Hong Kong University of Science and Technology
COMP 4211: Machine Learning
Spring 2024

Programming Assignment 2
Due: 27 March 2024, Wednesday, 11:59pm

Figure 1: Example of a Style Transfer Result

1 Introduction

This assignment is designed to provide students with a hands-on experience with implementation of generative models and application of generated data in data augmentation for classification tasks. Specifically, students will use the AdaIN (Adaptive Instance Normalization) style transfer model for generating new data and apply these techniques in a classification task.

2 Objectives

In this programming assignment, students are expected to learn to:

- Utilize the `tensorflow` and `keras` modules to construct and train a neural network model.
- Construct dataloaders that read and process image files from the filesystem as model input.
- Understand and implement the AdaIN style transfer model.
- Implement a custom training routine for Tensorflow models.
- Construct and train a neural network model for classification.
- Utilize the style-transferred images as a data augmentation technique for a classification task.
- Analyze and comment on the performance of neural network models, as well as the effect of data augmentation on model performance.

3 Tasks

This assignment is divided into two main parts: the coding part and the written report. Both parts are required for completing this assignment and will contribute to your final grade.

3.1 Coding Part

In the coding part of this assignment, you are tasked with:

1. **Building an AdaIN Style Transfer Model**: Implement the AdaIN style transfer model using TensorFlow and Keras. Ensure that your model correctly applies the style from one image to the content of another.

2. **Building a Classification Model**: Construct a model capable of classifying images. This model will be trained on images processed by your AdaIN style transfer model as part of a data augmentation strategy.

3. **Experimentation**: You are expected to experiment with different configurations, architectures, and hyperparameters to achieve the best performance from your models.

All code must be submitted in a Jupyter Notebook format containing all the running results. Make sure to include comments and explanations in your notebook to clarify how your code works and the rationale behind your implementation choices. We have provided a skeleton code for you. Note that $[C_n]$ refers to a specific coding task in the skeleton code.

3.2 Written Report

Report the results and answer some questions. $[Q_n]$ refers to a specific question that you need to answer in the written report.

Note 1: Refer to Section 9 for submission guidelines.

Note 2: Ensure that you adhere to the academic integrity guidelines in Section 11. Your submission should be your own work. In case where collaboration or external references are used, proper citations must be included in your report.

4 Environment Setup

All the necessary libraries are imported in the given Jupyter Notebook. Please refrain from importing additional libraries.

We recommend that all of the coding task be done on Google Colab using GPU as hardware accelerator. The option can be found in the arrow menu next to Connect → Change runtime type. Do not use CPU for training/testing or else the execution time will be extremely long.

Do note that Google Colab has a (hidden) quota for GPU usage, after which you will not be able to use the GPU in your runtime for a while. If that happens, you have three options:

- Wait until the (undetermined) timeout is over. You are therefore advised to start the assignment as soon as possible so that you have enough time to finish the required tasks.

- Pay for Google Colab Pro to have more access to their GPU.

- Run the Jupyter Notebook offline in your local machine. This is not recommended as we are unable to provide support for any technical issues in your computer.

  Here are the library settings verified to be working (but not required):

- `tensorflow==2.12.0`
- `cuda==11.8`

5 Datasets and Data Loaders

In this assignment, you will work with two distinct tasks: Style Transfer and Image Classification. Each task requires different datasets and consequently, different data loaders to efficiently manage the data during training.

5.1 Unzipping Datasets

We have provided code snippets to help you unzip the COCO, WikiArt, and PACS datasets from a shared Google Drive folder directly to your Google Colab Virtual Machine (VM). This step is crucial for making the datasets accessible for your data loaders.

5.2 Dataloader Details

In this section, you are required to implement two Dataloader classes: `ImageDataset` and `ClassificationDataset`.

`ImageDataset` is a dataloader that, given a path to a directory and a specific file extension, finds (recursively) all the files within the directory with the matching file extension, and loads on-demand during training. These files are unlabeled and are used for the Style Transfer task.

`ClassificationDataset` is a dataloader that, given a path to a directory and a `.tsv` file, loads the files listed in the TSV file and provides them with their perspective labels during training. The format of the TSV file is as follows:

- Each row represents an image sample with their corresponding labels. There are three tab-separated values in each entry, which are the file name, style and classification label respectively.
- The filename contains the base name of the image file, which should be found within the given directory path.
- There are no headers in the TSV file. Be careful when loading the file into a table or else you may have an incorrect amount of samples in your dataloader.

To facilitate data loading in the training and testing processes, you are required to implement the following for the two classes.

5.2.1 Initialization

The dataloaders should, given the corresponding parameters, collect a list of file paths along with other relevant information to be loaded during training or testing. Since the entries of this list are to be loaded one by one in each epoch, also initialize a pointer (denoted as `ds_pointer` here) for use during batch generation.

The parameters for `ImageDataset` are:

- The directory path to find images from
- The file extension to look for
- Batch size
- (optional) Seed for randomization, for reproducibility.

Since the exact file names are not provided, they have to be searched from the filesystem. The

function for search for matching files is given to you as `find_images()`.

The parameters for `ClassificationDataset` are:

- The directory path to load images from
- The file path to the TSV file
- Batch size
- (optional) Seed for randomization, for reproducibility.

Since the labels in the TSV file are strings, it is also recommended to create a one-hot encoding for the classification label (not needed for style).

Implement the `__init__()` method that performs the above.

5.2.2 Load Batch

Implement the `get_batch()` method, which when called returns a batch of training data according to the batch size provided during initialization. Note the following:

- Each sample should not be loaded more than once per epoch. Enforce this by using the `ds_pointer` and incrementing it after loading the images.
  If there are not enough samples left to be loaded in the batch (i.e., less than the batch size), reset the pointer and shuffle the list of data. The remaining samples in the dataset are skipped for this round.

- The images are to be loaded and preprocessed to a specific format. The function for loading and preprocessing are given as `get_image()` and `preprocess()`.

- For `ImageDataset`, the batch of images is the only thing you need to return.
  For `ClassificationDataset`, return both the batch of images and one-hot classification labels.

Note: While not required for grading, it is also recommended to shuffle the data once after loading them (see Section 5.2.4).

5.2.3 Dataset Length

Implement the `__len__` method, which returns the number of samples in the dataset. This is mainly used for calculating the number of batches in an epoch in the tasks.

5.2.4 Reset pointer

Implement the `reset_pointer()` method, which resets the `ds_pointer` to 0 and shuffles the list of data.

Note on randomization: For possible reproducibility, we recommend using the (optional) `seed` parameter as the seed for shuffling, or some internal random state (initialized using `random_state` in Section 5.2.1) if not provided.

This is **not required** for grading (in this or other sections), so you can ignore the randomizer seed parameters if you find this step too difficult. It is however useful if you want, for example, a consistent setup during testing (using the same `seed`).

If you do decide to incorporate the seeds however, make sure that in the case where a seed is not provided, the order after shuffling should be different from that before shuffling (with high probability), otherwise the list is not properly shuffled.

5.2.5 Add Data (`ClassificationDataset` only)

Implement the `add_data` method, which when called, augments the existing data loaded within the dataset instance by loading from another directory and TSV file, effectively using multiple sources as a single dataset. The parameters are similar to the constructor, without the batch size specification.

Reset the pointer and reshuffle the list afterwards (see Section 5.2.4).

5.3 Dataloader Implementation

$[C_1]$ $[Q_1]$ Implement `ImageDataset` according to the above description, and report the result of `__len__` when loading the COCO and WikiArt dataset respectively.

$[C_2]$ $[Q_2]$ Implement `ClassificationDataset` according to the above description, and report the result of `__len__` when loading the PACS training and test datasets respectively.

Note: For `ClassificationDataset`, you are advised to also store a pandas DataFrame of the TSV data (or something equivalent) and the number of label classes in the dataset in the class instance. These two will be used throughout the Classification coding tasks.

6 Style Transfer

This part of the assignment focuses on implementing the AdaIN style transfer model, encompassing the selection of datasets, construction of the data loader, and development of the model itself, including the encoder and decoder components. You can refer to the original paper here: this webpage: `https://arxiv.org/pdf/1703.06868.pdf`

6.1 Dataset Description

In this task, you will use the COCO dataset for content images and the WikiArt dataset for style images. These datasets provide a diverse collection of images that will enable you to explore a wide range of style transfer effects.

6.2 Model

6.2.1 Background Information

The AdaIN (Adaptive Instance Normalization) style transfer model brings together content and style from two different images. The core idea is to adjust the content image's feature statistics to match those of the style image, thus transferring the style effectively.

6.2.2 Building the Encoder with VGG19

The encoder component of our style transfer model employs the VGG19 architecture. VGG19 is renowned for its effectiveness in feature extraction, making it an ideal choice for capturing the content and style of images.

6.2.2.1 VGG19 Architecture

$[C_3]$ Construct a partial VGG19 model up to the `conv4_1` layer to serve as our encoder. Below is the detailed structure of the model layers, including the naming convention used for each layer:

Layer Type	**Configuration**	**Output Size**	**Name**
Convolution + ReLU	64 filters, 3x3, same padding	Input Size	`conv1_1`
Convolution + ReLU	64 filters, 3x3, same padding	Input Size	`conv1_2`
Max Pooling	2x2, stride 2	Input Size / 2	`pool1`
Convolution + ReLU	128 filters, 3x3, same padding	Input Size / 2	`conv2_1`
Convolution + ReLU	128 filters, 3x3, same padding	Input Size / 2	`conv2_2`
Max Pooling	2x2, stride 2	Input Size / 4	`pool2`
Convolution + ReLU	256 filters, 3x3, same padding	Input Size / 4	`conv3_1`
Convolution + ReLU	256 filters, 3x3, same padding	Input Size / 4	`conv3_2`
Convolution + ReLU	256 filters, 3x3, same padding	Input Size / 4	`conv3_3`
Convolution + ReLU	256 filters, 3x3, same padding	Input Size / 4	`conv3_4`
Max Pooling	2x2, stride 2	Input Size / 8	`pool3`
Convolution + ReLU	512 filters, 3x3, same padding	Input Size / 8	`conv4_1`

Table 1: Structure of the VGG19 Model (Partial) with Layer Naming

Tip: This module (and other modules involving just a stack of layers) can be implemented with the keras `Sequential` function.

Important Note: It is crucial that you adhere strictly to the naming convention provided for each layer of the VGG19 model. The names assigned to the layers are utilized in subsequent steps, such as constructing `build_vgg19_relus()` and defining the loss function. Inconsistencies in naming could lead to errors or unexpected behavior in these components of your model. Ensure each layer is named exactly as specified to maintain compatibility throughout your implementation.

6.2.2.2 Loading Pretrained Weights Pretrained weights are crucial for leveraging the VGG19 model's capability to extract meaningful features from images. The weights are typically saved in a NumPy file (in this case `vgg19.npz`), which you will need to load into the model.

[C_4] Follow these steps to correctly load the pretrained weights:

1. First, load the weights file using the `numpy.load` function. This file contains the pretrained weights and biases for the VGG19 model stored in a structured array format. Each layer's weights and biases are saved with specific keys following the pattern `'arr_i'` for some 0-based index `i`.

2. Iterate through each layer of the model in the `model.layers` order. Whenever you find a convolution layer (identified by the prefix `'conv'` in their name), load the weights and biases as follows:

   - Each convolution layer corresponds to 2 arrays in the file, in the form of `'arr_i'` and 'arr_$(i+1)$' for some i. These represent the kernel and bias weights respectively. For example, the weights for the first convolutional layer are stored under `'arr_0'`, and the biases under `'arr_1'`.

     The recommended way to match the indices is to keep an index `i` starting from 0 and increment it by 2 every time a convolution layer's weights (and biases) have been loaded.

   - Retrieve the kernel (weights) and biases with the corresponding key (in the form of `weights[key]`).

   - Transpose the kernel to match TensorFlow's weight format using `kernel = kernel.transpose([2, 3, 1, 0])`. This rearranges the kernel's dimensions from (`#output channels`, `#input channels`, `height`, `width`) to TensorFlow's expected format (`height`, `width`, `#input channels`, `#output channels`).

   - Convert the kernel and bias to `np.float32` type if necessary, to ensure compatibility with the TensorFlow model.

   - Use `layer.set_weights([kernel, bias])` to load the prepared weights and biases into the model layer.

3. After loading the weights into all relevant layers, set `model.trainable = False`. This action freezes the weights during training, allowing the encoder to maintain its feature extraction capabilities without adjustment.

This method ensures that the pretrained VGG19 model is correctly initialized with weights that have been trained on a large dataset, providing a strong feature extraction capability for the style transfer task. Students should pay close attention to the indexing and transposition steps

to avoid errors in weight assignment.

6.2.2.3 (Given) `build_vgg19_relus` The `build_vgg19_relus()` function is designed to extract activations from specified ReLU layers within our VGG19 encoder. By selecting key layers (`conv1_1`, `conv2_1`, `conv3_1`, `conv4_1`), we can capture essential content and style features from images. This tailored model outputs the activations from these layers, facilitating the computation of style and content losses in our style transfer framework.

6.2.3 Building the Decoder

The decoder is an essential part of the style transfer model that reconstructs an image from the encoded features, allowing the transferred style to be visualized in the output image.

6.2.3.1 Decoder Architecture Below is the architecture of the decoder model. This model is designed to progressively upscale the feature maps and decrease their depth back to the three channels of a standard RGB image.

Layer Type	**Configuration**	**Output Size**
Convolution + ReLU	256 filters, 3x3, same padding	Input Size
UpSampling	2x2	2x Input Size
Convolution + ReLU	256 filters, 3x3, same padding	2x Input Size
Convolution + ReLU	256 filters, 3x3, same padding	2x Input Size
Convolution + ReLU	256 filters, 3x3, same padding	2x Input Size
Convolution + ReLU	128 filters, 3x3, same padding	2x Input Size
UpSampling	2x2	4x Input Size
Convolution + ReLU	128 filters, 3x3, same padding	4x Input Size
Convolution + ReLU	64 filters, 3x3, same padding	4x Input Size
UpSampling	2x2	8x Input Size
Convolution + ReLU	64 filters, 3x3, same padding	8x Input Size
Convolution	3 filters, 3x3, same padding	8x Input Size

Table 2: Architecture of the Decoder Model

This table outlines the sequence of layers in the decoder, starting from the encoded features' input size and progressively upscaling to reconstruct the image. Note that the final convolutional layer has 3 filters, corresponding to the RGB channels of the output image, and does not include an activation function, allowing the model to output raw pixel values.

[Q_3] What is the usage of upsampling layers?

[C_5] [Q_4] Implement the decoder according to Table 2. Report the number of trainable parameters in the decoder.

[Q_5] Compare the architectures of the encoder and decoder. Discuss the usage of the decoder architecture.

6.2.4 (Given) AdaIN Layer

The AdaIN (Adaptive Instance Normalization) layer is a core component of our style transfer model, playing a critical role in the style transfer process. This layer takes a content image

and a style image as input and aligns the channel-wise mean and variance of the content input to those of the style input. Unlike other normalization layers like Batch Normalization (BN) or Instance Normalization (IN), AdaIN does not have learnable affine parameters. Instead, it computes these parameters adaptively from the style input.

Given content input x and style input y, the AdaIN operation is defined as follows:

$$\text{AdaIN}(x, y) = \sigma(y) \left(\frac{x - \mu(x)}{\sigma(x)} \right) + \mu(y)$$

where $\mu(x)$ and $\sigma(x)$ represent the mean and standard deviation of the content features, and $\sigma(y)$ and $\mu(y)$ represent those of the style features. These statistics are computed across spatial dimensions, ensuring that the style transfer occurs in the feature space by transferring specific feature statistics, particularly the channel-wise mean and variance.

The AdaIN layer effectively allows for the combination of the content from the input image and the style from the style image by transferring feature statistics, which results in a stylized image that preserves the content structure while adopting the appearance of the style reference. Additionally, this process is computationally efficient as the AdaIN layer adds minimal overhead, making real-time style transfer feasible.

Note to students: In this programming assignment, you are not required to implement the AdaIN layer as it has already been provided to you. Understanding its underlying theory and functionality, however, is crucial for grasping how the style transfer model operates.

6.2.5 (Given) The complete AdaIN Style Transfer model

To construct the AdaIN style transfer model, we will utilize the encoder and decoder architectures previously described. The model will take a content image, a style image, and an alpha blending parameter as inputs. These inputs pass through the encoder to produce feature maps which are then adjusted by the AdaIN layer to match the mean and variance of the style feature maps to the content feature maps. The modified content feature map is then passed through the decoder to generate the stylized image.

Below is a conceptual overview of the AdaIN style transfer model (see Figure 2). The VGG-19 network is used up to the `relu4_1` layer to encode both content and style images. The AdaIN layer is then used to perform style transfer in the feature space. A decoder network is trained to invert the AdaIN output back to the image space. The same encoder network is also employed to compute a content loss L_c and a style loss L_s.

The alpha parameter controls the trade-off between content and style, with $\alpha = 0$ producing an image close to the original content and $\alpha = 1$ producing an image that closely resembles the style characteristics.

The AdaIN style transfer model is constructed using the provided `build_model()` function. This function is responsible for integrating the encoder, the AdaIN layer, and the decoder into a complete end-to-end model. The encoder and decoder architectures follow the guidelines provided in the respective sections of this document.

Note to Students: The `build_model` function is already given to you, and there is no need for you to implement it. You will utilize this function to instantiate the AdaIN style transfer

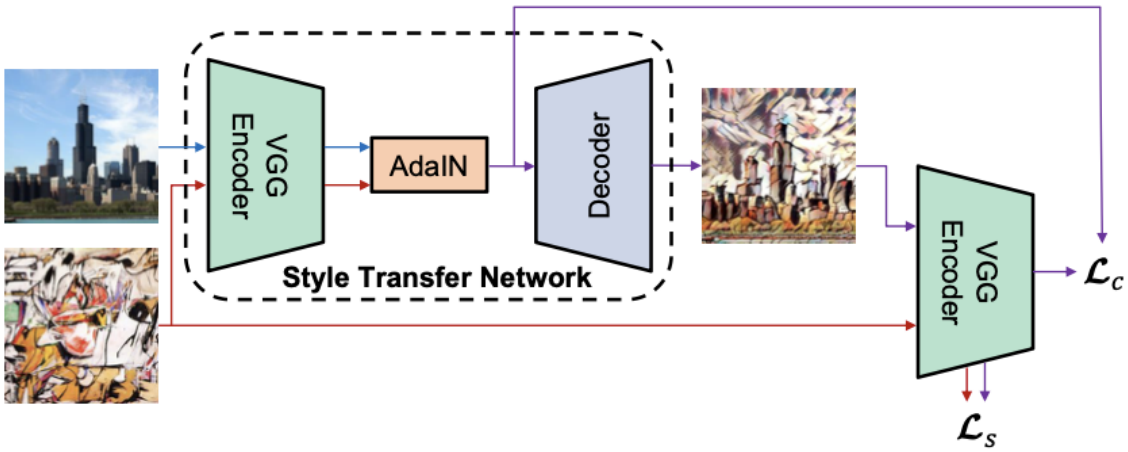

Figure 2: An overview of the AdaIN style transfer algorithm. The encoder maps content and style images into a common feature space where the AdaIN layer aligns the feature distributions. The decoder then inverts the features back into image space to generate the stylized output.

model. This function takes the content and style inputs and the degree of style transfer, α, to output the stylized image and the transformed features after the AdaIN layer.

6.2.6 (Given) Loss Function

Following the construction of the AdaIN Style Transfer Model, it is crucial to define a loss function that guides the training process towards generating images that not only preserve the content of the input images but also accurately capture the style of the style images. The loss function consists of two main components: the content loss (L_c) and the style loss (L_s), combined to form the total loss.

6.2.6.1 Content Loss The content loss ensures that the transformed image maintains the original content of the input image. It is computed as the mean squared error between the features of the output image and the features of the content image after the AdaIN transformation.

6.2.6.2 Style Loss The style loss measures how well the style of the style image is transferred to the output image. It involves comparing the Gram matrices of the style features between the generated image and the style image, capturing the texture and color information.

6.2.6.3 Total Loss The total loss is a weighted sum of the content loss and the style loss, where the weighting factor (`style_weight`) adjusts the relative importance of the style loss in the total loss calculation. This balance allows for control over the extent to which the style is transferred to the content image.

The provided `get_loss()` function implements this loss calculation, utilizing the encoder and a series of VGG-19 ReLU layers (`vgg19_relus`) to extract the necessary features from the images. The function employs an epsilon value for numerical stability during division operations and allows for the adjustment of the style weight to fine-tune the style transfer effect.

[Q_6] In the training of a style transfer model, why is it necessary to use a combination of style loss and content loss?

6.3 Training Routine

For this task, you will implement a custom training routine for the model, which a skeleton is provided for you as `part1_train()`. The function is provided with the `model`, a `loss` function, the relevant dataset and other training settings like number of epochs to train for or the save/load location for the model.

The training routine should consist of the following steps:

6.3.1 Before the training loop

1. **Initialize the Optimizer**: Initialize an Adam optimizer for updating the model gradients. For an effective training process, the recommended hyperparameter to use is a learning rate of `1e-4` and weight decay of `5e-5`.

2. **Calculate Batch Count**: Compute the total number of batches for training based on the dataset's size and batch size.

3. **(given) Initialize other parameters and objects**, for the subsequent steps.

4. **(given) Load model weights**, if a file path is provided. This allows you to continue the progress from a previous training.

6.3.2 During the training loop

1. **Reset Dataloader Iterators**: At the start of each epoch, reset the iterators for both content and style datasets to ensure that training uses the entire dataset anew.

2. **(Given) Reset Losses**: We would like to monitor the performance of the model as the training goes on. A dictionary is initialized to record the losses to be tracked, which in this case are the `total`, `content`, and `style` losses.

3. **Fetch Training Batches and Train the model**: In this step of the training process, your task involves preparing the inputs for the model and calculating the loss based on its predictions. Each training step involves the following:

    - Load a batch of samples from the datasets (content and style here)
    - Pass the content and style images (along with an `alpha` value) wrapped in a single array to obtain the model output.
    - Pass the model output along with the content and style image to the `loss` function to compute the loss.
    - Obtain the gradients of the `total` loss with respect to the model's trainable variables, using `tf.GradientTape()`.
    - Apply the gradients using the optimizer initialized before the loop. This is where the model learns by updating its weights based on the computed gradients.

4. **Update Loss Metrics**: With the loss computed in the previous step, update all the values in the loss dictionary so that the function can reflect the results after each batch.

   Make sure the update correctly calculates the **average loss over the batches** trained so far.

5. **(Given) Visualize Results**: A progress bar created using the `tqdm` library is rendered to visualize the epoch progress and current losses in real time.

6. **(Given) Save Model Weights**: In case the training process is interrupted due to manual intervention or runtime problems, the weights are saved to the given save path every few epochs. This can be adjusted in the function parameters.

[C_6] Implement the training function `part1_train()` given the above instructions.

Tip: Refer to `build_model()` for how the input (or output) for the model are structured.

6.4 Training the Style Transfer model

6.4.1 Model Setup and Hyperparameters

Once the training function is ready, proceed to configure your model for the training session. The `part1_setup` function is where you define key hyperparameters:

- **STYLE_WEIGHT**: Controls the contribution of the style loss to the total loss.
- **EPSILON**: A small constant used for numerical stability in the loss function.
- **BATCH_SIZE**: Determines the number of images processed together in each training step.
- **INPUT_SHAPE**: The shape of the input images. Ensure that it matches your dataset.

These parameters significantly influence the training dynamics and the quality of the generated images.

6.4.2 Executing the Training Process

With the model and loss function configured, initiate the training process.

[Q_7] Train your model for at least 15 epochs aiming for a total loss of less than 18000. Report the loss obtained at the end of training.

Important Recommendations:

- Save your model weights regularly to your Google Drive to prevent loss of progress due to unexpected Colab session terminations.
- If training is interrupted, you can reload the saved weights using the `load_path` parameter in the `part1_train` function and continue from where you left off.

These steps are designed to guide you through the training of your style transfer model, emphasizing the importance of hyperparameter tuning and iterative training with checkpointing. Monitor the loss metrics closely, and adjust your training strategy as needed to meet the target

loss.

6.5 Inference

After training, you can now use the model to perform style transfer on new images, which will be done using the `part1_inference()` function. Given a path of the model weights, content image and style image, along with the `alpha` parameter, it should do the following:

1. (Given) Create a style transfer model with the weights loaded from the given path.
2. Load and preprocess the content and style images from the given paths (the same way you load the image in the data loader). Make sure the loaded images have 3 dimensions (the first being the "batch" dimension of size 1).
3. Convert the `alpha` scalar into a Tensorflow tensor of dimension 1 (the "batch" dimension of size 1).
4. Generate an image from the model with the 3 tensors using `model.predict()`.
5. Convert the image back to 2 dimensions, then to the proper RGB format using the given `deprocess()` function.
6. (Given) Display the image in the notebook output (or save it somewhere).

[C_7] Implement the `part1_inference()` function according to the above description.

[Q_8] Demonstrate the model's ability to combine the content and style of images effectively, showcasing several examples in your report.

Note: Pay close attention to the implementation details of both the encoder and decoder, as the quality of the style transfer relies heavily on these components. Experiment with different architectures for the decoder and different loss weight ratios to achieve the best results.

7 Classification Task

This part of the assignment focuses on implementing a CNN model to solve a multiclass classification problem, and applying the style transfer model as an (offline) data augmentation technique to improve the test-time performance of the classifier model.

7.1 Problem Statement

This classification task uses the **PACS** (Photo-Art-Cartoon-Sketch) dataset, an image dataset with images of different objects and in different styles.

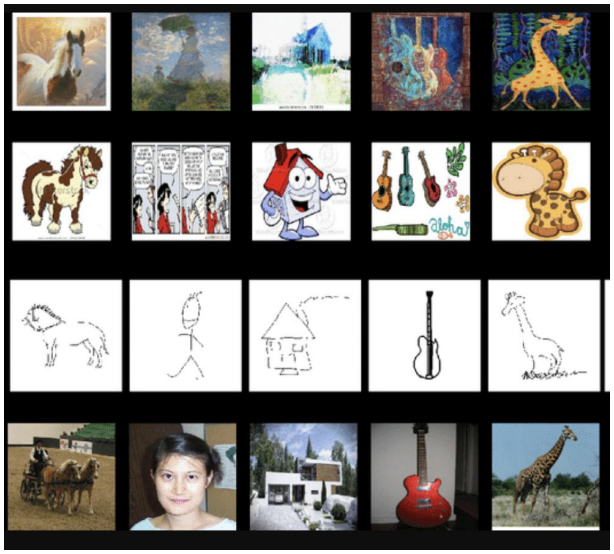

Figure 3: Sample Images from the PACS Dataset

The goal of this task is to predict the object labels (denoted as `label` from now on) given the images (in different styles), but with a catch: In the provided training dataset, the images available are in a non-ideal distribution among different styles and classes. The idea of this task is thus to overcome this issue with the training data by producing additional samples using the style transfer model, using the images available to you.

The training and test datasets are provided in the zip file `classify/pacs-dataset.zip` in the provided Google Drive.

Warning: Do not use the test data or any other images outside the provided ones during your training process, unless stated otherwise. You should assume that the only available data during training-time are the training data images.

7.2 Analyzing the Dataset

Before we get started with the model implementation, we first analyze the training dataset.

[C_8] Implement a function to count the number of samples for each pair of (`style`, `label`) in a dataset.

[Q_9] Compare the distribution of image styles and labels between the training and test datasets. What is the difference between the two datasets?

[Q_{10}] Write down your prediction on how the above difference would affect test-time performance.

7.3 Model Implementation

The classification model is a simple extension of the VGG19 model in Table 1. In particular, we add the following layers afterwards:

Layer Type	**Output Size**	**Name**
Global Average Pooling	512 dimension	`global_pool`
Dense + ReLU	1024 dimension	`dense1`
Dense + ReLU	1024 dimension	`dense2`
Dense + ReLU	512 dimension	`dense3`
Dense + Softmax	(Number of classes)	`dense4`

Table 3: Additional Layers for the VGG19-based classifier model, layer names optional

[C_9] [Q_{11}] Implement a function to construct the classifier model. Report the number of trainable parameters in the model.

7.4 Training the Classification model

[C_{10}] Implement the training function. The idea of the training process is similar to that of Section 6.3, but with the following changes:

- **Datasets**: Only one dataloader is required for the classification training.
- **Model Input**: The model takes one input and returns one output, unlike the style transfer model. The inputs and outputs are not wrapped in arrays.
- **Losses**: Categorical Cross Entropy Loss is used in this training. Pass the true labels from the batch and the predicted labels from the model to the function to obtain the loss.

  Since this is the only loss to be tracked. Modify the relevant steps to track a single `cross_entropy` loss.

A skeleton code `part2_train()` is provided for you.

[Q_{12}] Train the model with the given training dataset for at least 100 epochs. Record the loss obtained at the end of training (or better, throughout training).

7.5 Testing Routine

[C_{11}] Implement a testing function. Given a model and a dataset:

- Calculate and accumulate the accuracy of the model across all batches in the given dataset.
- Calculate the confusion matrix of the model on the given dataset.

[Q_{13}] Report the accuracy and confusion matrix of the model trained in the previous section, when tested against the training and test datasets respectively.

[Q_{14}] Showcase some of the mislabeled sample images (with their true and reported samples). Discuss your observations (and possible explanation) on some of these failure cases.

7.6 Data Augmentation by Style Transfer

We now generate new samples using the available training data. We can generate a sample of label ℓ and style s by the following procedure:

- Pick a random image A with label ℓ, and another B with style s.
- Using the trained style transfer model, create a new image C with A as content image and B as style image. α is a hyperparameter that can be adjusted for fine-tuning the performance.
- Save the image C somewhere (e.g., as a file in the server) for future use, along with its labels. For simplicity, you may want to store it in a similar format as the given datasets.

  Some of the image/file saving code has been provided for you.

[C_{12}] Create a function that performs the above augmentation. Generate at least 50 samples for each style-label pair, as a new dataset (denoted as "generated dataset" from now on).

[Q_{15}] Showcase some samples from the generated dataset, with at least one per object label, and one per style.

7.7 Retraining with Augmented Dataset

[Q_{16}] Train a new model using a dataset combining both the given and generated training datasets for at least 100 epochs, and report the loss at the end of training.

[Q_{17}] Test the new model against the test (and un-augmented training) dataset, then report and compare the result with one before augmentation.

[Q_{18}] Give a possible explanation on the two results (e.g., how the augmentation affected the test-time performance).

7.8 Self-Exploration: Refining the Augmentation process

The following section serves as ideas for self-exploration on how to improve the training process, and will not be counted towards your grade for this assignment.

- Train and test a model trained on just the generated dataset. How is the performance affected?
- Try increasing or decreasing the number of samples generated, or vary them by style/label combinations. How much does it affect the test-time performance?
- Any other modifications to the augmentation or training procedure that can improve the model performance?

Experiment and report your findings in your written report.

8 Some Programming Tips

As is always the case, good programming practices should be applied when coding your program. Below are some common ones but they are by no means complete:

- Using functions to structure your code and scope variables properly
- Using meaningful variable and function names to improve readability
- Using consistent styles
- Including concise but informative comments
- Using a small subset of data to test the code
- Using checkpoints to save partially trained models

9 Assignment Submission

Assignment submission should only be done electronically in the Canvas course site.

There should be two files in your submission with the following naming convention required:

1. **Report** (with filename `report.pdf`): in PDF format.

2. **Source code and prediction** (with filename `code.zip`): all necessary code and running processes should be recorded into a single ZIP file. The ZIP file should include at least one notebook recording all the training and evaluation results. The data should not be submitted to keep the file size small.

When multiple versions with the same filename are submitted, only the latest version according to the timestamp will be used for grading. Files not adhering to the naming convention above will be ignored.

10 Grading Scheme

This programming assignment will be counted towards 15% of your final course grade. The maximum scores for different tasks are shown below:

Table 4: [C]: Code, [Q]: Written report

Grading Scheme	Code(64)	Report(36)
Dataloader Implementation (12)		
- [C1 + Q1] Implement `ImageDataset`	5	1
- [C2 + Q2] Implement `ClassificationDataset`	5	1
Style Transfer (43)		
- [C3] Build `VGG19` model up to `conv4_1`	4	
- [C4] Load pretrained weights	6	
- [Q3] Explain upsampling layers		2
- [C5 + Q4] Build decoder model	4	1
- [Q5] Compare encoder and decoder architectures		2
- [Q6] Explain loss function combination		2
- [C6] Implement `part1_train` function	8	
- [Q7] Execute training & report loss		3
- [C7] Implement `part1_inference` function	8	
- [Q8] Demonstrate Model		3
Classification (45)		
- [C8] Tally dataset by style-label pair	2	
- [Q9] Compare the dataset distributions		2
- [Q10] Prediction of performance		2
- [C9 + Q11] Build `VGG19`-based classifier model	2	1
- [C10] Implement `part2_train` function	6	
- [Q12] Execute training & report loss		3
- [C11] Implement testing function	8	
- [Q13] Report accuracy and confusion matrix		2
- [Q14] Showcase mislabeled sample images		2
- [C12] Data augmentation (generate new dataset)	6	
- [Q15] Showcase samples from generated dataset		2
- [Q16] Train a new model & report loss		3
- [Q17] Test the new model and compare the result		2
- [Q18] Explain result		2

Late submission will be accepted but with penalty.

The late penalty is deduction of one point (out of a maximum of 100 points) for every hour late after 11:59pm with no more than two days (48 hours). Being late for a fraction of an hour is considered a full hour. For example, two points will be deducted if the submission time is 01:23:34.

11 Academic Integrity

Please refer to the regulations for student conduct and academic integrity on this webpage: `https://registry.hkust.edu.hk/resource-library/academic-standards`.

While you may discuss with your classmates on general ideas about the assignment, your submission should be based on your own independent effort. In case you seek help from any person or reference source, you should state it clearly in your submission. Failure to do so is considered plagiarism which will lead to appropriate disciplinary actions.