes to the performance of these systems is their expressive prowess. While this is dependent on many different properties of the backbone model, the main thing to highlight is the number of trainable parameters. The CNN-based feature extractors used in this thesis possess about **20 million** trainable parameters whereas the smallest vision transformer provided by CLIP has around **80 million** parameters.

7.2 Future Improvements

The field of abstract image understanding is an interesting and broad field of computer vision which has recently taken a massive leap forward with the advancements of machine learning and computer vision. As such, there is a vast amount of area to explore in this field, and the research presented in this thesis is no different. This can be achieved both by further refining the systems and evaluations presented in this thesis and by introducing new ones based on more novel research.

A simple expansion that has high potential for good performance is a hybrid system. This would be a system that uses the semantically rich embeddings of CLIP with the efficient and texturally inclined embeddings of VGG19-MSTD. It would be interesting to see if combining the two best performing systems would lead to a better performing system or to a system with a degraded performance and analyze why either might occur.

Another possible expansion of the systems presented here is through the use of metric learning. Metric learning is the learning paradigm in machine learning which aims to learn a distance/similarity metric from data. In the case of this system, such a model can be trained to create an embedding space that mimics the distances of the Gatys systems presented in this thesis, essentially learning a space that couples texture and content.

Finally, it would also be interesting to explore whether increasing the textural diversity of the pre-training task of models like CLIP would lead to embeddings that encode both textural and content similarity. This would also allow for testing the performance of such embeddings in down-stream tasks, which might provide some indication as to the importance of textural information to computer vision tasks.

8 Conclusion

This thesis set out to answer the following research questions:

1. Can style transfer-based representations be used to compare images?
2. Can style transfer-based representations be used to formulate coherent and viable image recommendation/retrieval systems?
3. How do CNN-based systems compare to natural language-supervised systems like CLIP?

Arguably, the answers to all of these questions, based on the results of the quantitative and qualitative evaluations is yes. The representations for content and style presented by style transfer research show to be good ways to compare images along these components. As such, these systems can be used to formulate efficient and effective image recommendation systems that take into account content and style. Finally, there is a marked difference between these style transfer, CNN-based systems and CLIP. From the data presented in this thesis, it seems that CNN-based systems can be much more biased by the texture of an image, whereas CLIP based systems are much more easily biased by the content of images.

These findings, along with serving as the foundation for strong, abstract image recommendation systems, explain in depth the way these models “see” and “understand” images, which is a very important insight into the effects of different training paradigms, architectures, and the types of data a model is exposed to during training. This information

can be gleaned from analyzing what a system sees as similar pairs of images. This sort of information not only helps researchers and developers understand the cause and effect of the properties of models but may also be an additional tool in guiding more intuitive understandings of these models.

Overall, it can be safely said that this thesis achieved what it set out to do. It not only presented powerful image recommendation models built atop of strong theoretical bases, it also provided intuitive insight into the models and systems it used and defined.

References

Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). *Layer Normalization* (arXiv:1607.06450). arXiv. http://arxiv.org/abs/1607.06450

Bahdanau, D., Cho, K., & Bengio, Y. (2016). *Neural Machine Translation by Jointly Learning to Align and Translate* (arXiv:1409.0473). arXiv. https://doi.org/10.48550/arXiv.1409.0473

Bradski, G. (2008). The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, *25.11*, 120–125.

Brauwers, G., & Frasincar, F. (2023). A General Survey on Attention Mechanisms in Deep Learning. *IEEE Transactions on Knowledge and Data Engineering*, *35*(4), 3279–3298. https://doi.org/10.1109/TKDE.2021.3126456

Ciresan, D., Meier, U., Masci, J., Gambardella, L. M., & Schmidhuber, J. (2011). *Flexible, High Performance Convolutional Neural Networks for Image Classification.* 1237–1242. https://doi.org/10.5591/978-1-57735-516-8/IJCAI11-210

Dalal, N., & Triggs, B. (2005). Histograms of Oriented Gradients for Human Detection. *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, *1*, 886–893. https://doi.org/10.1109/CVPR.2005.177

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., & Houlsby, N. (2021). *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale* (arXiv:2010.11929). arXiv. https://doi.org/10.48550/arXiv.2010.11929

Gatys, L. A., Ecker, A. S., & Bethge, M. (2015a). *A Neural Algorithm of Artistic Style* (arXiv:1508.06576). arXiv. http://arxiv.org/abs/1508.06576

Gatys, L. A., Ecker, A. S., & Bethge, M. (2015b). *Texture Synthesis Using Convolutional Neural Networks* (arXiv:1505.07376). arXiv. https://doi.org/10.48550/arXiv.1505.07376

Gehring, J., Auli, M., Grangier, D., Yarats, D., & Dauphin, Y. N. (2017). *Convolutional Sequence to Sequence Learning* (arXiv:1705.03122). arXiv. https://doi.org/10.48550/arXiv.1705.03122

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. The MIT press.

Goodfellow, I. J., Bulatov, Y., Ibarz, J., Arnoud, S., & Shet, V. (2014). *Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks* (arXiv:1312.6082). arXiv. https://doi.org/10.48550/arXiv.1312.6082

Han, J., & Kamber, M. (2012). *Data mining: Concepts and techniques* (3rd ed). Elsevier.

Hastie, T., Tibshirani, R., & Friedman, J. H. (2009). *The elements of statistical learning: Data mining, inference, and prediction* (2nd ed). Springer.

He, K., & Sun, J. (2014). *Convolutional Neural Networks at Constrained Time Cost* (arXiv:1412.1710). arXiv. http://arxiv.org/abs/1412.1710

He, K., Zhang, X., Ren, S., & Sun, J. (2015). *Deep Residual Learning for Image Recognition* (arXiv:1512.03385). arXiv. https://doi.org/10.48550/arXiv.1512.03385

Huang, X., & Belongie, S. (2017). *Arbitrary Style Transfer in Real-time with Adaptive Instance Normalization* (arXiv:1703.06868). arXiv. https://doi.org/10.48550/arXiv.1703.06868

IBM. (n.d.). *What Is Machine Learning (ML)?* | *IBM*. Retrieved March 29, 2024, from https://www.ibm.com/topics/machine-learning

James, G., Witten, D., Hastie, T., Tibshirani, R., & Taylor, J. E. (2023). *An introduction to statistical learning: With applications in Python*. Springer.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems*, *25*. https://proceedings.neurips.cc/paper_files/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989). Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, *1*(4), 541–551. Neural Computation. https://doi.org/10.1162/neco.1989.1.4.541

Le-Khac, P. H., Healy, G., & Smeaton, A. F. (2020). Contrastive Representation Learning: A Framework and Review. *IEEE Access*, *8*, 193907–193934. https://doi.org/10.1109/ACCESS.2020.3031549

Li, Y., Wang, N., Liu, J., & Hou, X. (2017). *Demystifying Neural Style Transfer* (arXiv:1701.01036). arXiv. https://doi.org/10.48550/arXiv.1701.01036

Lu, Y., Guo, C., Dai, X., & Wang, F.-Y. (2024). ArtCap: A Dataset for Image Captioning of Fine Art Paintings. *IEEE Transactions on Computational Social Systems*, *11*(1), 576–587. https://doi.org/10.1109/TCSS.2022.3223539

Miller, J. (2007). *Keeping a Research Journal for Your Scholarly Project*. https://www.utoledo.edu/library/info/dir/JoleneMillerKeepingResearchJournal.pdf

Mitchell, T. M. (1980). *The Need for Biases in Learning Generalizations*.

Mitchell, T. M. (1997). *Machine learning* (Nachdr.). McGraw-Hill.

Mohri, M., Rostamizadeh, A., & Talwalkar, A. (2018). *Foundations of machine learning* (Second edition). The MIT Press.

Obsidian. (n.d.). *Obsidian—Sharpen your thinking*. Retrieved September 1, 2024, from https://obsidian.md/

OpenCV. (n.d.). *OpenCV: Histogram Comparison*. Retrieved February 27, 2025, from https://docs.opencv.org/3.4/d8/dc8/tutorial_histogram_comparison.html

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., … Chintala, S. (2019). *PyTorch: An Imperative Style, High-Performance Deep Learning Library* (arXiv:1912.01703). arXiv. https://doi.org/10.48550/arXiv.1912.01703

Phuong, M., & Hutter, M. (2022). *Formal Algorithms for Transformers* (arXiv:2207.09238). arXiv. https://doi.org/10.48550/arXiv.2207.09238

Pyne, Y., & Stewart, S. (2022). Meta-work: How we research is as important as what we research. *The British Journal of General Practice*, *72*(716), 130–131. https://doi.org/10.3399/bjgp22X718757

PyTorch. (n.d.). *Conv2d—PyTorch 2.3 documentation*. Retrieved April 25, 2024, from https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html

Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., Krueger, G., & Sutskever, I. (2021). *Learning Transferable Visual Models From Natural Language Supervision* (arXiv:2103.00020). arXiv. https://doi.org/10.48550/arXiv.2103.00020

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). *Language Models are Unsupervised Multitask Learners*.

Reimers, N., & Gurevych, I. (2019). *Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks* (arXiv:1908.10084). arXiv. https://doi.org/10.48550/arXiv.1908.10084

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., & Fei-Fei, L. (2015). *ImageNet Large Scale Visual Recognition Challenge* (arXiv:1409.0575). arXiv. https://doi.org/10.48550/arXiv.1409.0575

Simonyan, K., & Zisserman, A. (2015). *Very Deep Convolutional Networks for Large-Scale Image Recognition* (arXiv:1409.1556). arXiv. https://doi.org/10.48550/arXiv.1409.1556

TechTarget. (n.d.). *What is a proof of concept (POC)? – TechTarget Definition*. CIO. Retrieved September 1, 2024, from https://www.techtarget.com/searchcio/definition/proof-of-concept-POC

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). *Attention Is All You Need* (arXiv:1706.03762). arXiv. https://doi.org/10.48550/arXiv.1706.03762

Zotero. (n.d.). *Zotero | Your personal research assistant*. Retrieved September 1, 2024, from https://www.zotero.org/

Appendix 1. Material Management Plan

The main materials used during this thesis process comprise of the cited works used to formulate the theoretical framework presented in the first part of the thesis, the research journal kept which details and sketches out the step from theory to practice and how the two parts are related, the Python implementations of the various systems presented in the second part and in Appendix 2, and the data used for the quantitative and qualitative evaluations. All of these materials are subject to the same data retention management plan: they are to be stored on the author's local drive along with back ups being kept in the Google Drive cloud solution. Beyond this, for the code and data used during the evaluation, links are provided below. These images have only been used for research purposes and this distribution implicitly carries the same license as the original distributions (Pexels: https://www.pexels.com/license/, ArtCap: https://github.com/luttie2022/ArtCap-Dataset).

No sensitive user or respondent information was used in the formulation of this thesis and as such no extra data security precautions (beyond the cloud back-ups being shared via the link only) are taken. The links to the datasets are as follows:

1. ArtCap: https://drive.google.com/drive/folders/1KYNWHBmsXSVbavObso0SEPudE4d81jHi?usp=sharing
2. Pexels: https://drive.google.com/drive/folders/1DAT1xA0BNkAHgy01A-x91-Wc-rQ2gWpt?usp=sharing

The Pexels repository also includes a .csv file containing the appropriate attributions for each image in the set.

Appendix 2: Python Implementations

This appendix presents a mapping of the pseudocode presented in this thesis to simple Python implementations used to facilitate the presented evaluations. All of this code can be seen and accessed here: https://drive.google.com/drive/folders/1U3H6M0coVwASn_2GbdYaU2XXZ4MCCcvQ?usp=sharing

Cosine Similarity:

```
def _np_cosine_similarity(
        v1: ArrayLike,
        v2: ArrayLike
) -> float:
    if len(v1.shape) > 1:
        v1 = v1.reshape(-1)
        v2 = v2.reshape(-1)
    dot = np.dot(v1, v2)

    v1_norm = np.linalg.norm(v1)
    v2_norm = np.linalg.norm(v2)

    return dot/(v1_norm * v2_norm)
```

CLIP Similarity:

```
def clip_similarity(
        X: ArrayLike,
        Y: ArrayLike,
) -> float:
    return _np_cosine_similarity(X, Y)
```

Gram Similarity:

```
def gram_similarity(
        X: ArrayLike,
        S_X: Dict[str, ArrayLike],
        Y: ArrayLike,
        S_Y: Dict[str, ArrayLike],
        layer_weights: Dict[str, float]
) -> float:

    content_similarity = _np_cosine_similarity(X, Y)
    style_similarity = 0
    for layer in layer_weights:
        style_similarity += layer_weights[layer] * _np_cosine_similarity(S_X[layer],
S_Y[layer])

    return (content_similarity * style_similarity) * (content_similarity +
style_similarity)
```

Mean-STD Similarity:

```
def mean_std_similarity(
        X: ArrayLike,
        S_X: Dict[str, ArrayLike],
        Y: ArrayLike,
        S_Y: Dict[str, ArrayLike],
) -> float:
    content_similarity = _np_cosine_similarity(X, Y)
    S_X_flat = []
    S_Y_flat = []
    for layer in S_X:
        S_X_flat.append(S_X[layer])
        S_Y_flat.append(S_Y[layer])

    S_X_flat = np.concatenate(S_X_flat, axis=-1)
    S_Y_flat = np.concatenate(S_Y_flat, axis=-1)

    style_similarity = _np_cosine_similarity(S_X_flat, S_Y_flat)
    return (content_similarity * style_similarity) * (content_similarity +
style_similarity)
```

CLIP Recommendation:

```
def recommend_clip(
        target_name: str,
        candidate_names: ArrayLike,
        embedding_path: str,
):
    T_c = load_image_embeddings(target_name, embedding_path, None)
    scores = []
    for candidate_name in candidate_names:
        C_c = load_image_embeddings(candidate_name, embedding_path, None)
        scores.append(clip_similarity(T_c, C_c))
    scores = np.argsort(scores)[::-1]
    return scores
```

Gatys Recommendation:

```
def recommend_gatys(
        target_name: str,
        candidate_names: ArrayLike,
        embedding_path: str,
        style_path: str,
        weights: Dict[str, float] = None,
        stats: str = "grams",
):
    T_c, S_c = load_image_embeddings(target_name, embedding_path, style_path)
    scores = []
    for candidate_name in candidate_names:
        C_c, S_c = load_image_embeddings(candidate_name, embedding_path, style_path)
        if stats == "grams":
            scores.append(gram_similarity(T_c, S_c, C_c, S_c, weights))
        elif stats == "mean-std":
            scores.append(mean_std_similarity(T_c, S_c, C_c, S_c))
        else:
            raise ValueError(f"Stats {stats} not supported.")
    scores = np.argsort(scores)[::-1] # sort in descending order
    return scores
```

VGG19 Embedder:

```
class VGG19Embedder(nn.Module):
    def __init__(self, *args, **kwargs):
        """
        Gatys embedder based on the VGG19 architecture.
        """
        super().__init__(*args, **kwargs)
        vgg = vgg19(weights=VGG19_Weights.IMAGENET1K_V1)
        vgg.eval()
        self.features = vgg.features
        self.avgpool = vgg.avgpool
        self.preprocess = Compose([
            Resize((224, 224)),
            ToTensor(), # scales to 0 - 1 range
            Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]), #
ImageNet normalization
        ])
        self.device = 'cpu'
        targets = {
            "conv1_1": 1, # relu1_1
            "conv2_1": 6, # relu2_1
            "conv3_1": 11, # relu3_1
            "conv4_1": 20, # relu4_1
            "conv5_1": 29, # relu5_1
        }
        self.activations = {}

        for layer, idx in targets.items():
          self.features[idx].register_forward_hook(self._get_activations_hook(layer))
# registering hooks to capture activations at target feature layers

    def _get_activations_hook(self, layer_name):
        def hook(module, input, output):
            self.activations[layer_name] = output.view(output.size(0), -1)

        return hook
    def to(self, device: str):
        self.device = device
        self.features = self.features.to(device)
        self.avgpool = self.avgpool.to(device)
        return self

    def forward(
            self,
            image: Image.Image,
            stats: str="grams"
    ) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
        image_tensor = self.preprocess(image).to(self.device)

        with torch.no_grad():
            features_out = self.features(image_tensor)
            content_embedding = self.avgpool(features_out)
        content_embedding = content_embedding.view(-1)

        if stats == "grams":
            grams = {}
            for layer in self.activations:
                grams[layer] = torch.matmul(self.activations[layer],
torch.transpose(self.activations[layer], 0, 1))
            return content_embedding, grams
        elif stats == "mean-std":
            meanstd = {}
```

ResNet50 Embedder:

```
class ResNet50Embedder(nn.Module):
    def __init__(self, *args, **kwargs):
        # here we use the entire model instead of submodules due to the fact that the
ResNet50 architecture is more sub-modular than the VGG19
        super().__init__(*args, **kwargs)
        self.resnet = resnet50(weights=ResNet50_Weights.IMAGENET1K_V2)
        self.resnet.eval()
        self.preprocess = Compose([
            Resize((224, 224)),
            ToTensor(), # scales to 0 - 1 range
            Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]), #
ImageNet normalization
        ])
        self.device = 'cpu'
        target_layers = {
            "block1": "layer1",
            "block2": "layer2",
            "block3": "layer3",
            "block4": "layer4",
            "content": "avgpool",
        }
        self.activations = {}
        for layer, n in target_layers.items():
            l = getattr(self.resnet, n)
            l.register_forward_hook(self._get_activations_hook(layer))

    def _get_activations_hook(self, layer_name):
        def hook(module, input, output):
            output = output.squeeze(0)
            self.activations[layer_name] = output.view(output.size(0), -1)
        return hook

    def to(self, device: str):
        self.device = device
        self.resnet = self.resnet.to(device)
        return self

    def forward(
            self,
            image: Image.Image,
            stats: str="grams"
    ) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
        image_tensor = self.preprocess(image).to(self.device)
        if len(image_tensor.size()) < 4:
            image_tensor = image_tensor.unsqueeze(0)
        with torch.no_grad():
            self.resnet.eval()
        if stats == "grams":
            grams = {}
            for layer in self.activations:
                if layer != "content":
                    grams[layer] = torch.matmul(self.activations[layer],
torch.transpose(self.activations[layer], 0, 1))
            return self.activations["content"].view(-1), grams
        elif stats == "mean-std":
            meanstd = {}
            for layer in self.activations:
                if layer != "content":
                    meanstd[layer] = torch.cat([torch.mean(self.activations[layer],
dim=-1), torch.std(self.activations[layer], dim=-1)], dim=-1)
            return self.activations["content"].view(-1), meanstd
```

Appendix 3: Uncurated Examples