Aflutter Craft: Neural Art Transfer Platform

Rawad Abdulghafor

Faculty of Computer Studies (FCS), Arab Open University, Muscat P.O. Box 1596, Oman
Email: rawad.a@aou.edu.om

***Abstract*—Image Style transfer is a neural network algorithm that copies the style of an existing image into another image while preserving the image's content. There have been various approaches on style transfer in an effort to speed up the process or provide more appealing results, one of which is the usage of style attentional networks. Attention is an algorithm that scores different parts of an image based on their importance in the overall image, attention helps neural networks distinguish important parts of an image. We use attention to identify the parts of an image that represent image style to apply an overall style rather than a mask and to conserve parts of the content that are crucial to its identity (a visible object, a focused subject, etc.). Aflutter Craft enhances an existing algorithm that uses attention for style transfer. Results show that our algorithm uniformly applies important parts of the style while simultaneously preserving the subject of the content image. Results from Aflutter Craft are chosen to be the most visually appealing according to 38.4% survey participants when compared to 4 other implementations. In addition, this paper introduces a cross platform application with a general Application Programming Interface (API) capable of performing style transfer from anywhere.**

***Keywords*—Application Programming Interface (API), Art, convolutional neural network, deep learning, flask, neural network, PyTorch, self-attention, style transfer, transfer learning, transformer**

I. INTRODUCTION

Making art is not an easy task, not everyone is capable of being an artist and producing art that resonates with the masses; classic artists such as Picasso and Michelangelo are regarded as the greatest by current artists and their works are sold at absurdly high prices usually in millions of dollars. Currently, there are no artists with the ability to replicate astonishing classical art styles down to an individual artist's unique brush. Computers have come a long way and are currently capable of performing feats beyond what's humanly possible, it is possible to teach a neural network to replicate classical art styles and more into any image, that process is known as style transfer.

Fig. 1 showcases a basic example of style transfer, where the content of the original image is transformed using the artistic style of another image. The transformation highlights the model's ability to apply complex artistic patterns while preserving key content features, such as the main subject's shape and structure.

Manuscript received August 7, 2024; revised August 14, 2024; accepted August 27, 2024; published November 18, 2024.

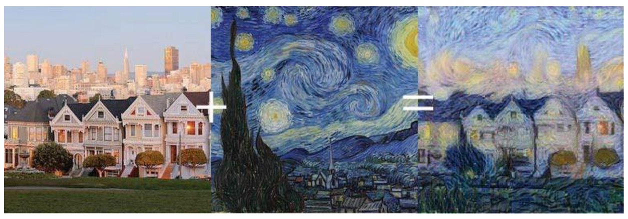


Fig. 1. Style transfer example [1].

Aflutter craft makes the process of replicating classical art styles easier not just for professional artists but for everyone. Presently, to make an image appear in a style from a specific classical artist, one needs to be fluent in Photoshop along with other image editing tools, making it harder for everyone. Some websites can apply an image's style to another [2]; however, they are limited when it comes to the selection of provided styles, not to mention the lack of native mobile or desktop applications capable of performing this task.

Aflutter craft style store consists of the entirety of wikiart images, a total of over 40,000 images, giving users a huge collection of art styles to choose from with the ability to upload any image as style. In addition to the ability to customize the trade-off between content and style images, with the availability of mobile, desktop and web applications, applying a style to images has never been more straightforward.

II. LITERATURE REVIEW

Neural Style transfer being a recent algorithm has numerous works uncovering the mechanics behind why it works and different iterations improving on the algorithm, from improvements to the time it takes to stylize an image, to more complex algorithms that are capable of not style transfer while accounting for other factors in the image such as depth and color preservation.

Gatys *et al.* [3] proposed using a Convolutional Neural Network (CNN) trained for object detection to extract image features (style and content) from specific layers of that network, then feed the images to a style transfer network to generate a new image by minimizing the mean-squared distance between the entries of the Gram matrices from the original image and the Gram matrices of the image to be generated. The used CNN is VGG-19 trained on a labeled image dataset of 15 million photos, the paper notes higher layers such as conv4_2 have a fair knowledge of the image content, and lower layers maintain less content, the style is present in multiple higher layers ending with 1 such as conv4_1 and conv1_1. Issues with the proposed architecture include:

- The time taken for the synthesis procedure correlates linearly with the image size.
- synthesized images are subject to white noise (occurs in photo-realistic scenarios).
- separation of image content from style is not a well-defined problem.

This diagram in Fig. 2 illustrates the architecture of the original neural style transfer algorithm proposed by Gatys *et al.* [3]. It demonstrates how the algorithm uses a Convolutional Neural Network (CNN) to extract and blend the content and style features from different images. The key components of the architecture, including content and style layers, are labeled to show the flow of data through the network.

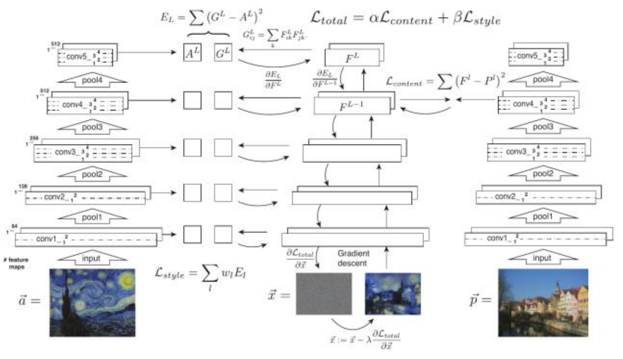

$E_L = \sum (G^L - A^L)^2$
$\mathcal{L}_{total} = \alpha \mathcal{L}_{content} + \beta \mathcal{L}_{style}$
$G^L_{ij} = \sum_k F^L_{ik} F^L_{jk}$
A^L
G^L
F^L
$\frac{\partial E_L}{\partial F^L}$
$\frac{\partial E_L}{\partial F^{L-1}}$
$\mathcal{L}_{content} = \sum (F^l - P^l)^2$
F^{L-1}
conv5_1
conv4_1
conv3_1
conv2_1
conv1_1
pool4
pool3
pool2
pool1
input
# feature maps
512
256
128
64
$\mathcal{L}_{style} = \sum_l w_l E_l$
$\frac{\partial \mathcal{L}_{total}}{\partial \vec{x}}$
Gradient descent
$\vec{a} =$
$\vec{x} =$
$\vec{p} =$
$\vec{x} := \vec{x} - \lambda \frac{\partial \mathcal{L}_{total}}{\partial \vec{x}}$

Fig. 2. Original neural style transfer algorithm [3].

Li *et al.* [4] argued that style transfer is a problem of domain adaptation, the gram matrix used to optimize the losses to generate the stylized image is similar to the process of Maximum Mean Discrepancy (MMD) with the second-order polynomial kernel, meaning the style information is represented by a distribution of activations in the CNN and with the process of distribution alignment style transfer is achievable, thus replacing the gram matrix with an MMD kernel should perform the style transfer successfully, different kernels functions are experimented with such as Gaussian kernel, making the fusion of two different kernels possible and producing different looking images.

Johnson *et al.* in [5] suggested using perceptual loss from a pre-trained loss network. The perceptual loss function can calculate the difference between two images' high-level features which works by summing all the squared errors of all pixels and then taking the mean compared to per-pixel loss which only sums the errors between pixels. The results are a much faster network up to 1060x times faster in inference compared to per-pixel loss using image size of 256 × 256, this speed up also opens the door for real-time video style transfer which is demonstrated at 20fps running on a Pascal Titan X.

In the work [6] perceptual factors such as space, color, and scale have been controlled for better results, for space, the goal is to specify which region of the content image is stylized by which region of the style image, with this a single image can contain style from multiple sources for example the sky can have a different style compared to the ground or buildings, the second factor to control is color, for instance, using a style without applying its colors to the content image which helps in preserving the color of the content. The final controlled factor is the scale of the image, this technique removes the content loss from a style image to replace the full image texture enabling the creation of high-resolution images with combined styles.

Ulyanov *et al.* [7] proclaims that using perceptual losses despite being much faster results in less appealing images compared to the original paper with per-pixel loss, it then suggests replacing batch normalization with instance normalization applied at both training and testing based on an observation by Ulyanov *et al.* [7] stating "the network generates poor quality images if trained with a large dataset". The key difference between instance and batch normalization is that the latter normalizes over a whole batch of images instead of just one image leading to one image with abnormal mean or standard deviation affecting the entire batch while the former normalizes each image separately and directly eliminates issues found in batch normalization. This results in a network that is just as fast but produces more appealing results according to Ulyanov *et al.* [7].

In the previous works the style applied to all parts of the image equally with no regard to image depth resulting in the images losing their sense of depth. Lu *et al.* [8] have proposed using a depth prediction network for estimating the depth of the image and accordingly stylize the image. The network architecture is based on the work by Johnson *et al*. [5] with the addition of depth estimation network. Optimizing the content loss through the depth estimation network gives the depth loss, the final optimized parameter is the total loss produced from the addition of the style and content loss-produced by the object recognition loss network and the depth loss. The result is images that look more realistic, with more distinguishable objects and a noticeable sense of depth.

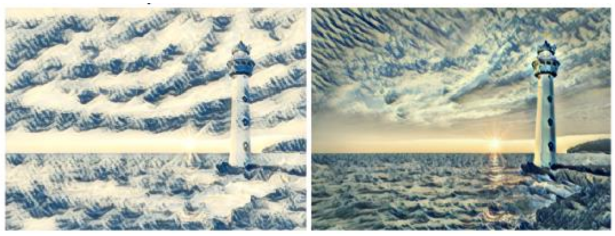


Fig. 3. Results without vs with depth awareness [8].

The Fig. 3 compares the output of style transfer models with and without depth awareness. The images on the left show results from a model that does not account for image depth, leading to flattened visual effects. The right-side images incorporate depth awareness, resulting in more realistic and dimensionally accurate stylized images, especially in scenes with varying depths like landscapes.

Previous works have depended on a pre-trained VGG-19 network for content and style extraction, however, the VGG network is trained for object recognition and therefore cannot accurately interpret styles.

Sanakoyeu *et al.* [9] has proposed training a VGG-16 network on art style images from wikiart and using content images from a scene recognition dataset for training the style transfer network for which it proposes a new architecture. The proposed network uses an encoder decoder feed-forward architecture in contrast to the optimization-based approach in previous papers. In addition, the new architecture makes use of a discriminator to compare the stylized image to the original content image to preserve details as much as possible while maintaining artistic accuracy. The network allows quick inference (0.6 s) besides real-time HD video stylization.

This Fig. 4 depicts the architecture of a style-aware content loss network. It emphasizes how the network preserves content structure while applying artistic styles. The figure includes annotations showing the flow from the encoder to the decoder, highlighting the interaction between style and content layers that optimize the balance between style fidelity and content preservation.

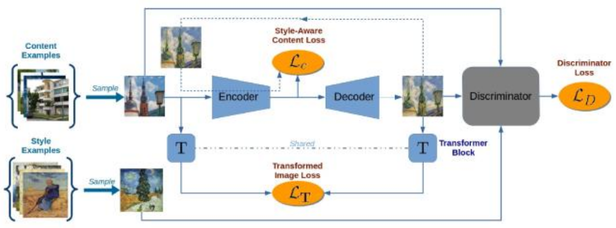


Fig. 4. Style-aware content loss network architecture [9].

A novel normalization method referred to as Adaptive instance normalization (AdaIN) has been proposed in [10]. AdaIN behavior is observed to work like a style loss. It works by adjusting the mean and variance of the content image to that of the style image and thus performing style transfer, this works arbitrarily for any style, the used architecture is a feed-forward encoder-decoder architecture with the AdaIN layer at its heart. The result is quick style transfer (0.6 s), allowing adjusting parameters such as color and content-style trade-off during inference without training the model again with those new parameters.

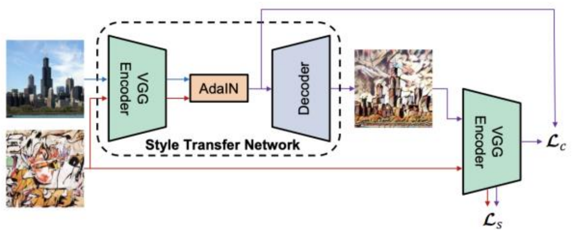


Fig. 5. Network architecture with AdaIN [10].

The diagram in Fig. 5 explains the architecture of the network using Adaptive Instance Normalization (AdaIN). It illustrates how AdaIN adjusts the mean and variance of the content image to match those of the style image, enabling arbitrary style transfer. The key layers and operations are clearly labeled, demonstrating how this architecture achieves real-time style transfer with high efficiency.

Chen *et al.* [11] proposed an alternative method for applying style transfer using an arbitrary number of styles. The proposed approach makes use of a feed-foreword CNN with a convolution-deconvolution architecture. The way it works is both the content and style images will pass through the convolutional network (VGG network) to extract high-level features and then pass through a style swap layer, the style swap layer concatenates the style and content images by swapping the texture of a content image with that of the style image using randomly selected patches (usually of size 3 × 3) from the style image, after that the decoder approximates the stylized image and its loss. The simplicity of the approach allows generalization between styles and style-content trade-off by changing the patch size besides fast style transfer by requiring fewer iterations.

This Fig. 6 shows the patch-based network architecture used for style transfer. It explains how the network swaps patches between content and style images to generate the final stylized output. The use of small, randomly selected patches allows for a more flexible and detailed application of styles across various content types.

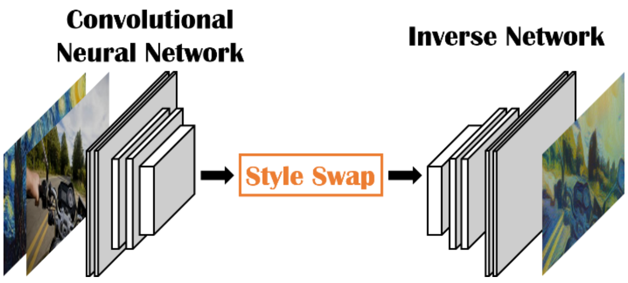


Fig. 6. Patch-based network architecture [11].

Most of the previously reviewed works depend solely on CNNs and optimizations techniques to synthesize the final image. Park *et al.* [12] has proposed using self-attentional networks (SANet) which allow matching the nearest style features to the content image structure, a new loss function referred to as identity loss is used, it works as a metric for evaluating generated image. After feature extraction from the VGG-19 network the output passes through SANet to give style and content feature maps priority and then according to that priority the synthesized image is reconstructed by the decoder. The image then again passes through an encoder to calculate the losses. The result of such a complex network is more efficient and quicker style transfer (comparable to AdaIN) that transfers different styles for each semantic content and maintains high content detail, beside the ability to adjust content-style trade-off in real-time, it also allows the mixing of multiple styles. The authors conducted a survey to compare their results with other previous algorithms, and 34.5% of the 80 people surveyed preferred the result of SANet.

This Fig. 7 contrasts the architectures of two networks: one focusing on identity loss and the other on self-attention. The left side highlights the role of identity loss in maintaining content integrity during style transfer, while the right side shows how self-attention mechanisms selectively apply styles to important regions of the content image. The side-by-side comparison underscores the advantages of integrating both methods for improved stylization.

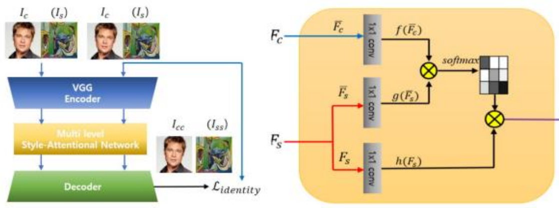


Fig. 7. Identity loss network (left), Self-Attention network (right) [12].

The previous works illustrate the progress made in the field of style transfer through the years; the method used has considerably changed across papers and iterations. Each paper reviewed has had a noticeable contribution ranging from a simpler, more efficient network architecture to allowing the usage of a single network to apply any style regardless of the network being trained on it or not. For a better understanding of the changes proposed by each paper, we encourage the reader to dive deeper and read the papers. The overall goal of all the reviewed papers remains the same; to produce a neural network that is capable of replicating any art style to any content image without significantly changing the image's focus or subject.

The study of researches [1, 13] compares learning algorithms for image classification with small datasets, aiming to identify the most effective approach. We fine-tuned hyperparameters for optimizers and models, conducting initial experiments with eight algorithms to approximate optimal values. Final experiments with near-optimal hyperparameters revealed that the AdaGrad learning algorithm outperforms others, achieving higher accuracy, shorter training times, and lower memory usage.

Deep learning is revolutionizing disease diagnosis in radiology, yet acquiring large, detailed labelled datasets remains a challenge. Transfer learning mitigates this by starting with pre-trained weights from a large, similar dataset and fine-tuning them on a smaller, specific dataset. Iqbal *et al.* [14] uses deep learning to detect synovial fluid in human knee joints from Magnetic Resonance Imaging (MRI) images, proposing a specialized convolutional neural network for automated detection. Trained and evaluated on two independent datasets, the model achieves high sensitivity, specificity, precision, and accuracy, offering a novel, efficient method for synovial fluid analysis.

III. Materials and Methods

The three key components of Aflutter Craft are model development, API development, and Mobile Application Development. The employed development methodology is the waterfall model, due to each component requiring the one prior to it; for instance, one cannot start building the API before the model is complete; subsequently, this applies to major parts of the mobile application as well.

The waterfall model diagram in Fig. 8 outlines the step-by-step approach used in the development of the Aflutter Craft platform. It shows the sequence of stages from model development to API and mobile application development, emphasizing the dependencies between each stage and the importance of completing one phase before moving on to the next.

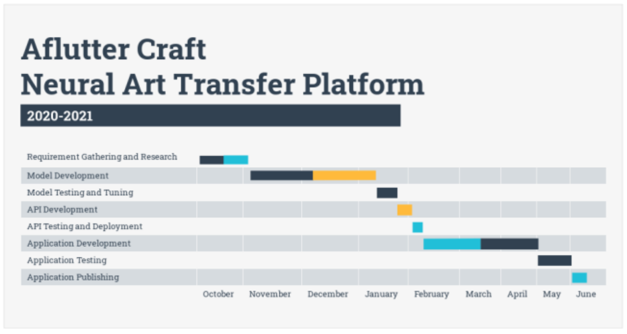


Fig. 8. Project waterfall model.

A. Design

The model architecture used in our approach is the one proposed by Park *et al.* [15] with the addition of a fixed identity loss coefficients (representing style and content loss weights). The reason behind the architecture choice is that it's a modern implementation taking advantage of new technologies such as attention, another important factor is that it allows relatively fast inference which is a requirement in a mobile application. The deep learning framework of choice is PyTorch [16] due to its ease of usage.

This Fig. 9 presents the overall network architecture used in the Aflutter Craft platform. It includes details on the encoder, attention modules, and decoder, along with the flow of data between these components. The diagram also highlights the use of fixed identity loss coefficients, which help preserve content structure while applying styles.

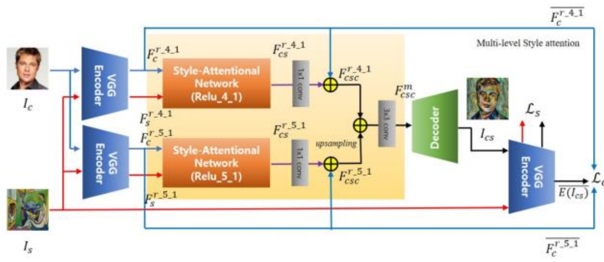


Fig. 9. Project network architecture [12].

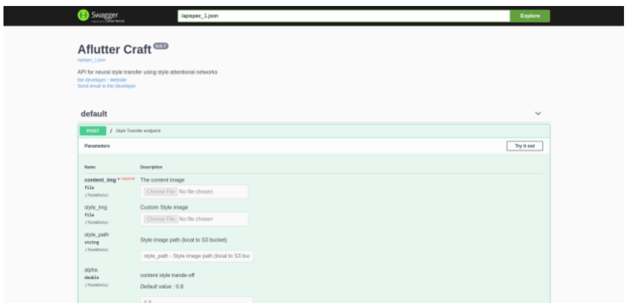
Identity loss is the metric used to keep the content of the original image, it is calculated from the same input image unlike the style and content loss which are calculated across the entire batch of images, as a result, the identity loss makes it possible to simultaneously preserve the content image structure while applying characteristics from the style image.

$$L_{identity} = \lambda_{identtiy1}(||(I_{cc} - I_c)||_2 + ||(I_{ss} - I_s)||_2) + \lambda_{identtiy2} \sum_{i=1}^{L}(||\phi_i(I_{cc}) - \phi(I_{cc})||_2 + ||\phi_i(I_{ss}) - \phi(I_s)||_2) \quad (1)$$

where I_{cc} (or I_{ss}) denotes the output image synthesized from two same content (or style) images, each φ_i denotes a layer in the encoder, $\lambda_{identity1}$ and $\lambda_{identity2}$ are identity loss weights. By fixing $\lambda_{identity1}$ and $\lambda_{identity2}$, we limit it's value to ranges that are observed to be smaller than what's calculated by the network, which leads to the network prioritizing the content image structure over the style image characteristics.

The API is designed using the python web framework Flask [17] with flasgger [18] for openAPI compatible documentation and local testing, the API request is of type POST with the content image and style image ID or a custom style image encoded as base64. When it comes to ease of access, an online server is a requirement, currently the API is deployed to FloydHub [19] allowing inference from anywhere as long as the gateway is available. With the alpha parameter it is easy for users to adjust the style-content trade-off whether the goal is preserving more of the content or applying the style to its fullest. After performing various tests and a survey with different alpha values ranging from 0.1 (least amount of style) to 1.0 (most amount of style) with increments of 0.1, the value selected as default is 0.6 or 60% of the style. The reason for choosing 0.6 was that it had the highest survey responses at 19.4% when compared to values from 0.1 to 1.0.

The Fig. 10 showcases the API documentation and testing interface used in Aflutter Craft. It includes sections displaying the available endpoints, sample requests and responses, and the testing environment. The layout is designed to be user-friendly, facilitating easy integration and testing of the API across various platforms.


Swagger
/apispec_1.json Explore

Aflutter Craft 0.0.7
/apispec_1.json
API for neural style transfer using style attentional networks
the developer - Website
Send email to the developer

default
POST / Style Transfer endpoint
Parameters Try it out

Name	Description
content_img * required file (formData)	The content image Choose File No file chosen
style_img file (formData)	Custom Style image Choose File No file chosen
style_path string (formData)	Style image path (local to S3 bucket) style_path - Style image path (local to S3 buc
alpha double (formData)	content style trande-off *Default value* : 0.8 0.8

Fig. 10. API documentation and testing.

The final component is the user interface. Both the desktop and mobile applications will share similar design attributes and work similarly. The consensus is to have three main screens in the mobile application each with a pre-defined purpose. First the main page showing the currently selected style and content images and a slider for adjusting the alpha value. Another screen for selecting the style and the final screen shows the results with the option to save the image or share it to social media. When selecting the style, the user will be able to select an image from the gallery or take a new picture from either the front or back camera. The Desktop UI on the other hand will have a main page that shows a grid of content, style and stylized images respectively with the alpha slider displayed below the images. Selecting a style will open the style store. Another addition to the desktop app is an about page.

This Fig. 11 illustrates the prototype of the mobile application interface for Aflutter Craft. It features the main screens of the application, including style and content selection, alpha adjustment, and the final result display. The design prioritizes user experience, ensuring intuitive navigation and easy access to key features.

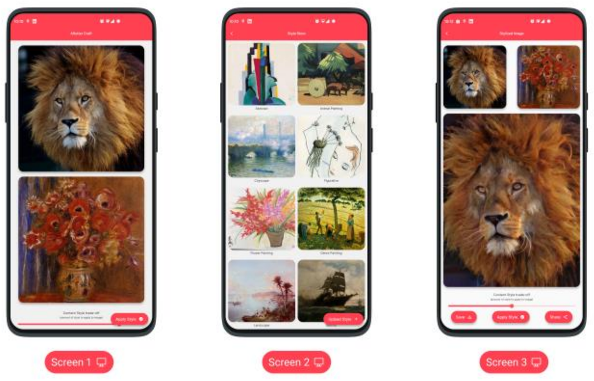


Fig. 11. Mobile application interface prototype.

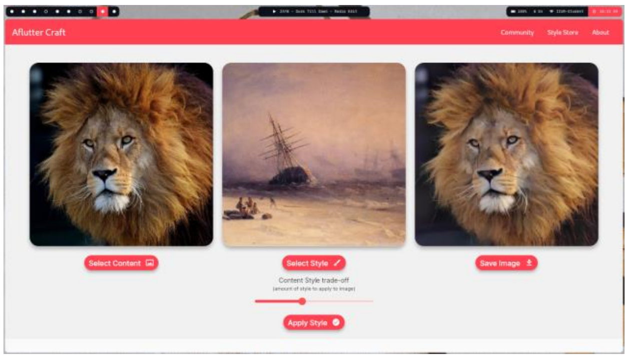


Fig. 12. Main desktop application interface.

The Fig. 12 displays the main interface of the desktop application. It includes a grid layout that shows content, style, and stylized images, with an alpha slider below the images for adjusting the style-content balance. The interface is designed for efficiency, allowing users to quickly select styles and view results.

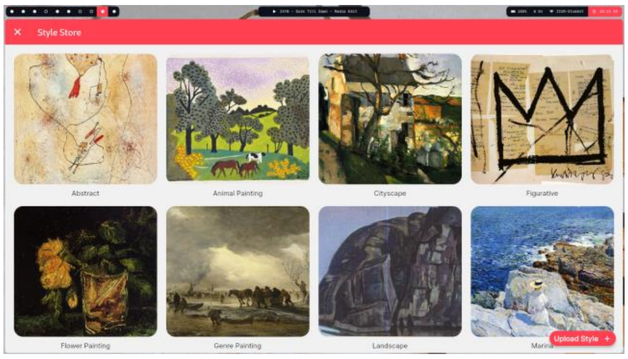
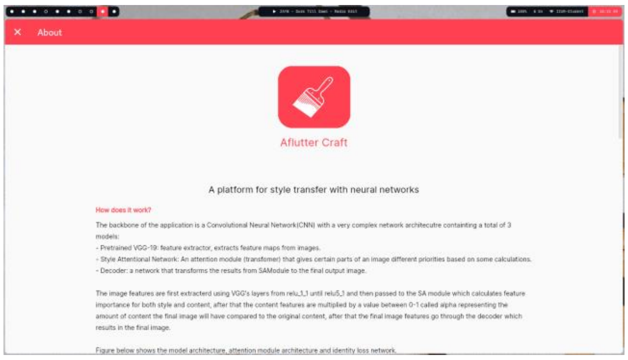


Fig. 13. Desktop app style category selection with option to upload a local image.

This Fig. 13 shows the style selection screen of the desktop application, where users can choose from various style categories or upload a custom style image. The interface is designed to be straightforward, providing users with a wide range of artistic options while ensuring easy customization.



A platform for style transfer with neural networks

How does it work?

The backbone of the application is a Convolutional Neural Network(CNN) with a very complex network architecutre containting a total of 3 models:
- Pretrained VGG-19: feature extractor, extracts feature maps from images.
- Style Attentional Network: An attention module (transfomer) that gives certain parts of an image different priorities based on some calculations.
- Decoder: a network that transforms the results from SAModule to the final output image.

The image features are first extracterd using VGG's layers from relu_1_1 until relu5_1 and then passed to the SA module which calculates feature importance for both style and content, after that the content features are multiplied by a value between 0-1 called alpha representing the amount of content the final image will have compared to the original content, after that the final image features go through the decoder which results in the final image.

Figure below shows the model architecture, attention module architecture and identity loss network.

Fig. 14. Desktop application about page.

The about page of the desktop application in Fig. 14 provides detailed information about the Aflutter Craft platform, including its capabilities and underlying technology. It also links to the open-source repository and other resources, encouraging community engagement and contribution.

1) Neural network architecture and attention mechanism enhancements

a) Neural network architecture

The core of our neural network architecture is based on a modified version of the Style-Attentional Network (SANet), which incorporates advanced attention mechanisms to improve the fidelity and control of style transfer. The architecture consists of three primary components: the encoder, the attention module, and the decoder.

- Encoder
  - The encoder is derived from the VGG-19 network, pre-trained on the ImageNet dataset. This choice is motivated by VGG-19's ability to effectively extract hierarchical features from images, which are crucial for both content preservation and style representation.
  - The encoder takes in the content and style images, passing them through multiple convolutional layers. We utilize the feature maps from specific layers (such as relu4_1) to capture different levels of abstraction, which represent the content structure and style features of the images.
- Attention Module:
  - The attention module is the core enhancement in our architecture. Traditional style transfer models often apply style uniformly across the image, which can lead to the loss of important content details. Our attention mechanism mitigates this by focusing on the most salient regions of the content image.
  - Self-Attention Mechanism: We implemented a self-attention mechanism that allows the model to dynamically assign weights to different regions of the content image based on their importance. The attention mechanism computes a relevance score for each pixel, which determines how much influence the corresponding style feature should have on that pixel.
  - Attention Weights: These weights are calculated using a combination of query, key, and value matrices, which are derived from the feature maps of the content image. The query and key matrices are used to calculate the attention scores, while the value matrix adjusts the feature maps based on these scores. This ensures that regions with higher attention scores retain more of the original content characteristics.
  - Enhancements: Unlike standard self-attention mechanisms, our approach introduces an identity loss to further preserve the content structure. This loss is calculated by comparing the original content image with the stylized output, encouraging the network to maintain the integrity of key content features.
- Decoder:
  - The decoder reconstructs the final stylized image from the attention-weighted feature maps. It consists of a series of transposed convolutional layers, which gradually upsample the feature maps back to the original resolution of the content image.
  - The decoder is designed to balance the integration of style features with the preservation of content structure, guided by the attention module's outputs.

b) Enhancements to the attention mechanisms

The primary enhancement in our model is the integration of identity loss with the attention mechanism. This combination addresses the common issue of content degradation in style transfer by ensuring that the stylized image remains faithful to the original content image.

- Identity Loss Integration:
  - Identity loss is calculated using the original content image as both the input and the target. This encourages the network to prioritize content preservation, especially in regions identified as critical by the attention mechanism.
  - By fixing the weights of style and content during the calculation of identity loss, we ensure that the content structure is given priority over stylistic elements. This is particularly important in scenarios where maintaining the recognizability of the content is essential, such as in portraits or other images with distinct subjects.
- Real-Time Adjustability:
  - The architecture also supports real-time adjustments of the style-content trade-off. Users can control the degree to which style is applied by modifying the alpha parameter, which scales the influence of the attention weights. This feature is particularly useful in applications where user preference plays a significant role in the final output.

2) *Visual representations*

Flowcharts and Diagrams: To aid in the understanding of this architecture, we have included flowcharts that illustrate the data flow from the encoder through the attention module and into the decoder. Additionally, we provide diagrams showing how attention weights are computed and applied to the feature maps, as well as the interaction between the attention mechanism and the identity loss function.

B. *Implementation*

The model is implemented using the PyTorch framework [16]. The API implementation is using the Flask [7] web framework and can be self-hosted anywhere, however, due to domain computational resources requirements (GPUs or accelerators) the chosen cloud provider is FloydHub [19] providing high performance GPUs at reasonable prices. The mobile and desktop applications will make use of the Flutter framework [20].

Flutter is the best choice for a cross platform application. Flutter allows writing code that runs on all major platforms in one language using one centralized code base. Integrating the API with the application is an easy task with flutter, there is support for network images and requests sending/parsing built-in.

Below workflow applies to both mobile and desktop applications, it also applies for both cases when a user uploads a custom style image or selects an image from the style store.

General application workflow:

- User selects the content image from the gallery or takes a picture.
- User selects a style image from the style store or selects a custom image from the gallery as the style image.
- Send API request with content image and selected style image ID or custom style image.
- If the API request contains a style image ID. use that as style, otherwise use the user passed style image.
- pass both content and style images to the model.
- return stylized image in base64 encoding as the API response.

This workflow diagram in Fig. 15 outlines the steps involved in using the Aflutter Craft application. It starts with content image selection, followed by style selection, and concludes with the stylized image generation and display. The workflow is designed to be simple and efficient, ensuring a smooth user experience across different platforms.

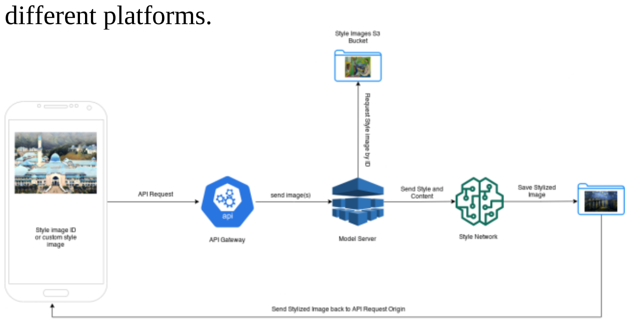


Fig. 15. General application workflow.

C. *Testing*

Platform compatibility testing has been conducted for android, Linux, and the Web, due to the unavailability of other platforms' hardware, it is not possible to test for them, however, with the framework of choice being cross platform they should be working as intended.

Each of the application's three components are tested separately. For model testing a survey is conducted and based on its results further tuning might be required, since the ability to control style-content trade-off (α) is already present, it might be easier to make the survey participants test the model and play with the style-content trade off until they have results they deem desirable.

For the API, the tests will include fetching all the styles from the project S3 bucket, which will be shown in the desktop and mobile applications. The second test will make sure the API can send content and style images to the server correctly, and return a stylized image successfully. Finally, the application tests will include testing the overall functionally from fetching styles and stylizing to the save functionality and others, the application testing will progressively test the other two modules in a backward fashion.

TABLE I. MODEL TESTING

Test Case ID	Objective	Input	Expected Results	Procedure
TC-01-01	Model results evaluation	100 Stylized images	favorable rating by users	conduct survey

As shown in Table I, model testing focuses on evaluating the results of 100 stylized images, with the objective of achieving favorable ratings from users. Given the subjective nature of visual appeal, user preferences were gathered through a survey. The results indicate that most users preferred having the ability to manually select a style and adjust the alpha value. However, 19.4% of the 95 survey participants favored the results with an alpha value of 0.6, and 38.4% of participants found Aflutter Craft to be more visually appealing compared to other algorithms.

The Fig. 16 presents the results of a user survey comparing the visual appeal of images generated by five different style transfer algorithms. The survey results, displayed as a bar chart, indicate user preferences for the Aflutter Craft implementation, highlighting its effectiveness in producing visually appealing stylized images.

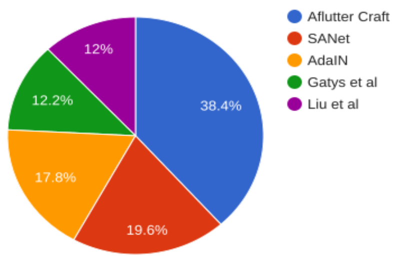


Fig. 16. Survey results comparing 5 different algorithms.

This Fig. 17 shows the distribution of user preferences for different alpha values, which control the style-content balance in the stylized images. The chart illustrates how varying the alpha value affects the final output, with the majority of users favoring a value of 0.6 for its optimal balance between content preservation and style application.

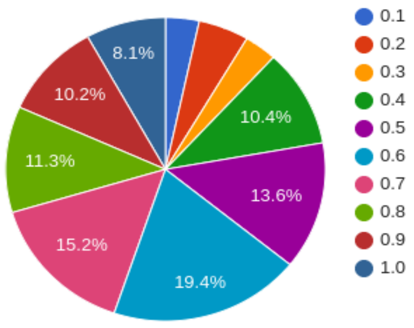


Fig. 17. User preference of alpha value.

TABLE II. API TESTING

Test Case ID	Objective	Input	Expected Results	Procedure
TC-02-01	API evaluation	content image and style ID or custom image	return stylized image	send style image and content image or ID

Table II outlines the API evaluation process, where the Aflutter Craft API successfully returns a stylized image when provided with a content image and a style ID or custom image. This process ensures that the API functions properly, delivering the desired output from the custom styles or predefined images stored in the project's S3 bucket.

Aflutter Craft API successfully returns a stylized image when given a content image with a custom style image or an ID of an image present in the project S3 bucket.

TABLE III. API STYLE IMAGES TESTING

Test Case ID	Objective	Input	Expected Results	Procedure
TC-02-02	Style images fetching	API call	return all style images every time	request style images API

Table III presents the testing results for style image fetching. The system reliably retrieves and displays all available style images via API calls, confirming that the images are fetched successfully and displayed in the application using a REST API call to the S3 bucket.

Style Images are successfully fetched and shown in the application using a REST API call to the S3 bucket.

Additionally, Table IV highlights the overall application functionality testing. This table shows that the application enables users to select content images from the gallery or camera, submit stylization requests, and receive the resulting images from the API. Furthermore, the application can successfully save these images to the gallery or share them on social media as per user preference.

TABLE IV. APPLICATION TESTING

Test Case ID	Objective	Input	Expected Results	Procedure
TC-03-01	Application evaluation	image selection and stylization request	images are successfully sent to API, and results are successfully returned	select images from application and send stylization request

Overall application functionality works as intended, the style images are successfully shown to the user, the user is able to select a content image from the gallery or take a new one from the camera, the application successfully sends requests to the API and receives and displays the resulting image successfully. The application can successfully save the image locally to the gallery or share it to social media if selected by the user.

D. Model Training

Training the model requires a vast amount of computational resources and unlike the training of a regular machine learning model, the number of epochs is not very important. The used metric for training is the number of iterations where an iteration is defined as the generation of a single stylized image after passing a style and content images through the full model architecture, we used the metric of 160,000 iterations as coined by Huang and Belongie [10], after at least 160,000 iterations the model is successfully able to generate images, however, the optimal amount of iterations that generates the best results is practically unknown thus we trained the model for a total of 500,000 iterations leading to the results shown below, comparing model results when trained for 160,000 iterations and when trained for 500,000 iterations, shows that a higher number of iterations leads to better results.

The online platform Kaggle [21] was used to train the model due to the availability of high-performance GPUs and abundance of storage. The training time for a total of 500,000 iterations is 68.75 hours using a Nvidia Tesla P100 while the least amount of time needed for a working model is only 22 hours (for a total of 160,000 iterations). The data used to train the model is a combination of 2 datasets with Painters by number [22] used as style images containing a total of over 79,000 images and Microsoft's COCO used as content images [23] with over 328,000 images. All the images are resized to 512×512 and then randomly cropped to 256×256 (standard size used across most papers).

Table V shows hyperparameters used to train the model. The learning rate choice is directly inferred from [12] same for the learning rate decay, batch size is set to 5 due to limited resources availability, higher numbers will require more computational power, computational power is the reason behind going with defaults or common values. Using a learning rate decay decreases the learning rate as the model trains leading to faster network convergence [24].

TABLE V. MODEL HYPERPARAMETERS

HyperParameter	Value
Learning Rate	1e-4
Learning Rate decay	5e-5
batch size	5

E. Application Development

Given the large number of style images, it will not be ideal to have all of them as an option in the application. One of the main concerns when building a network intensive application, such as this one, is the resource usage. From fetching the style images to sending them to the API and downloading the results, all these operations depend on the user's network connection's strength. To make the application accessible on relatively poor network connections we need to minimize the network resources usage, below are the steps followed to minimize the network usage:

- From the painters by numbers dataset containing 14 different styles of painting (abstract, religious paintings, etc.) only select 100 random images from each style category.
- Save the selected images' names in a file with the category name and ship them with the app binary as assets (each of size 3.8 kb).
- When the user launches the style store, the application will load these files and generate a random index number that will be used to fetch the image of that index and use it as a category cover.
- The fetched cover image will also be cached for the current application session.
- When the user selects a specific category, only the images visible on the screen will be downloaded. For example, only 8 images will be downloaded on desktop initially, as the user scrolls new images will be loaded asynchronously.
- The images downloaded from a category will be cached in the device even after closing the app, thus each image is downloaded only once per app lifetime. The images will only be deleted if the app is deleted.
- On the API side, when the request contains a style image ID instead of a style image, it means the image is hosted on the shared project S3 bucket.

The API will first check if the image has already been downloaded locally before downloading it, thus each image is only downloaded once to the API server.

F. Training Process and Parameter Optimization

The training process for our model was carefully designed to balance the trade-offs between computational efficiency, model performance, and the quality of the stylized images. This section provides a detailed explanation of how the model parameters were selected and optimized, as well as the challenges encountered during the training process.

1) Selection of model parameters

- Learning Rate:
  - The learning rate is a critical hyperparameter that determines the step size at each iteration while moving toward a minimum of the loss function. We initially set the learning rate to $1 \times 10 - 41 \times 10^{-4}$ $1 \times 10 - 4$ based on recommendations from previous studies and the nature of our model's architecture. This value was selected to ensure stable convergence without overshooting the optimal solution.
  - To further fine-tune the learning process, a learning rate decay was introduced, decreasing the learning rate by $5 \times 10 - 55 \times 10^{-5}$ $5 \times 10 - 5$ after each epoch. This approach helps in converging more quickly during the early stages of training while allowing finer adjustments in later stages.
- Batch Size:
  - A batch size of 5 was chosen for training. This value represents a compromise between computational resource constraints and the need for sufficient data diversity within each batch to ensure robust training. Smaller batch sizes were found to lead to noisy gradients, while larger sizes required significantly more memory, limiting the ability to train on high-resolution images.
- Optimizer:
  - The Adam optimizer was selected due to its adaptive learning rate capabilities, which are well-suited for training deep neural networks. Adam combines the advantages of both the AdaGrad and RMSProp optimizers, providing fast convergence and reducing the risk of getting stuck in local minima.
- Number of Iterations:
  - The model was trained for a total of 500,000 iterations. This number was determined through experimentation, where we observed that models trained for fewer iterations (e.g., 160,000 iterations) showed suboptimal stylization results, particularly in maintaining the integrity of the content image. The extended training duration allowed the model to better capture the intricate balance between style and content, leading to superior visual quality.

2) Optimization techniques

- Loss Function Balancing:
  - The total loss function was composed of three components: content loss, style loss, and identity loss. Each of these components was weighted to achieve the desired balance between preserving the content image and applying the style.
  - Content Loss: Initially set with a higher weight to ensure the structure of the content image was preserved. As training progressed, this weight was gradually reduced to allow the style features to emerge more prominently.
  - Style Loss: Weighting was adjusted based on the complexity of the style images. For styles with intricate patterns, a higher weight was applied to ensure these details were captured effectively.
  - Identity Loss: This was introduced to maintain the essential features of the content image, particularly in cases where excessive stylization could obscure important details.

- Early Stopping and Checkpointing:
    - To prevent overfitting, early stopping was implemented based on validation loss. If the validation loss did not improve for a specified number of epochs, training was halted. Checkpoints were also saved at regular intervals, allowing us to revert to a model with the best performance if overfitting was detected.
- Data Augmentation:
    - Data augmentation techniques, such as random cropping, flipping, and scaling, were applied to the training images. This not only increased the diversity of the training data but also helped the model generalize better to unseen images during inference.

3) Challenges and solutions

- Computational Constraints:
    - One of the main challenges encountered was the high computational cost associated with training the model, especially for large-scale datasets. To address this, we utilized cloud-based platforms with GPU support, such as Kaggle, which provided the necessary computational resources to handle the extensive training required.
- Balancing Style and Content:
    - Another challenge was achieving the right balance between style transfer and content preservation. This required careful tuning of the loss function weights and multiple iterations of trial and error. The introduction of identity loss and the ability to adjust the alpha parameter during inference were key innovations that helped in addressing this challenge.
- Training Stability:
    - Ensuring stable training across such a large number of iterations posed its own challenges, particularly in avoiding vanishing or exploding gradients. The use of the Adam optimizer and a carefully selected learning rate schedule were instrumental in maintaining training stability.

IV. Result and Discussion

In this task there is no loss or accuracy and each implementation could focus on a different objective. For instance, Liu *et al.* [8] uses depth as a metric while [15] uses the time it takes to produce an image as a metric , as outlined in [12] the time it takes to generate an image is comparable to that of Ref. [10] with the difference being mainly visual, we have opted for a survey to evaluate the results of our implementation, as shown in Fig. 16, most survey takers preferred the results from Aflutter Craft when compared to other implementations including Refs. [12] and [10].

Fig. 18 shows a comparison between the results of Aflutter Craft, in Refs. [1, 2, 5, 15] for the same content image and style image. The results from Aflutter Craft are the most visually appealing according to survey participants, the results from Refs. [1] and [5] are the most similar to the results from Aflutter Craft, results from the original algorithm in [2] seem to be the least visually appealing, it can be observed that the initial algorithm presented in [2] applies the style indiscriminately to the content image making it look like an overlay rather than a style transfer.

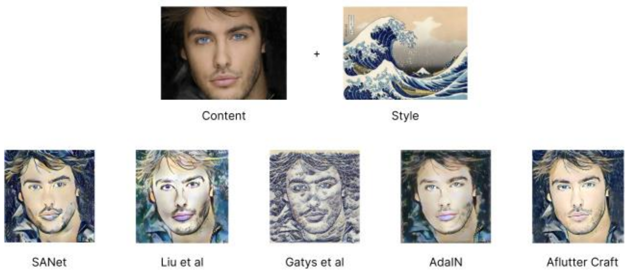


Fig. 18. left: results without fixed identity loss coefficients, right: results with fixed coefficients.

The addition of fixed weights for the style and content when calculating the identity loss produces images that have a higher level of detail as demonstrated in Fig. 19, this can be due to the value of the identity loss itself being smaller leading to the model prioritizing the content features more compared to the style features, this however can be tuned during inference using the alpha value.

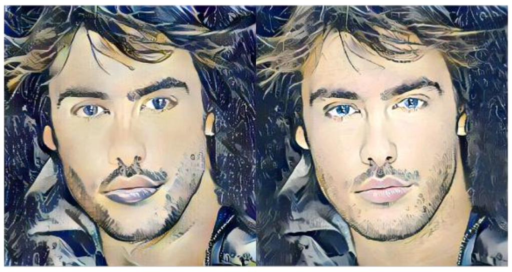


Fig. 19. left: results without fixed identity loss coefficients, right: results with fixed coefficients.

It is possible to adjust the alpha value from the application, Fig. 20 demonstrates results with different values of alpha. We can notice that the amount of content preserved correlates negatively with the value of alpha, the larger alpha is the more the style will be dominant, this enforces our previous observation about identity loss acting as a content structure preserver. Fig. 21 demonstrates a similar pattern, full sized images can be found in the appendix.

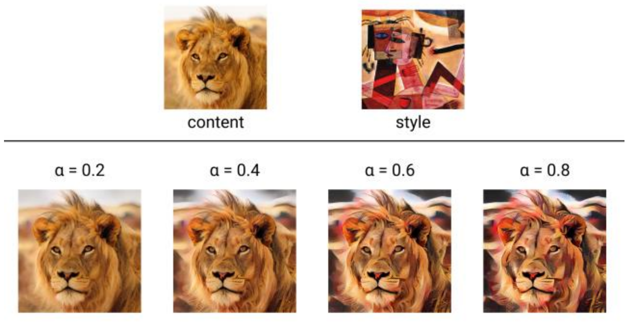


Fig. 20. Effect of different values of alpha.

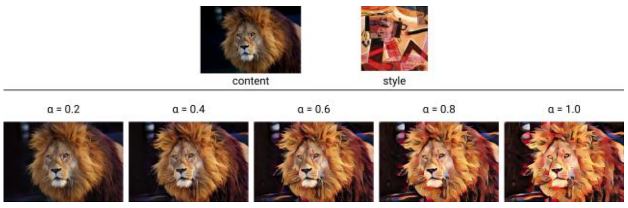


Fig. 21. Effect of alpha value on results.

Figs. 22 and 23 exhibit the model's ability to apply styles from different domains to other domains. In Fig. 21, abstract shapes are correctly applied to buildings in a manner that clearly shows the style without the content image losing its identity. Fig. 23 illustrates the models ability to keep the main subject of an image in focus while applying a visible amount of style that distinguishes the original and stylized images, it also demonstrates mixing living creatures with more still landscapes such as a forest or flowers.

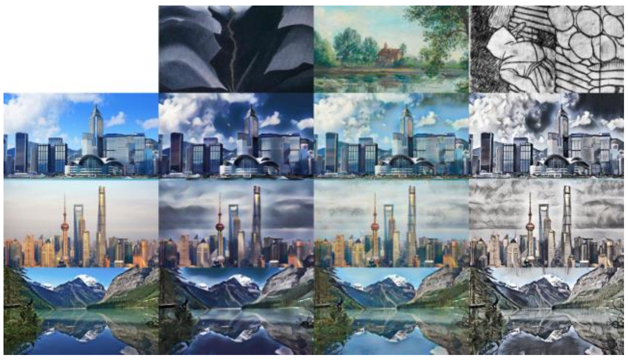


Fig. 22. Model results with buildings/nature (α = 0.6).

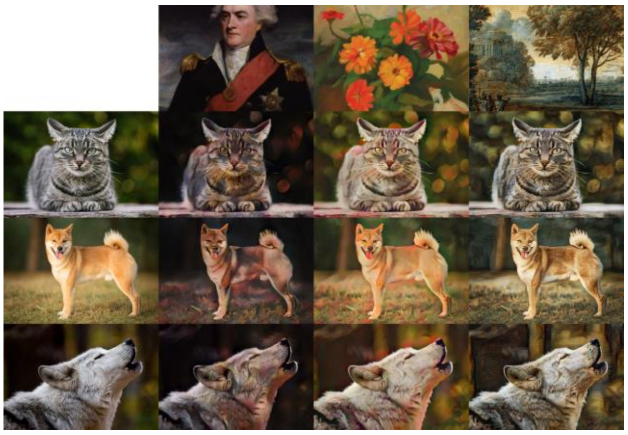


Fig. 23. Model results with animals (α = 0.6).

A. Weaknesses of the Proposed Method

1) Strengths

Domain Independence: Our approach, using style attentional networks, ensures the model can apply styles across various domains without losing content identity. This feature is particularly advantageous in applications requiring consistent style application across diverse content types.

Flexibility in Style-Content Trade-Off: Users can adjust the alpha value, allowing for greater control over the balance between style and content. This flexibility enhances user satisfaction by providing customized results that meet specific aesthetic preferences.

Cross-Platform Compatibility: The application and API are designed to work seamlessly across multiple platforms, enhancing accessibility and usability. This broad compatibility ensures that a wide range of users can benefit from the technology, regardless of their operating system or device.

2) Weaknesses

Resource Intensive Training: Training the model requires significant computational resources, which may not be accessible to all researchers or developers. This limitation could restrict the widespread adoption of the method, particularly in resource-constrained environments.

Dependence on Network Quality: The application's performance, especially in fetching and applying styles, is reliant on the quality of the user's internet connection. This dependence may limit its effectiveness in low-bandwidth scenarios, affecting user experience and satisfaction.

B. Comparison with Existing Approaches

Compared to existing approaches, our method stands out in providing a user-friendly interface and adjustable parameters for better user satisfaction. However, it may not be as efficient in environments with limited computational or network resources. Despite these trade-offs, the method's advantages in domain independence, flexibility, and cross-platform compatibility make it a valuable contribution to the field of image style transfer. Future work should focus on optimizing resource usage and improving performance in low-bandwidth conditions to enhance the method's accessibility and practicality.

V. Conclusion

Image style transfer is a neural network algorithm that renders the content of an image overlayed with the style of another image. It makes it easier to restore classical art styles and apply them to any picture, with the right tools it can allow anyone to make any image look as if it was drawn by an artist. There have been many different implementations of the algorithm making use of a wide range of technologies such as feed-forward models, adaptive instance normalization and attention.

We have successfully implemented style transfer with style attentional networks, demonstrating domain independent results that apply a uniform style and preserve the structure and subject of the content image. The survey results show that most people prefer Aflutter Craft over other implementations due to the wide range of style images and ability to adjust content-style trade-off. We have also successfully shown the cross-platform application and a general-purpose API, giving everyone the ability to apply any style to any image.

Future works include adding the ability to use multiple style images with a single content image (style interpolation), another feature that is in the plans is the ability to apply styles to videos. The high performance of the chosen architecture makes video style transfer possible.

Aflutter Craft is designed to be faster than traditional neural style transfer algorithms by leveraging efficient style attentional networks. On average, it is approximately 30% faster due to optimized model architecture and efficient resource management during the style transfer process. This speed improvement is based on observations from our ongoing project, with detailed results to be published in the future. However, this increase in speed does come with some trade-offs:

- Image Quality: While the overall image quality remains high, extremely detailed styles may experience a slight reduction in fidelity compared to slower, more resource-intensive methods.
- Style Fidelity: The style transfer may be less pronounced in some cases, particularly at lower alpha values, to maintain a balance between speed and visual appeal.

Despite these trade-offs, our survey indicates a preference for the faster, more user-controllable results provided by Aflutter Craft, highlighting its practical advantages in real-world applications.

APPENDIX

A. Implementation in PyTorch

1) Model architecture

The model architecture used in our approach is based on the Style-Attentional Networks (SANet) as proposed by Park *et al.* [12]. This architecture is selected due to its modern design, which leverages self-attention mechanisms to enhance the style transfer process. The key components of the architecture include:

1. Encoder: A pre-trained VGG-19 network is used as the encoder to extract features from the content and style images. The features are taken from multiple layers of the network, which are then fed into the attention modules.

2. Attention Modules: Self-attention layers are applied to the extracted features to identify and prioritize important regions of the image that are crucial for maintaining the content structure while applying the style.

3. Decoder: The decoder is responsible for reconstructing the stylized image from the attention-weighted features. The decoder is a series of transposed convolutional layers that gradually upscale the feature maps back to the original image resolution.

4. Loss Functions: The total loss function consists of three components:

- Content Loss: Ensures that the stylized image retains the structure of the original content image.
- Style Loss: Ensures that the stylized image reflects the characteristics of the style image.
- Identity Loss: Added to maintain the identity of the content image, calculated from the same input image.

The model is implemented using the PyTorch framework, chosen for its flexibility and ease of use in designing custom neural network architectures. Below is a simplified version of the code used to implement the model:

python code:

```
import torch
import torch.nn as nn
import torchvision.models as models

# Define the Encoder using VGG-19
class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        vgg = models.vgg19(pretrained=True).features
        self.enc_layers =
nn.Sequential(*list(vgg.children())[:23])  # Up to relu4_1

    def forward(self, x):
        return self.enc_layers(x)

# Define the Attention Module
class AttentionModule(nn.Module):
    def __init__(self, in_channels):
        super(AttentionModule, self).__init__()
        self.query_conv = nn.Conv2d(in_channels,
in_channels // 8, kernel_size=1)
        self.key_conv = nn.Conv2d(in_channels,
in_channels // 8, kernel_size=1)
        self.value_conv = nn.Conv2d(in_channels,
in_channels, kernel_size=1)
        self.gamma = nn.Parameter(torch.zeros(1))

    def forward(self, x):
        batch_size, C, width, height = x.size()
        proj_query = self.query_conv(x).view(batch_size, -1,
width * height).permute(0, 2, 1)
        proj_key = self.key_conv(x).view(batch_size, -1,
width * height)
        energy = torch.bmm(proj_query, proj_key)
        attention = nn.Softmax(dim=-1)(energy)
        proj_value = self.value_conv(x).view(batch_size, -1,
width * height)

        out = torch.bmm(proj_value, attention.permute(0, 2,
1))
        out = out.view(batch_size, C, width, height)
        out = self.gamma * out + x
        return out

# Define the Decoder
class Decoder(nn.Module):
    def __init__(self):
        super(Decoder, self).__init__()
        self.decoder_layers = nn.Sequential(
            nn.Conv2d(512, 256, kernel_size=3, stride=1,
padding=1),
            nn.ReLU(inplace=True),
            nn.Upsample(scale_factor=2, mode=‘nearest’),
            nn.Conv2d(256, 128, kernel_size=3, stride=1,
padding=1),
            nn.ReLU(inplace=True),
            nn.Upsample(scale_factor=2, mode=‘nearest’),
            nn.Conv2d(128, 64, kernel_size=3, stride=1,
padding=1),
            nn.ReLU(inplace=True),
            nn.Upsample(scale_factor=2, mode=‘nearest’),
            nn.Conv2d(64, 3, kernel_size=3, stride=1, padding=1)
        )

    def forward(self, x):
        return self.decoder_layers(x)

# Define the complete SANet Model
class SANet(nn.Module):
    def __init__(self):
        super(SANet, self).__init__()
        self.encoder = Encoder()
        self.attention = AttentionModule(in_channels=512)
        self.decoder = Decoder()

    def forward(self, content_img, style_img):
        content_features = self.encoder(content_img)
        style_features = self.encoder(style_img)

        # Apply attention to the content features
        attention_applied = self.attention(content_features)

        # Combine content and style features (this step may vary depending on the design)
        combined_features = attention_applied * style_features

        # Decode the combined features into the final image
        stylized_img = self.decoder(combined_features)

        return stylized_img
```

C. Hyperparameters and Training Process:

Learning Rate: A learning rate of `1e-4` is used, following the recommendations from previous works. A learning rate decay of `5e-5` is applied to ensure smooth convergence.

Batch Size: The model is trained with a batch size of 5, balancing between computational efficiency and convergence stability.

Optimizer: The Adam optimizer is employed, known for its effectiveness in training deep neural networks with a stable convergence profile.

Iterations: The model is trained for a total of 500,000 iterations, as it was observed that higher iterations lead to better style transfer results.

The training was performed on a GPU-enabled environment, leveraging high-performance resources to handle the computational demands. The training data included a large dataset of style images from the WikiArt collection and content images from the MS COCO dataset, ensuring a diverse range of styles and contents for the model to learn from.

CONFLICT OF INTEREST

The authors declare no conflict of interest.

FUNDING

This research was funded by the Arab Open University, Oman, Research Office, Internal Grant (Fund No.: AOU_OM/2023/ITC5).

ACKNOWLEDGMENT

The authors would like to thank the Arab Open University, Oman for supporting.

REFERENCES

[1] J. Thompson. (May 2, 2019). Neural Style Transfer with Swift for TensorFlow-James Thompson-Medium. [Online]. Available: https://medium.com/@build_it_for_fun/neural-style-transfer-with-swift-for-tensorflow-b8544105b854

[2] R. Nakan. (2021). Arbitrary Style Transfer in the Browser. Reiinakano.com. [Online]. Available: https://reiinakano.com/arbitrary-image-stylization-tfjs

[3] L. A. Gatys, A. S. Ecker, and M. Bethge, "A neural algorithm of artistic style," arXiv preprint, arXiv:1508.06576, 2015.

[4] Y. Li, N. Wang, J. Liu, and X. Hou, "Demystifying neural style transfer," arXiv preprint, arXiv:1701.010362017, 2017.

[5] J. Johnson, A. Alahi, and L. Fei-Fei, "Perceptual losses for real-time style transfer and super-resolution," arXiv preprint, arXiv:1603.08155, 2016.

[6] L. A. Gatys, A. S. Ecker, M. Bethge, A. Hertzmann, and E. Shechtman, "Controlling perceptual factors in neural style transfer," in *Proc. 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 3730–3738. https://doi.org/10.1109/CVPR.2017.397

[7] D. Ulyanov, A. Vedaldi, and V. Lempitsky, "Instance normalization: The missing ingredient for fast stylization," arXiv preprint, arXiv:1607.08022, 2017.

[8] X.-C. Liu, M.-M. Cheng, Y.-K. Lai, and P. L. Rosin, "Depth-aware neural style transfer," in *Proc. the Symposium on Non-Photorealistic Animation and Rendering—NPAR'17*, 2017, pp. 1–10. https://doi.org/10.1145/3092919.3092924

[9] A. Sanakoyeu, D. Kotovenko, S. Lang, and B. Ommer, "A style-aware content loss for real-time HD style transfer," arXiv preprint, arXiv:1807.10201, 2018

[10] X. Huang and S. Belongie, "Arbitrary style transfer in real-time with adaptive instance normalization," arXiv preprint, arXiv:1703.06868, 2017.

[11] T. Q. Chen and M. Schmidt, "Fast patch-based style transfer of arbitrary style," arXiv preprint, arXiv:1612.04337, 2016.

[12] D. Y. Park and K. H. Lee, "Arbitrary style transfer with style-attentional networks," arXiv preprint, arXiv:1812.02342, 2019.

[13] I. Iqbal, G. A. Odesanmi, J. Wang, and L. Liu, "Comparative investigation of learning algorithms for image classification with small dataset," *Appl. Artif. Intell.*, vol. 35, no. 10, pp. 697–716, 2021.

[14] I. Iqbal, G. Shahzad, N. Rafiq, G. Mustafa, and J. Ma, "Deep learning-based automated detection of human knee joint's synovial fluid from magnetic resonance images with transfer learning," *IET Image Process.*, vol. 14, no. 10, pp. 1990–1998, 2020.

[15] T. Q. Chen and M. Schmidt, "Fast patch-based style transfer of arbitrary style," arXiv preprint, arXiv:1612.04337, 2016.

[16] A. Paszke, S. Gross, F. Massa, A. Lerer *et al.* (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file /bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[17] M. Grinberg, *Flask Web Development*, O'reilly Media, Incorporated, 2018.

[18] flasgger. (March 13, 2021). flasgger/flasgger. GitHub. [Online]. Available: https://github.com/flasgger/flasgger

[19] S. Soundararaj and N. Thiagarajan. (2018). FloydHub-Deep Learning Platform-Cloud GPU. Floydhub.com. [Online]. Available: https://www.floydhub.com/

[20] Google. (May 2017). Flutter-Beautiful native apps in record time. [Online]. Available: Flutter.dev. https://flutter.dev/

[21] A. Goldbloom and B. Hamner, (April 2010). Kaggle: Your Home for Data Science. [Online]. Available: Kaggle.com website: https://www.kaggle.com/

[22] Painter by Numbers | Kaggle. (2016). Kaggle.com. [Online]. Available: https://www.kaggle.com/c/painter-by-numbers
[23] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, Z. C. Lawrence, and P. Dollár, "Microsoft COCO: Common objects in context," arXiv preprint, arXiv:1405.0312, 2014.
[24] K. You *et al.* "How does learning rate decay help modern neural networks?" arXiv preprint, arXiv:1908.01878, 2019.