ENHANCING AMATEUR PHOTOGRAPHY: A DEEP LEARNING MOBILE APPLICATION FOR REAL-TIME AESTHETIC FEEDBACK

Tian Zhan[1], Austin Amakye Ansah[2]

[1]Brooks School, 1160 Great Pond Road, North Andover 01845
[2]The University of Texas at Arlington, 701 S Nedderman Dr, Arlington, TX 76019

ABSTRACT

Capturing aesthetically pleasing photographs can be challenging for amateur photographers due to the complexity of factors such as lighting, composition, and contrast. To address this issue, we propose a mobile application powered by deep learning models and regression analysis. This application analyzes real-time image frames using a pre-trained MobileNet backbone and a custom classification layer [8]. By leveraging the Aesthetics and Attributes database, the app calculates an aesthetic score for each photograph, providing instant feedback to users. Challenges encountered during development, including interfacing with machine learning models and implementing camera functionalities, are addressed. Through experiments, we evaluate different training approaches and compare our methodology with existing research. Our solution aims to empower users to capture high-quality photographs by assisting them in understanding and applying fundamental principles of photography.

KEYWORDS

Machine Learning, Mobile, Tensorflow, Flutter

1. INTRODUCTION

It can be difficult to find the right angle or scenery when taking pictures. The aesthetic quality of an image is judged by commonly established photographic rules, affected by factors like lighting, contrast, and image composition. It is very hard for the average person to take aesthetically pleasing photographs when they are not professional photographers. If the average phone holder were able to instantly know when their camera shots were satisfying or perfect, it would be easier and quicker to take good photos when necessary and accumulate better photography aesthetics.

We present a mobile application that utilizes image regression to assist users in capturing better photos or analyzing scenery for acceptable artistic or photographic aesthetic. Flutter is used for the frontend, whereas tensorflow and tensorflow lite are utilized for real-time image classification and inference. The model was trained using the AADB dataset on the CUDA platform provided by PyTorch. The model inference is done on a flask server written in Python that hosts an ONNX representation of the model [7].

2. CHALLENGES

In order to build the project, a few challenges have been identified as follows.

David C. Wyld et al. (Eds): DSCC, AI&FL, SESBC, CSE– 2024
pp. 09-16, 2024. - CS & IT - CSCP 2024 DOI: 10.5121/csit.2024.141602

2.1. Which machine learning library to use

There was an issue choosing the right flutter library to interface with the trained machine learning model. Google MLKit is a good choice, but it proved difficult to adapt to the trained image model from the teachable machine platform [1]. TFlite flutter was also a good choice to use our model directly, but it was difficult to write the code to interface the model to flutter. A lot of isolate programming was required to get the model running fast without halting the app. Ultimately, it was decided that Pytorch would be used in a server. This meant that the solution wouldn't run in real time, but any issues present with the tflite flutter library were no longer considerable.

2.2. Picture-taking issues

There were some issues when implementing camera snapping capabilities. One of these issues was that the camera would take a picture twice and halt the app because the camera was not ready for the second shot. A fix to this could be to make sure that the camera is granted permission and is not taking a picture currently before executing the camera shot process.

2.3. Sizing issues on the gallery screen

Lastly, the gallery screen proved to have some issues. Flutter's semantic debugger would halt when the screen was loaded due to sizing issues with the images and the cards used to display the images in a list [9]. A fix to this could be to crop the images so that they are the same size as the cards that contain them. There was also the requirement to fit all of the images to the gallery screen at once, so some spacing and limitation to the cross axis count was required.

3. SOLUTION

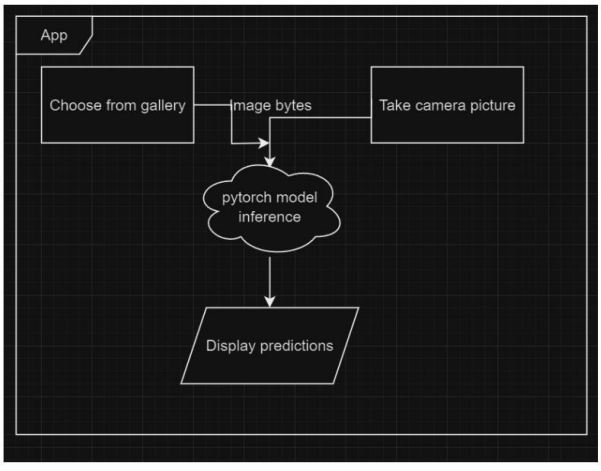


Figure 1. App Logic Flowchart

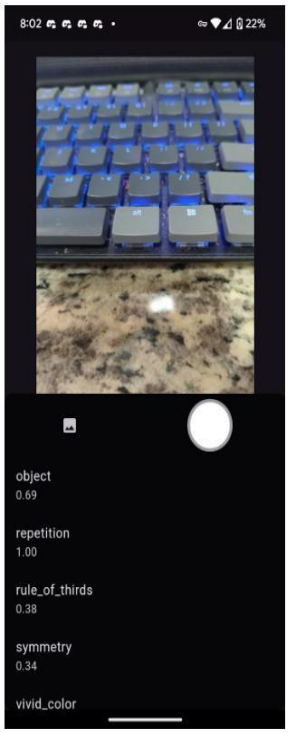
Our solution focuses on regressing an image to certain ratings like depth of field, lighting, balance, etc. The app has a basic page that allows users to either choose photos from their gallery or camera. The image is sent to the server the second it is selected. At the current iteration of the solution, it is up to the user to decide when they think they have a good image to take a shot of. The solution is a regression task and so the machine learning model is trained to rank an image

based on 10 different quality factors: balancing elements, color harmony, content, depth of field, light, motion blur, object, repetition, rule of thirds, symmetry, and vivid color. The final model was trained using Pytorch.

Flutter is used for the app frontend to capture and save pictures. The app also features a gallery to observe captures. The camera feature is basic and allows for capturing and changing camera faces.

3.1. Camera Screen

The camera screen is the main component of the app that displays the model output as well as the user's camera feed [10]. It features two buttons to take shots and switch camera sides. After a



picture is taken, it is saved to the device's image storage location for the app.

Figure 2. Image saving

```
// save to gallery
final appDir = await getApplicationDocumentsDirectory();
final fileName = imageFile.path.split('/').last;
final savedImage = await image.copy('${appDir!.path}/$fileName');
log('Image saved to gallery: ${savedImage.path}');
}
```

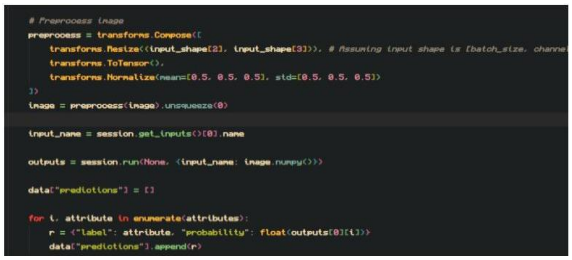


Figure 3. ML inference

This piece of code is responsible for processing images and running them through the PyTorch model [11]. Simply put, the engine outputs a dictionary of the class labels for the model output. The scores for the image are sent to the client and displayed on the screen.

3.2. Gallery

This component focuses on the gallery screen which can be used to show pictures taken by the camera app.

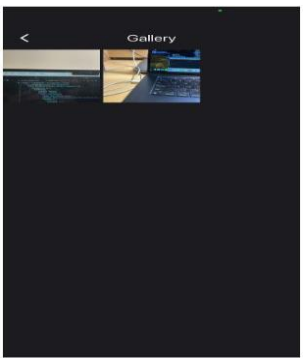


Figure 4. Screenshot of gallery

```
Future<bool> isImageValid(List<int> rawList) async {
  final uInt8List =
      rawList is Uint8List ? rawList : Uint8List.fromList(rawList);

  try {
    final codec = await instantiateImageCodec(uInt8List, targetWidth: 32);
    final frameInfo = await codec.getNextFrame();
    return frameInfo.image.width > 0;
  } catch (e) {
    return false;
  }
}
```

```
Future<List<String>> getImages() async {
  // get image paths
  final directory =
      await getExternalStorageDirectories(type: StorageDirectory.pictures);
  final dir = Directory(directory![0].path);
  final files = dir.listSync();
  final imageFiles = files.where((element) {
    return element.path.endsWith(".jpg") ||
        element.path.endsWith(".jpeg") ||
        element.path.endsWith(".png");
  }).toList();

  return imageFiles.map((e) => e.path).toList();
}
```

Figure 5. Screenshot of code 2

In Figure 4, we present a basic interface for users to see their previously captured photos. Before an image is loaded into the image grid, it is important to make sure that it is valid. We use a try-catch clause in Figure 5 so that if loading the image from raw byte data fails, then it can be inferred that the image is invalid. In Figure 5, we load all of the image paths from the app's storage directory and later use those path strings to render images locally to the scene tree.

4. EXPERIMENT

In Figure 1a, we present a basic interface for users to see their previously captured photos. Before an image is loaded into the image grid, it is important to make sure that it is valid. We use a try-catch clause in Figure 5 so that if loading the image from raw byte data fails, then it can be inferred that the image is invalid. In Figure 1c, we load all of the image paths from the app's storage directory and later use those path strings to render images locally to the scene tree.

4.1. Design

The first step involves create a backbone for the regression model. The backbone is any pretrained model or convolutional neural network that can process image data. The backbone is responsible for feature extraction for each image sample. Each sample is a 256x256 colored image with 3 channels, and as a result, the input shape of the backbone is (256, 256, 3). In the second step, the regression neural network is created by first setting the input layer, a (256, 256, 3) shape layer that takes an arbitrary batch size of samples. The backbone is then added as a second layer on top of the input and a regression block is added. The regression block consists of a GlobalAvergagePooling2D layer and a Linear layer for regressive tasks [12].

The final model is then trained on the input data over about 20-30 epochs.

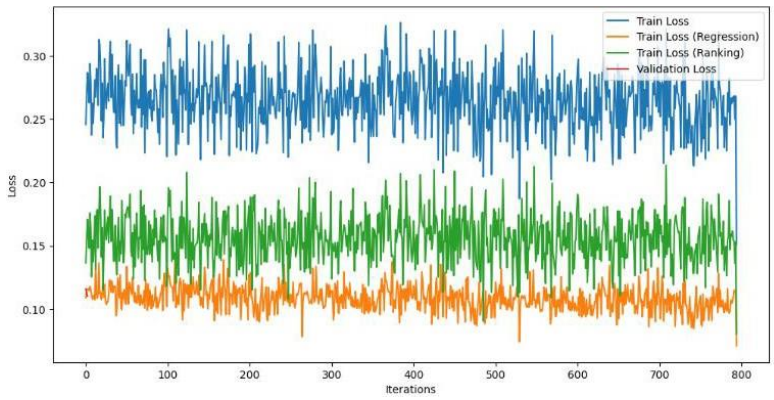


Figure 6. Figure of experiment 1

4.2. Analysis

None of the model losses decreased during the training of the model. There were several fluctuations in the loss, but on average, there was no significant change or improvement in the model performance. The training loss for the regression and ranking loss functions remained relatively constant throughout the training loop. One plausible reason for this is that the model is unable to pick out the correct features to train on when an image is passed though it.

The goal of the second training loop was to determine how well the model could learn if given pretrained weights using a MobileNetV2 backbone. In this scenario, the number of training epochs was reduced to 15.

In both cases, the model was adapted from a kaggle notebook that used PyTorch [6]. In this case, Tensorflow was used to streamline the process of running the model on mobile and reduce memory utilization.

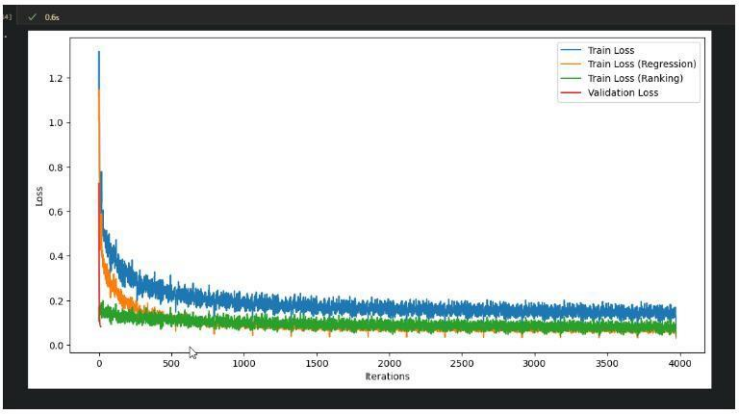


Figure 7. Figure of experiment 2

The fluctuation of the losses could be a sign of overfitting in the model, so hyperparameters were adjusted to get better results [13].

The results of the experiment show that the losses converge from 500 to 1,500 iterations. With this in mind, the number of epochs required to train the model can be dropped significantly to reduce training time.

Eventually, PyTorch was favored as the better machine learning library, and yielded similar, but more stable results in less time [14].

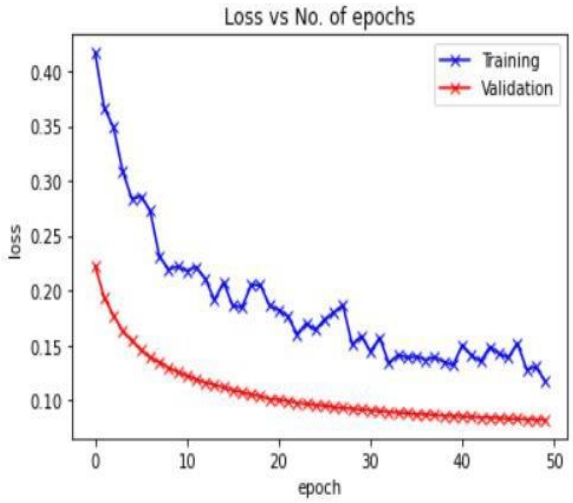


Figure 8. Figure of experiment 3

5. RELATED WORK

We analyzed the research by Lu, et al, and compared their implementation to our own [2]. Their approach trains an image classifier specialized for image aesthetics. They use a double column deep convolutional neural network (DCNN) to learn multiple image features. Their model is also trained on 1.5 million images, making it efficient for aesthetic classification in almost all kinds of images.

Secondly, we looked at an implementation by Alzayer, et al [3]. Their implementation serves to automate the process of taking good pictures by utilizing robots and deep learning. Their approach uses reinforcement learning to train their model on the AVA dataset to correctly classify and rate images with desirable aesthetics. The output of their model determines what direction the robot will move in and what kind of picture its camera will take.

Thirdly, we examined the paper Towards Artistic Image Aesthetics Assessment: a Large-scale Dataset and a New Method by Ran Yi et al [4]. The paper described the process of developing a deep neural network for determining image aesthetic. Our solution utilizes the AADB dataset which initially comes with around 10,000 images. Their dataset has around 60,000 images for training. Their solution aims to extract the style specific aesthetic feature of an image, whereas ours extracts the generic aesthetic score of an image sample. They use Resnet50 as the backbone for the feature extractor, while our older model used MobileNetV2 for feature extraction. After switching to a server-oriented approach, we made use of resnet for its accuracy.

6. CONCLUSIONS

Summary of machine learning experiments

Throughout the initial design of the image aesthetic model, we determined that using a pre-trained image model to extract features would be best since we are dealing with light, image orientation, and smaller details in images that would be more difficult to detect with a custom model. We used the MobileNetV2 architecture from Keras and used a custom output layer for the regression task. Initially, we used a pre-trained model with no weights, resulting in a model that had to learn features from scratch. In this scenario, model accuracy was poor, and loss dropped slowly.

With pre-trained weights, the loss dropped significantly after the first few epochs, and the accuracy of the model increased [5]. It was found that the model reached convergence in under 6 epochs with the current hyperparameters. Further tuning could see improved performance in the model.

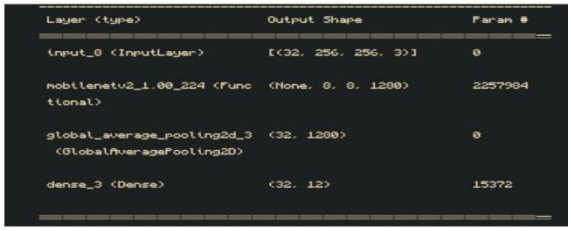


Figure 9. Figure of layer and output

PyTorch's ONNX platform proved to be better in the end for model inference, despite having to rely on a server [15]. Hopefully, in the future, there will be better on-device ONNX runtimes for Flutter.

REFERENCES

[1] Mathew, M., and Y. M. Therese. "Object detection based on teachable machine." Journal of VLSIDesignand Signal Processing 7.2 (2021): 20-26.

[2] Lu, Xin, et al. "Rating image aesthetics using deep learning." IEEE Transactions on Multimedia 17.11 (2015): 2021-2034.

[3] AlZayer, Hadi, Hubert Lin, and Kavita Bala. "Autophoto: Aesthetic photo capture using reinforcement learning." 2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE, 2021.

[4] Yi, Ran, et al. "Towards artistic image aesthetics assessment: a large-scale dataset and a new method." Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2023.

[5] Deng, Yubin, Chen Change Loy, and Xiaoou Tang. "Image aesthetic assessment: An experimental survey." IEEE Signal Processing Magazine 34.4 (2017): 80-106.

[6] Jin, Xin, et al. "IDEA: A new dataset for image aesthetic scoring." Multimedia Tools and Applications 79 (2020): 14341-14355.

[7] Shridhar, Ayush, Phil Tomson, and Mike Innes. "Interoperating deep learning models with onnx. jl." Proceedings of the JuliaCon Conferences. Vol. 1. No. 1. 2020.

[8] Vasu, Pavan Kumar Anasosalu, et al. "Mobileone: An improved one millisecond mobile backbone." Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. 2023.

[9] Cheon, Yoonsik, and Carlos Chavez. "Converting Android native apps to Flutter cross-platform apps." 2021 International conference on computational science and computational intelligence (CSCI). IEEE, 2021.

[10] Jansen, Michiel JW. "Analysis of variance designs for model output." Computer Physics Communications 117.1-2 (1999): 35-43.

[11] Subramanian, Vishnu. Deep Learning with PyTorch: A practical approach to building neural network models using PyTorch. Packt Publishing Ltd, 2018.

[12] Moczulski, Marcin, et al. "Acdc: A structured efficient linear layer." arXiv preprint arXiv:1511.05946 (2015).

[13] Probst, Philipp, Marvin N. Wright, and Anne-Laure Boulesteix. "Hyperparameters and tuning strategies for random forest." Wiley Interdisciplinary Reviews: data mining and knowledge discovery 9.3 (2019): e1301.

[14] Paszke, Adam, et al. "Pytorch: An imperative style, high-performance deep learning library." Advances in neural information processing systems 32 (2019).

[15] Manca, Federico, Francesco Ratto, and Francesca Palumbo. "ONNX-to-Hardware Design Flow for Adaptive Neural-Network Inference on FPGAs." arXiv preprint arXiv:2406.09078 (2024).