



Module Name: Deep Learning

Module Code: CT100-3-M-DL

Title: Learn Unsupervised Localized Activation Maps through Image Classification Model

Student Name: Kelvin Hong Khay Boon

Student ID: TP069358

Degree and Programme: MSc in Artificial Intelligence

Module Lecturers:

Prof. Dr. Mandava Rajeswari,

Assoc. Prof. Dr. Raja Rajeswary A/P Ponnusamy.

Date: 18th Jun 2023

Abstract

Most deep learning models, although powerful, act like a black box where it is hard to explain its decision-making process. We review the Class Activation Mapping (CAM) technique that modifies the last layer of a classification model on some image dataset, so that in addition to the usual logits' prediction, one can view a map that tells us which region from the image activates such prediction. When training an image classifier, researchers found out that the model can learn a certain map indicating the region of interest on the image without explicit supervision. Such activation map can be used to explain the reason of classification. Our report will be focus on building a simple baseline model on creating the activation maps of our datasets, and then improve the architecture to generate gradually more precise activation maps. We use the Flower-17 dataset that is available on Kaggle to experiment the CAM technique, which has not been done before (Kamal, 2021). To speed up our experiment, we use the feature extractor of a pretrained VGG16 model and freeze its parameter during training, which allow efficient memory usage when storing our trained models.

Keywords: Deep Learning, Image Classification, Convolutional Neural Network, Class Activation Mapping, Weakly Supervised Learning, Pseudo-mask.

Table of Contents

Abstract	2
Table of Contents	3
Introduction and Background	4
Literature Review and Matrix.....	8
Justification and Review on Existing Models	12
Implementation	16
Discussion.....	18
Classification Metrics.....	18
Critical Analysis on Classification Metrics	19
CAM generations	19
Critical Analysis on CAM generations.....	22
Conclusion.....	23
References	24
Appendix	26
Code	26

Introduction and Background

We have seen competitive advancements on deep learning in the past few years, and it is only going faster. Among all of the deep learning categories, image classification remains the most researched and there are plenty of state-of-the-art models available. Some examples include Residual Neural Network (ResNet) that alleviates gradient vanishing issue in a very deep neural network (He et al., 2016; Wightman et al., 2021) and the YOLO series that keep improves the result of classification (Wang et al., 2022).

With such improvements on classification accuracies, it strikes curiosity in our heart to wonder how exactly does a classification model makes a decision that an image belong to one of the classes? This process is kind of like mathematical theories, where people first assume something heuristically work and then take plenty of times to justify the rationale of its mechanics, in contrasts to common beliefs that mathematical theorems are first rigorously proved before widely used. However, current deep learning model, especially the ones involving convolutional neural network (CNN) layers are too complex to be able to clearly justified, thus we might take a step back and try to explain better than what we can for now. Class Activation Mapping (CAM) is one of such techniques that improves the interpretability of a classifier (Lin et al., 2014; Zhou et al., 2015; Patro et al., 2019; Jung & Oh, 2021), as shown in Figure 1. The CAM technique clearly explain the decision of classifying the data as “Australian terrier” by showing that the last convolutional layer is activated by the “dog” part of the image, rather than the “human” part. In the discussion below we will use CAM to refer to the class activation mapping technique and the class activation map itself interchangeably.

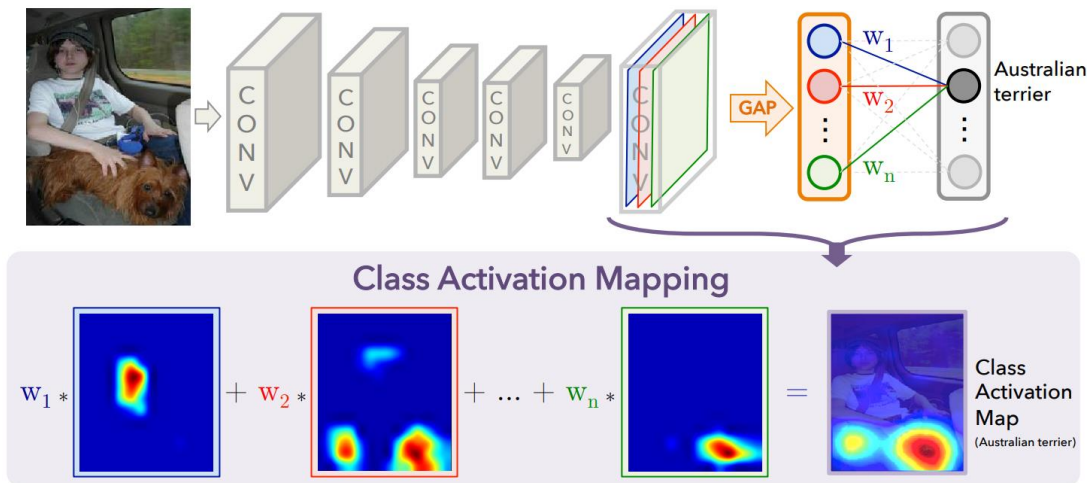


Figure 1: CAM visualization of an Australian terrier. (Image taken from Zhou et al., 2015)

The importance of CAM can sometimes help explain some ambiguity in a classifier, especially when the dataset is biased. For example, one classifier could be handling a task of classifying animals, and one of the classes is “polar bear”. It is highly possible that all of the “polar bear” images are taken in a snow-white scenery because it is unlikely for them to live in a more tropical environment, classifier could falsely rely on the white scenery for identifying “polar bear” class, which might lead to false classification when there is a white fox in a snow scenery. Therefore, we can analyze the CAM of some “white bear” images to know the model’s CAM is relying on the snow scenery or the bear itself, informing us potential issue with the model in advanced.

Domain Knowledge and Significance

Class Activation Mapping (CAM) can generate a mask similar to a heatmap indicating the region of interests. To use CAM, one does not need to create an entirely new model or heavily alter existing architecture. It is normally done by starting with a pretrained classification model then only modify the last linear layer of the model as in Figure 1. If the classifier is multiclass, one can generate CAM on a single class, where if it is a multilabel classifier, it can generate multiple CAMs overlaying on a single image, acting as pseudo-masks in a weakly-supervised manner (Jiang et al., 2015) (Figure 2).

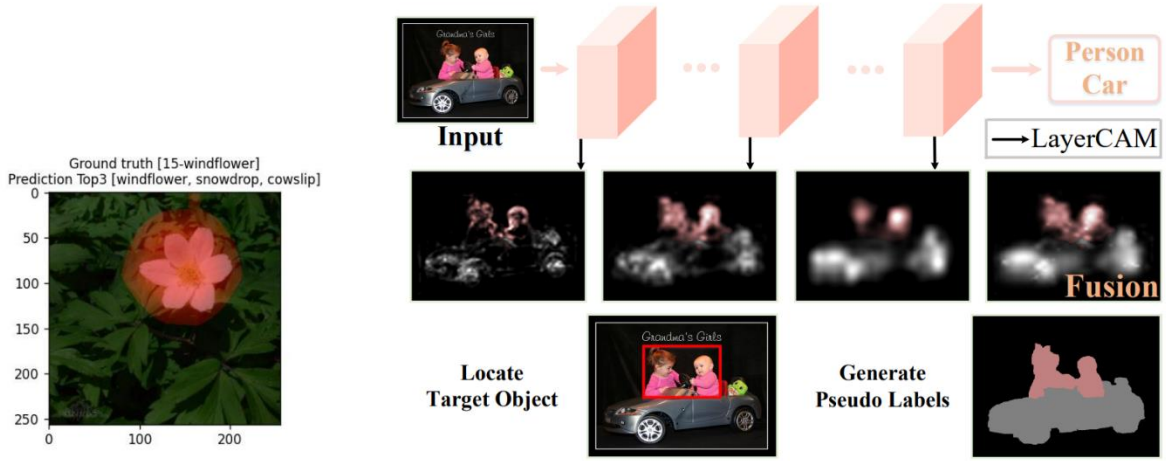


Figure 2 Multiclass and Multilabel CAM (left-side taken from our experiment, right-side taken from Jiang et al., 2015)

The first implementation of CAM does not generate a detailed map, several improvements have been carried since then, either by using feature maps from shallow layers or rethink the weight coefficients.

Our main objective is to investigate various implementations of CAM technique by using a pretrained VGG16 network and analyze its result critically. Moreover, different implementations will be carried out to generate fine grained CAM to better visualize the region of interest by the classifier. This project is mostly interesting in a particular way, that CAM generation can be done in a weakly supervised manner. This is because it only relies on the image level label where no mask is given. This provides another practical entry where instance segmentation is needed without ground truth masks at hand (Chen et al., 2022; Liu et al., 2022).

The significance of this report is to study difference CAM generating methods, and apply it to a sufficiently complicated but seldom used datasets (Flower-17) to illustrate the generalization ability of such CAM methods. Based on this, we only compare results different architectures because there is little experiment done on the Flower-17 dataset. We aim to improve the quality of CAM on our dataset by gradually implement more complex but manageable architectures. Furthermore, we provide code snippets in Appendix so that our implementations can be easily duplicated. The full repository can be cloned from GitHub: <https://github.com/KelvinHong/DL-assignment>.

Literature Review and Matrix

This section layout the literature matrix used for this report. We detail the methodology, analysis and results discovered from these literatures.

Paper Info	Methodology	Analysis and Results
Patro, B. N., Lunayach, M., Patel, S., & Namboodiri, V. P. (2019). U-CAM: Visual Explanation using Uncertainty based Class Activation Maps. IEEE/CVF International Conference on Computer Vision (ICCV), 7444–7453.	Using uncertainty-based and gradient-based CAM to visualize decision making process of a Visual Question Answering (VQA) model. Such model receives a question and an image, and should return an answer to the question. The uncertainty-based method here helps interpret which region from the image the model focusing at to generate the response.	Yields improved uncertainty estimates and can correlate bad decision to high uncertainty, focus on better attention region that is more suitable for answering the question in context. This can be an improved version of a VQA method when combining the highlighted region with the response as it makes more sense and people can consider the model interpretable.
Jung, H., & Oh, Y. (2021). Towards Better Explanations of Class Activation Mapping.	When constructing CAM, the original implementation is to heuristically use the coefficients from last linear layer. This article improves this choice by using SHAP values and analytically estimate better coefficients.	The proposed LIFT-CAM estimates the SHAP values analytically for the activated map with only one backward pass in the neural network. It achieves state-of-the-art results when comparing to other explanation models.

<p>He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep Residual Learning for Image Recognition. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 770–778.</p>	<p>Neural networks, especially with convolutional layers, suffers from gradient vanishing problems and accuracy degradation when the network become too deep. Accuracy degradation in a classifier is a phenomenon when increasing model capacity lead to lower accuracy. To address this problem, the authors introduces residual connection that lets features skip through layers, thus preserve magnitude of gradient and enable improved performance for even deeper network.</p>	<p>The difference between a standard deep neural network to the authors’ version is they introduced identity mapping by shortcuts, that add the original input into the output of a network. This alone helps alleviate gradient vanishing and improves the model performance. In the experiments, the authors compare plain networks (without residual connection) and their networks, both with 18 and 34 layers each. They showed plain network has worse performance when increasing layers, but residual network has improved loss when adding more layers. Finally, ResNet with 152 layers achieved best top1 and top5 errors when comparing to other architectures such as VGGNet, GoogLeNet and other ResNet with lesser layers.</p>
<p>Wightman, R., Touvron, H., & Jégou, H. (2021). ResNet strikes back: An improved training procedure in timm. ArXiv Preprint.</p>	<p>ResNet has been popular since the creation of its time (2015) as it introduces skip connection to solve vanishing gradient issue in a deep CNN. However, a lot of best practices have emerged since the appearance of ResNet. This paper re-evaluate ResNet with new training regimes to observe any new potentials in ResNet. The “timm” in the title stands for “PyTorch Image Model”.</p>	<p>By incorporate modern training techniques in CNN including learning rate decay, label smoothing, dropout, repeated augmentation, random augmentation, color jitter, PCA lighting and the others, the authors report comprehensive results and conclude that they have achieve new state-of-the-art training method for ResNet50 architecture. Nevertheless, they pointed out their training regimes are non-standard, which leads to some suboptimal performance on other model architectures.</p>

Liu, Y., Lian, L., Zhang, E., Xu, L., Xiao, C., Zhong, X., Li, F., Jiang, B., Dong, Y., Ma, L., Huang, Q., Xu, M., Zhang, Y., Yu, D., Yan, C., & Qin, P. (2022). Mixed-UNet: Refined class activation mapping for weakly-supervised semantic segmentation with multi-scale inference. <i>Frontiers in Computer Science</i> , 4.	Weakly supervised semantic segmentation (WSSS) method can reduce the workload of training a disease detection model on medical images so that it only depends on image level label to generate activation maps. This paper uses multi-scale U-net model to generate CAMs.	The authors created Mixed-Unet architecture that first encode the input image, then uses two parallel branches to upscale the result then mix the results afterward. This in practice shows the model perform better based on the multiscale inference and the parallelization of the upscale operation. However, the authors discussed about further research when images have unreliable labels as it could hinder the activation map's quality.
Zhou, B., Khosla, A., Lapedriza, A., Oliva, A., & Torralba, A. (2015). Learning Deep Features for Discriminative Localization. <i>Arxiv</i> .	The authors use global average pooling (GAP) layer to help visualize activation map of images in an unsupervised manner. This can be done by only relying on image-level label since the activation map can be calculated by skipping the GAP layers, then applying the coefficients of last linear layer directly to the features extracted from the CNN layers.	This method although trained while being unsupervised, it brought excellently well interpretation to explaining the labels of image. In some case it can also explain false label. For example, for an image containing a dome and a palace beneath them, the model output a softmax vector that its top-1 label is palace and top-2 label is dome, and when calculating class activation map (CAM) separately for these two classes, the dome CAM focus on the upper part and palace CAM focus on the lower part of the image. This model serves as a baseline for future unsupervised object localization technique.
Jiang, P.-T., Zhang, C.-B., Hou, Q., Cheng, M.-M., & Wei, Y. (2015). LayerCAM: Exploring Hierarchical Class Activation Maps for Localization. <i>Journal of Latex Class Files</i> , 14(8), 1–14.	The authors incorporate features extracted from shallower layers to add fine grain details on the final CAM. They use gradient-based method by calculate activation intensity from gradient magnitude as shallower layers have no direct relation to the last linear layer comparing with vanilla CAM.	By including gradient activation from shallower layers of the model, the authors report better object localization ability. Moreover, they can localize multiple objects in a single image as the original model is aiming for detecting multiple objects. The LayerCAM technique is also inherently better than vanilla CAM as it uses different weights on each individual pixels depending on gradient activation, hence reduce false positive on its CAM generation.

Selvaraju, R. R., Cogswell, M., Das, A., Vedantam, R., Parikh, D., & Batra, D. (2017). Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization. 2017 IEEE, 618–626.	In contrast to the heuristic coefficients taking from vanilla CAM, the authors proposed to use gradient magnitude as an alternative when calculating CAM. The CAM calculation is similarly only used on the feature extracted from the last CNN layer.	The authors find out that by using GradCAM, it increases the interpretability and trustworthiness of a model. It can discriminate objects from different classes more accurately and make decision based on unbiased region. Furthermore, GradCAM can be used to explained the response in a visual question answering (VQA) model.
Chattopadhyay, A., Sarkar, A., Howlader, P., & Balasubramanian, V. N. (2018). Grad-CAM++: Improved Visual Explanations for Deep Convolutional Networks. 2018 IEEE, 839–847.	On top of the previous GradCAM model, the authors introduce GradCAM++ that uses second derivatives, namely, gradient of gradient to calculate CAM. This version can be considered generalization of GradCAM as it behave like a differently weighted combination of feature maps.	GradCAM++ tackles the shortcoming of detecting multiple objects in the GradCAM model. This new model can also generate good image caption and understanding video better. More importantly, GradCAM++ motivates the training of shallower network (student network) that require less memory to run and achieve similar result.
Chen, Z., Wang, T., Wu, X., Hua, X.-S., Zhang, H., & Sun, Q. (2022). Class Re-Activation Maps for Weakly-Supervised Semantic Segmentation. ArXiv Preprint.	On top of vanilla CAM generation, the authors introduce ReCAM – a model that uses CAM as a pseudo-mask during training to help model focus on more prominent features in a feature map. Therefore, this model will first generate a set of CAMs, then use that CAMs to regenerate CAM, namely ReCAM.	The authors compare CAM generated by vanilla CAM and their ReCAM models on a multi-label task. They report that ReCAM reduces false negative and false positive pixels compare to vanilla CAM when two objects are intruding each other. It also performs well on some WSSS benchmarks.

Table 1

Justification and Review on Existing Models

A huge reason for selecting Class Activation Mapping (CAM) technique for this report is from the curiosity of understanding the rationale behind a working classifier. Modern state-of-the-art image classifiers that validate on the ImageNet Dataset (Deng et al., 2009) are usually very heavy and huge, around 200 million parameters. While certain effort has been spending on keeping the model small, including model distillation, pruning and quantization, understanding the decision-making process of a classifier is also equally important especially decision making in the medical field.

The original CAM implementation (Zhou et al., 2015) simply modifies the last layer of a classification model to include a Global Average Pooling (GAP) layer before feeding the neurons into the last linear layer. To elaborate in terms of technical details, a classification model can be viewed as a concatenation of a feature extractor and a direct classifier. Given an RGB image of shape $[3, H, W]$, let f be its feature map of shape $[C, K, K]$ (we assume the feature maps are squares). Since we have discarded the original classifier from the model, we design a new end classifier as follows: A GAP layer that average the values from each map hence giving an output of shape $[C]$, assuming this output is written as g and there are N classes in the dataset, we feed g through a linear layer to obtain an output of shape $[N]$ (Figure 3). This output can then be softmaxed and perform other operations normally for a classification model. Also note that GAP is a function for pure operation, it doesn't have any trainable parameters.

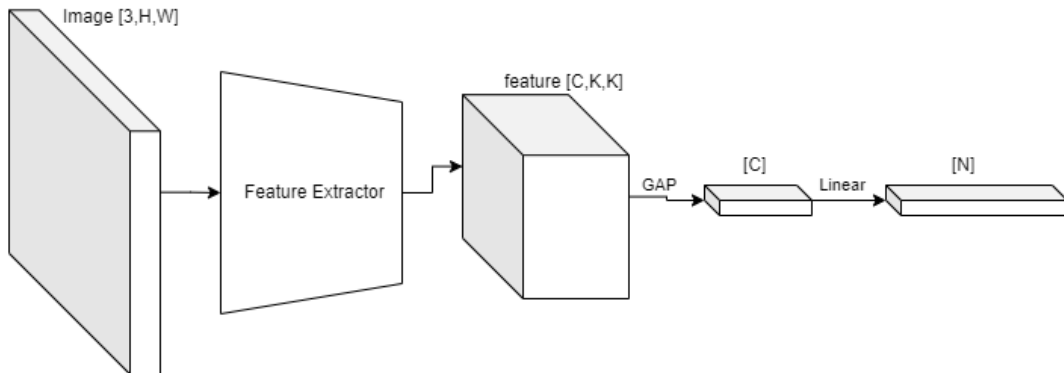


Figure 3 CAM architecture, obtain by slightly modify existing image classification model.

At first glance, this is just a standard image classification model, but after the training is completed, it can be used to generate CAM. The last linear layer has weight and bias parameters that are of shape $W [N, C]$ and $b [N]$. By skipping the GAP layer, one can pass the feature map of an image directly to the last linear layer to obtain a collection of CAMs with shape $[N, K, K]$. Then, resizing the CAMs to $[N, H, W]$ constitutes as a rough activation map for each of N classes.

The role of Global Average Pooling layer in CAM is subtle: It merely average the neuron activations from the feature map. While requiring no trainable parameters, it suits perfectly well with the objective. Since the model’s last linear layer is trained on the GAP output from the feature map, it reacts well to the original feature map as well. This is why CAM can be generated by skipping GAP layer, letting us observe the regions activating the decision (Figure 4).

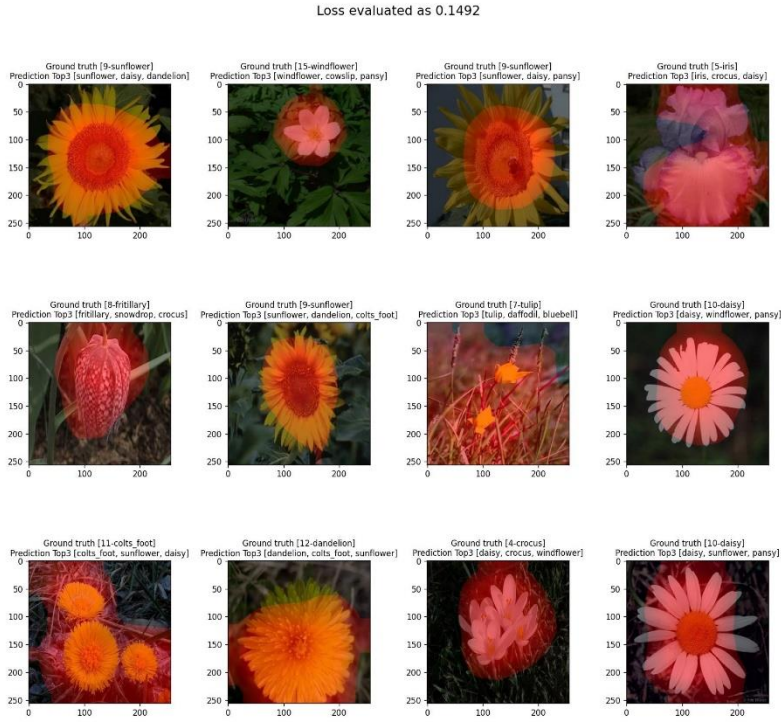


Figure 4 Original CAM Implementation on Flower-17 dataset.

However, the approach of CAM generates blurry mappings because the feature maps has very small dimensions, the last linear layer’s weight is heuristically used in the process, and the generated CAM might include false positive regions (for example, the 7th image from Figure 4). To remedy the situation, Chen et al. (2022) created ReCAM that aims to reactivate

class activation mapping in order to reduce false positive. Designed originally for multilabel model, the model is trained to first generate a set of CAMs for the multi-hot classes on an image. Then, normalized CAMs are used to mask over the original feature maps before generating another set of CAMs, which is called ReCAMs (Figure 5). The authors argue that this approach can make the CAM focus on a more prominent features of the image since the importance was highlighted by the first set of CAMs.

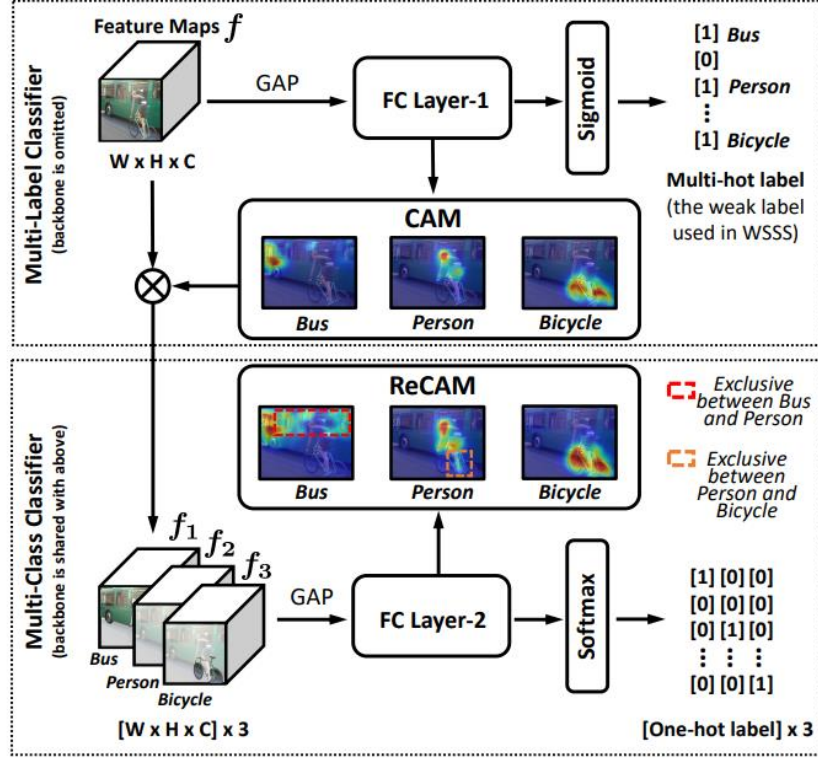


Figure 5 ReCAM architecture using a shared feature extractor, taken from Chen et al. (2022). The masking effect can be seen from the processed feature maps on the lower-left corner.

However, ReCAM still uses the weight parameters from the last linear layer, it is heuristic and assumes the weight suitable for the output of global average pooling layer is also suitable for the features itself. There are more rigorous approaches providing finer details on the generation of CAMs.

Recent innovation of gradient-based CAM models, such as Grad-CAM and Grad-CAM++ achieved far better visual results compared to the previous discussed approaches (Chattopadhyay et al., 2018; Selvaraju et al., 2017). In the framework of Grad-CAM, classification from an image classifier can generate gradients from the end to the start of the model. On the feature extractor level, the gradient is inspected and normalized (usually by a tangent hyperbolic function on nonnegative values) to obtain maps ranging from 0 to 1, thus constructed CAM in this way. The design is based off the believe that model will make a

decision based on the more influential neurons, hence assign higher relevance to the neurons with higher gradients in magnitude. The Grad-CAM++ framework increases this sensitivity by including second order gradients during the calculation, but it inevitably makes the computation more expensive.

LayerCAM is yet another gradient-based CAM model which utilizes low-level and high-level feature maps to generate realistic CAM (Jiang et al., 2015). It is based on a pretrained VGG16 model, consisting of 5 main convolutional blocks, each generate a feature regarding different level of details. Using the 3rd, 4th and 5th features from the VGG16 model, the authors generate a CAM in a hybrid manner that outperforms the previous two models (Figure 6). Although it is only being done on VGG16, its core implementation can be done on other standard architectures as well.

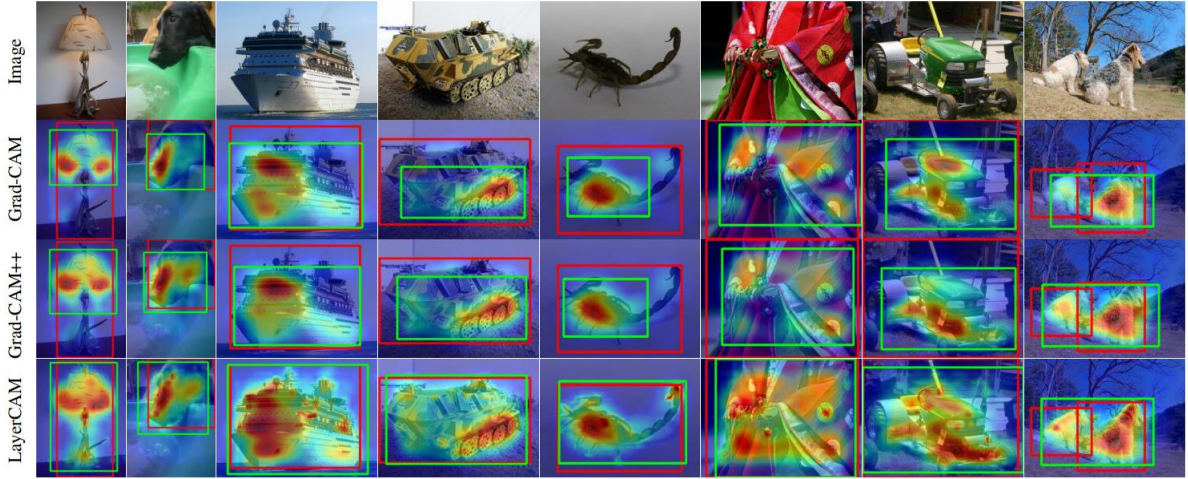


Figure 6 LayerCAM compared to Grad-CAM and Grad-CAM++, taken from (Jiang et al., 2015).

In our report, we will implement the original CAM model, together with the ReCAM and LayerCAM variant. Finally, we compare their results visually and eventually conclude the best out of the three models on the Flower-17 dataset.

Implementation

We implemented CAM, ReCAM and LayerCAM using Python with PyTorch as our deep learning framework. The feature extractor of a pretrained VGG16 are used in this implementation just like the LayerCAM implementation (Jiang et al., 2015), we merely change the last classifier to include a global average pooling layer. CAM and LayerCAM model use the same underlying architecture, the only difference of them is the methods of generating CAMs. On the other hand, ReCAM uses a slightly different architecture change as ReCAM is generated based on CAM. Therefore, we only implement models for CAM and ReCAM, while writing CAM generations as needed in the class methods. During training, we freeze the weights from the pretrained VGG16 feature extractor, while only train the last and customized part of the model.

We select the Flower-17 dataset from Kaggle (Kamal, 2021), it contains only 1360 images, within which has 80 images for each class. To make the image classification more robust, we perform offline augmentation on the dataset by a composition of random rotate from -20 to 20 degrees, random shift of 20% horizontally and vertically, and a 50% probability of horizontal flips. This increases our dataset to 13,600 images. We use the first out of three train and validation splits given from the Kaggle website. The original dataset has 1,360 images, where respectively 680, 340, 340 of them are train, valid, test dataset. After performing data augmentation, we use the 6,800 train images to train the model, then only use the 340 validation images to validate the model, this is because we believe the model shouldn't be tested on augmented validation images so that our result can be comparable to future studies that (might) use different augmentation settings. Since this is not a large project, we do not use the test dataset.

During training on the Flower-17 dataset, we use cross entropy loss function from PyTorch to track the model performance and save the model if there are improvements from the last best validation loss. We can afford to save a model for each epoch since only the last layer's weights need to be saved (around 37KB). Moreover, we train the model for 40 epochs using learning rate ranging from 0.00001 to 0.0002 with Adam optimizer. As a standard evaluation of image classification model, we calculate and report the Top1, Top3 and Top5 accuracies on the Flower-17 validation dataset and use it as a guide to choose the best model for CAM generation. This is because we notice a heavy overfitting starting from epoch 20

and it only becomes more serious until epoch 40, hence we figure the best model choosing strategy is to base on the top-k accuracies on validation set.

When generating CAM (or ReCAM) from a model, we often use the feature maps or the gradients of it, but its values are real values or nonnegative (if ReLU is applied). To make its value suitable for visualization, we tested different normalization methods. Minmax method is to take the minimum and maximum values of the map, then use it to scale the map into $[0,1]$. Sigmoid is another method that simply maps any number (negative or positive) to $[0,1]$. We also have a hybrid ReLU method that first apply ReLU to the map, then divides it by the maximum value so that final values are clipped into $[0,1]$. Out of the three methods, it is clear that ReLU might lead to loss of information. The three methods are generally only applied to CAM and ReCAM model, while LayerCAM model uses another new method since it inherently uses gradient-based method to generate CAM. With gradient at hand, it can be negative or non-negative. We use the standard approach as in GradCAM and LayerCAM (Jiang et al., 2015; Selvaraju et al., 2017) that first apply ReLU to the values of gradients so that only positive gradients are kept, divides it by its maximum value, double it, then apply tangent hyperbolic function to obtains a normalized values as in range of $[0,1]$. The formula is as below (Jiang et al., 2015):

$$\hat{M}^c = \tanh\left(\frac{2M^c}{\max(M^c)}\right)$$

The use of tangent hyperbolic function is to magnify small but positive activation, makes the effect more salient on our eyes.

We experiment with hyperparameter and model tuning. We create 4 CAM models respectively by training for 40 epochs with learning rate 0.00001, 0.00005, 0.0001, and 0.0002. Since the normalization method for creating CAMs is only applied during inference, we do not need to training CAM models separately for different normalization methods. However, ReCAM requires creating CAM during training, hence we trained 12 ReCAM models respectively similarly as above, but including the three normalization methods (Minmax, Sigmoid and ReLU hybrid) into the grid. For LayerCAM, we just reuse the 4 vanilla CAM models since the gradient-based method can also be applied during inference stage. We report our results in the next section.

The code used in this report will be shown in Appendix.

Discussion

Classification Metrics

Below shows the training result with hyperparameter tuning on the Flower-17 dataset (Table 2). We report on validation dataset, the best top-1 accuracy and the other corresponding top-k accuracies and cross entropy loss based on that epoch. To avoid taking results from overfitted models, we discard any models that have training loss less than half of validation loss. Again, we should emphasize that CAM only uses normalizing method during inference, so there is only one set of CAM models. On the other hand, ReCAM may use different normalizing methods during training process, hence model weights will be affected by the chosen normalizing method.

Model/LR	1e-5	5e-5	1e-4	2e-4
CAM	(epoch 40) Top1: 81.47% Top3: 96.18% Top5: 99.12% Loss: 1.003636	(epoch 37) Top1: 92.65% Top3: 98.53% Top5: 99.71% Loss: 0.288743	(epoch 22) Top1: 92.94% Top3: 99.12% Top5: 99.71% Loss: 0.271448	(epoch 11) Top1: 92.06% Top3: 98.82% Top5: 99.41% Loss: 0.282351
ReCAM (minmax)	(epoch 40) Top1: 77.06% Top3: 91.18% Top5: 96.18% Loss: 1.367373	(epoch 39) Top1: 87.65% Top3: 96.76% Top5: 99.12% Loss: 0.457615	(epoch 25) Top1: 89.41% Top3: 97.35% Top5: 99.41% Loss: 0.349892	(epoch 12) Top1: 90.29% Top3: 98.53% Top5: 99.12% Loss: 0.370260
ReCAM (sigmoid)	(epoch 40) Top1: 79.12% Top3: 92.94% Top5: 97.65% Loss: 1.052183	(epoch 40) Top1: 90.59% Top3: 99.12% Top5: 100% Loss: 0.289081	(epoch 18) Top1: 88.53% Top3: 97.94% Top5: 99.12% Loss: 0.373524	(epoch 9) Top1: 88.24% Top3: 97.35% Top5: 99.12% Loss: 0.366691
ReCAM (relu hybrid)	(epoch 40) Top1: 75.88% Top3: 89.12% Top5: 94.41% Loss: 1.457482	(epoch 39) Top1: 88.82% Top3: 97.65% Top5: 99.12% Loss: 0.381621	(epoch 24) Top1: 90.00% Top3: 98.24% Top5: 98.82% Loss: 0.365495	(epoch 14) Top1: 89.71% Top3: 97.94% Top5: 98.82% Loss: 0.341185

Table 2 Training results

Critical Analysis on Classification Metrics

By evaluating the models solely as classifiers, we notice a learning rate of $1e-5$ trains too slowly, while a learning rate of $2e-4$ seems to lead to mildly unstable result. The best of our models roughly lies on those which uses a learning rate of $5e-5$ or $1e-4$. After training these models, we notice a consistent trend that the vanilla CAM models always outperform the ReCAM models, regardless of normalizing methods, in which we will discuss the reason soon. Nonetheless, out of the three normalizing methods for training ReCAM models, the sigmoid method seems to perform slightly better than the other two, we believe it is because sigmoid maps values to the range $[0,1]$ consistently unlike minmax method which its range might heavily affected by the raw minimum and maximum values. Moreover, ReLU hybrid method perform slightly worse than sigmoid mostly because it discards some useful information (negative raw values) hence hinder the classification power.

CAM generations

We compare the generated CAMs from vanilla CAM, ReCAM and LayerCAM models. We select four trained models based on overall best accuracies and losses. Therefore, for CAM and LayerCAM we will choose the CAM model trained with a learning rate of $1e-4$. For ReCAM, we select models trained with learning rates $2e-4$, $5e-5$, $1e-4$ on its minmax, sigmoid and relu hybrid variants respectively. For this purpose, we name the models as CAM_main=[CAM_1e-4], ReCAM_1=[ReCAM_2e-4_minmax], ReCAM_2=[ReCAM_5e-5_sigmoid], ReCAM_3=[ReCAM_1e-4_relu]. We also show the graph of training and validation losses for these four models (Figure 7). We plot a vertical line when overfitting occur, so the models obtained after that are discarded.

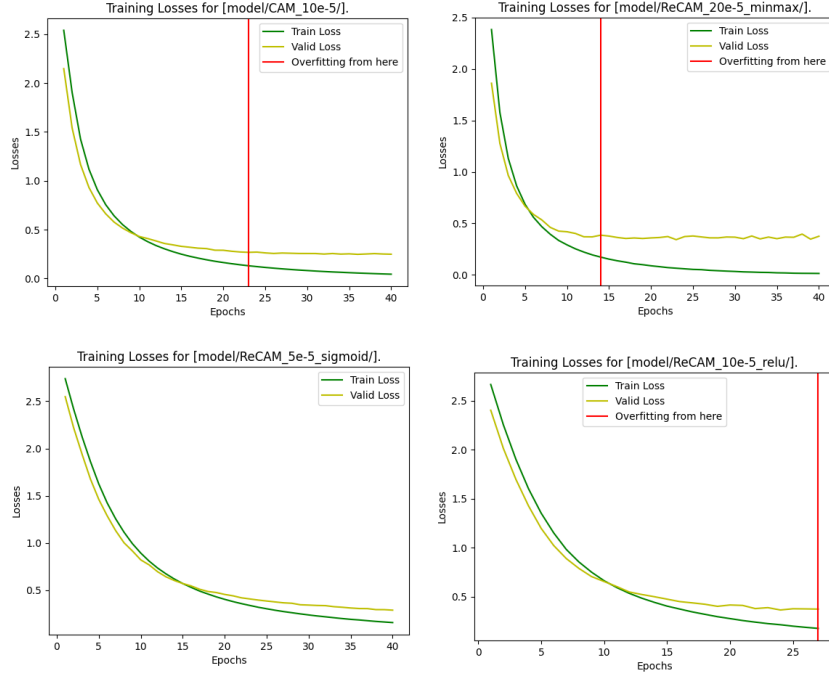


Figure 7 Losses for selected models, notice overfitting on larger learning rates. We didn't train ReCAM_Relu for 40 epochs because we recently change the training logic so that it will halt immediately during overfitting.

To qualitatively examine the goodness of CAM, we randomly choose 12 images from the validation set. After normalizing the activation map into range $[0,1]$, we post-process the CAMs by setting all values that are less than 0.3 to be zeros, then amplify the values by multiplying them with 1.5, then clamp the value with a maximum of 1. This completely removes the low activation values while emphasize the higher activation values for visualization purpose. In the LayerCAM setting, we did a little bit different than (Peng-Tao et al., 2015): They create CAM by averaging the gradients from 3rd, 4th and 5th feature maps, but we found that for Flower-17 dataset particularly, using the weights of 0.4 for 3rd feature map, 0.4 for 4th feature map, and 0.2 for 5th feature map gives better visual result, as we think that emphasizing the shallow features gives better details on the CAM generations. Figure 8 shows CAM generations using the weights from their respective predicted classes.

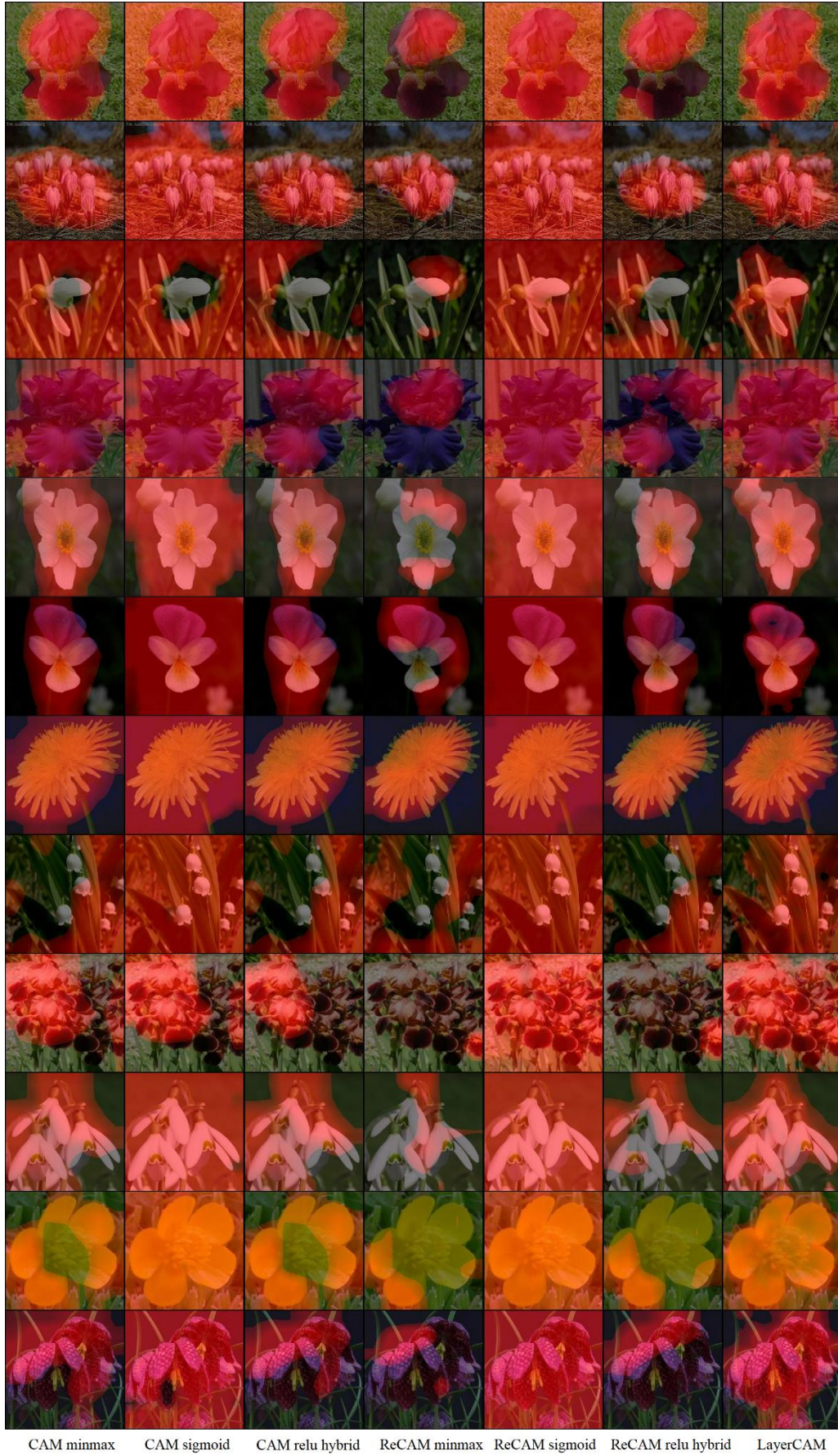


Figure 8 CAM comparisons from different model implementations.

Critical Analysis on CAM generations

From the comparisons, we found that the sigmoid normalization seems overkill on showing the region of interests, which doesn't provide any useful information. On CAM models, minmax and relu hybrid works reasonably well, but sometimes its CAM covers the region of interest only partially: It is more obvious on the last second row where the center of the yellow flower is not activated. Surprisingly, we obtained worse result on the ReCAM models regardless of normalizing methods. We believe since ReCAM generating CAM based on a previously generated CAM, if the first generated CAM doesn't cover the full region of interest, then it can only re-generate CAM based on the limited region covered before. Finally, LayerCAM generates objectively better CAMs than the other implementations. We can notice that it only covers the exact region of interest while deactivated on the background, and it does cover all the region of interests unlike ReCAM which only covers them partially.

To conclude, it is fair to say LayerCAM performs the best on visualizing activation maps, while ReCAM generate CAMs with the worst quality in general.

Conclusion

This report studies the decision visualization of an image classification model by generating activation maps for the predicted class (CAMs) on the Flower-17 dataset. This dataset contains limited images for 17 classes of flowers, some classes are highly similar which poses itself as a challenging classification task. To address this issue, we apply transfer learning to this problem by using a pretrained VGG16 model as our feature extractor, together with data augmentation that increase image count 10-fold, we obtain very good visualization using various CAM models.

We examined three different models that generate CAM using either different methods (CAM vs ReCAM) or different coefficient settings (heuristic vs gradient). The vanilla CAM model generate a good enough map, while ReCAM have subpar quality. However, we notice more importantly that the only gradient-based model, LayerCAM, generates very accurate CAM that exactly capture the region of interests. Thus, this report shows that gradient-based methods are clearly superior than heuristic coefficient methods, which might be the preferred way of visualizing such classifier since mathematically speaking, the region with higher gradient should indeed be a region that activates the decision and thus more relevant. Future research can include region of interest in other fields (such as audio and text) so that the model can explain its decision, either highlight the audio segment that contributes the most to a sentiment classification, or focus on the part of text that consists major part of any classification objective.

References

- Chattopadhyay, A., Sarkar, A., Howlader, P., & Balasubramanian, V. N. (2018). Grad-CAM++: Improved Visual Explanations for Deep Convolutional Networks. *2018 IEEE*, 839–847. <https://doi.org/10.1109/WACV.2018.00097>
- Chen, Z., Wang, T., Wu, X., Hua, X.-S., Zhang, H., & Sun, Q. (2022). Class Re-Activation Maps for Weakly-Supervised Semantic Segmentation. *ArXiv Preprint*.
<https://doi.org/10.48550/arXiv.2203.00962>
- Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., & Fei-Fei, L. (2009). ImageNet: A large-scale hierarchical image database. *IEEE Conference on Computer Vision and Pattern Recognition*, 248–255. <https://doi.org/10.1109/CVPR.2009.5206848>
- He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep Residual Learning for Image Recognition. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 770–778. <https://doi.org/10.48550/arXiv.1512.03385>
- Jiang, P.-T., Zhang, C.-B., Hou, Q., Cheng, M.-M., & Wei, Y. (2015). LayerCAM: Exploring Hierarchical Class Activation Maps for Localization. *Journal of Latex Class Files*, 14(8), 1–14. <https://doi.org/10.1109/TIP.2021.3089943>
- Jung, H., & Oh, Y. (2021). *Towards Better Explanations of Class Activation Mapping*.
<https://doi.org/10.48550/arXiv.2102.05228>
- Kamal, S. (2021). *17 Category Flower Dataset* (Version 1) [Data set].
<https://www.kaggle.com/datasets/sanikamal/17-category-flower-dataset>
- Lin, M., Chen, Q., & Yan, S. (2014). *Network in Network*.
<https://doi.org/10.48550/arXiv.1312.4400>
- Liu, Y., Lian, L., Zhang, E., Xu, L., Xiao, C., Zhong, X., Li, F., Jiang, B., Dong, Y., Ma, L., Huang, Q., Xu, M., Zhang, Y., Yu, D., Yan, C., & Qin, P. (2022). Mixed-UNet: Refined class activation mapping for weakly-supervised semantic segmentation with multi-scale inference. *Frontiers in Computer Science*, 4. <https://doi.org/10.3389/fcomp.2022.1036934>
- Patro, B. N., Lunayach, M., Patel, S., & Namboodiri, V. P. (2019). U-CAM: Visual Explanation using Uncertainty based Class Activation Maps. *IEEE/CVF International*

Conference on Computer Vision (ICCV), 7444–7453.

<https://doi.org/10.48550/arXiv.1908.06306>

Selvaraju, R. R., Cogswell, M., Das, A., Vedantam, R., Parikh, D., & Batra, D. (2017). Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization. *2017 IEEE*, 618–626. <https://doi.org/10.1109/ICCV.2017.74>

Wang, C.-Y., Bochkovskiy, A., & Liao, H.-Y. M. (2022). YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors. *ArXiv Preprint*. <https://doi.org/10.48550/arXiv.2207.02696>

Wightman, R., Touvron, H., & Jégou, H. (2021). ResNet strikes back: An improved training procedure in timm. *ArXiv Preprint*. <https://doi.org/10.48550/arXiv.2110.00476>

Zhou, B., Khosla, A., Lapedriza, A., Oliva, A., & Torralba, A. (2015). Learning Deep Features for Discriminative Localization. *Arxiv*. <https://doi.org/10.48550/arXiv.1512.04150>

Appendix

Code

The implementation for this assignment is written in Python, in particular using PyTorch, an efficient deep learning framework. We embrace object orientated programming concept, making several python scripts and utilize function for repeatable code for easy maintenance instead of using a single Jupyter Notebook. This is because we think the scale of this project is too big to be written in a single Jupyter Notebook.

After downloading dataset and other supporting files (please refer to the README file), we perform offline data augmentation to bump dataset count from 1,360 to 13,600 (Figure 9). For each image, we keep its original copy and create 9 others by random rotations of 20 degrees (clockwise and counter-clockwise), followed by random shifts of 20% width and 20% height, lastly by a 50% probability of horizontal flip to ensure generalization.

```
4 import os
5 import torchvision
6 from tqdm import tqdm
7 from torchvision import transforms
8 from torchvision.io import read_image
9
10
11 def augment(img_path):
12     # Augment a single image. It will apply resize 256,256 altogether.
13     # This will be saved in data/auged/.
14     os.makedirs("./data/auged/", exist_ok=True)
15     img_base = os.path.basename(img_path)
16     targets = [os.path.join("./data/auged/", img_base.replace(".jpg", f"_{i}.jpg")) for i in range(10)]
17
18     img_tensor = read_image(img_path)
19     img_256 = transforms.Resize((256,256))(img_tensor) / 255.
20     # Augmentations
21     transform = transforms.Compose([
22         transforms.RandomAffine(20, (0.2, 0.2)), # rotate and shift
23         transforms.RandomHorizontalFlip(p=0.5), # horizontal flip
24     ])
25     # Apply transform
26     # First image is original image.
27     torchvision.utils.save_image(img_256, targets[0])
28     for i in range(1, 10):
29         auged_img = transform(img_256)
30         # 2nd to 10th images are augmented.
31         torchvision.utils.save_image(auged_img, targets[i])
32
33 if __name__ == "__main__":
34     print("Performing offline image augmentation.")
35     img_paths = [os.path.join("./data/jpg/", p) for p in os.listdir("./data/jpg/") if p.endswith(".jpg")]
36
37     for path in tqdm(img_paths):
38         augment(path)
```

Figure 9 Data Augmentation (augment.py)

Following PyTorch convention, when loading custom dataset we create a custom class for our Flower-17 dataset (Figure 10). This class has a custom `__len__` method to return total number of data, and a `__getitem__` method to support indexing. In this framework, the label of an image is inferred by its absolute position in the array, so there is no need to use another text file to refer to the flower's label.

```

40
49 class FlowerDataset(Dataset):
50     """Face Landmarks dataset."""
51
52     def __init__(self, data_root = "./data/auged/", transform=None):
53         """
54         Arguments:
55             transform (callable, optional): Optional transform to be applied
56             on a sample.
57         """
58         if IN_COLAB:
59             # Use native colab storage when using google colab
60             # since loading dataset from google drive to colab takes a long time.
61             self.data_root = "/content/data/auged/"
62         else:
63             self.data_root = data_root
64         if not os.path.isdir(data_root):
65             raise ValueError("Please run augment.py script before using dataset. ")
66         if transform is None:
67             # Default transform
68             self.transform = transforms.Compose([
69                 # Rescale((256,256)),
70                 ToTensor(),
71                 Normalize(mean = torch.tensor([0.485, 0.456, 0.406]),
72                 std = torch.tensor([0.229, 0.224, 0.225])),
73             ])
74         else:
75             self.transform = transform
76         self.image_paths = [f"image_{str(i+1).zfill(4)}_{j}.jpg" for i in range(1360) for j in range(10)]
77         self.data_len = len(self.image_paths)
78
79     def __len__(self):
80         return self.data_len
81
82     def __getitem__(self, idx):
83         if torch.is_tensor(idx):
84             idx = idx.tolist()
85         img_name = os.path.join(self.data_root, self.image_paths[idx])
86         image = io.imread(img_name)
87         image = np.float32(image)
88         label = np.floor(idx/800)
89         if self.transform:
90             image = self.transform(image)
91         return image, int(label)
92

```

Figure 10 PyTorch Custom Dataset (dataloader.py)

Now we can train model. Below is a snippet of code that accept user configuration and then train a model (Figure 11). Inside the code, we have `CAM_workflow` and `ReCAM_workflow` functions to train respective models. We put an early stopping criteria when validation loss is larger than twice of training loss so we do not store any overfitted models.

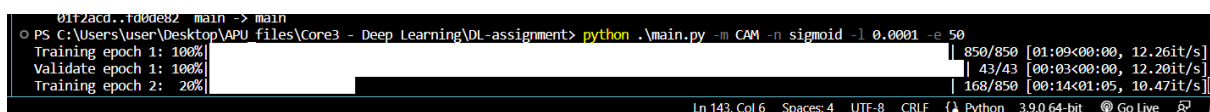
```

123 if __name__ == "__main__":
124     parser = argparse.ArgumentParser()
125     parser.add_argument("-m", "--model-type", help="Choose from 'CAM', 'ReCAM' or 'LayerCAM", type=str)
126     parser.add_argument("-n", "--normalize-by", help="Choose from 'minmax', 'sigmoid' or 'relu', defaults to 'minmax'.", type=str,
127                         default='minmax')
128     parser.add_argument("-l", "--lr", help="Learning rate, defaults to 1e-4.", default=1e-4, type=float)
129     parser.add_argument("-e", "--epochs", help="Number of epochs, defaults to 50.", default=50, type=int)
130     args = parser.parse_args()
131     os.makedirs("./model/", exist_ok = True)
132     timestamp = datetime.datetime.now().strftime("%d-%m-%Y_%H%M%S")
133     normalize_by = args.normalize_by
134     model_type = args.model_type
135
136     # No need to touch below
137     model_dir = f"./model/TS{timestamp}_{model_type}/"
138     os.makedirs(model_dir)
139
140     training_kwargs = {
141         "epochs": args.epochs,
142         "lr": args.lr,
143         "normalize_by": args.normalize_by,
144     }
145     # Save keywords
146     with open(os.path.join(model_dir, "config.json"), "w") as f:
147         f.write(json.dumps(training_kwargs, indent=4))
148
149     filename = os.path.join(model_dir, "training_logs.txt")
150     # Record model meta
151     with open(filename, "w") as f:
152         f.write(f"Model type: [{model_type}], using [{normalize_by}] normalizing method.\n")
153     # Redirect output to the file
154     sys.stdout = open(filename, "a")
155     if model_type == "CAM":
156         CAM_workflow(model_dir, **training_kwargs)
157     elif model_type == "ReCAM":
158         ReCAM_workflow(model_dir, **training_kwargs)
159     # Restore output to default.
160     sys.stdout.close()
161     sys.stdout = sys.__stdout__

```

Figure 11 Training Code (main.py)

The code can be used by providing model type, normalization method, learning rate and epochs. We provide an example using the command [python .\main.py -m CAM -n sigmoid -l 0.0001 -e 50] as below, and training will begin (Figure 12). Model will be saved on epochs that has improved validation loss compared to history. The code also redirect console output so that model type (such as CAM, ReCAM, LayerCAM) and training process is recorded into a log file.



```

01f2acd...f0bde82 main -> main
PS C:\Users\User\Desktop\APU files\Core3 - Deep Learning\DL-assignment> python .\main.py -m CAM -n sigmoid -l 0.0001 -e 50
Training epoch 1: 100% | 850/850 [01:09<00:00, 12.26it/s]
Validate epoch 1: 100% | 43/43 [00:03<00:00, 12.20it/s]
Training epoch 2: 20% | 168/850 [00:14<01:05, 10.47it/s]
Ln 143, Col 6 Spaces: 4 UTF-8 CRLF Python 3.9.0 64-bit Go Live

```

Figure 12 Training console output

After training completes, we can evaluate the models as typical classifiers. Here we use a function that evaluates topk accuracies, taken from stackoverflow (Figure 13). This function will be used in the main evaluate function that calculate top1, top3 and top5 accuracies of a model on the validation dataset (Figure 14).

```

8 def accuracy(output: torch.Tensor, target: torch.Tensor, topk=(1,)):
9     """...
29     with torch.no_grad():
30         # ---- get the topk most likely labels according to your model
31         # get the largest k \in [n_classes] (i.e. the number of most likely probabilities we will use)
32         maxk = max(topk) # max number labels we will consider in the right choices for our model
33         batch_size = target.size(0)
34
35         # get top maxk indicies that correspond to the most likely probability scores
36         # (note _ means we don't care about the actual top maxk scores just their corresponding indicies/labels)
37         _, y_pred = output.topk(k=maxk, dim=1) # _, [B, n_classes] -> [B, maxk]
38         y_pred = y_pred.t() # [B, maxk] -> [maxk, B] Expects input to be <= 2-D tensor and transposes dimensions 0 and 1.
39
40         # - get the credit for each example if the model's predictions is in maxk values (main crux of code)
41         # for any example, the model will get credit if it's prediction matches the ground truth
42         # for each example we compare if the model's best prediction matches the truth. If yes we get an entry of 1.
43         # if the k'th top answer of the model matches the truth we get 1.
44         # Note: this for any example in batch we can only ever get 1 match (so we never overestimate accuracy <1)
45         target_resaped = target.view(1, -1).expand_as(y_pred) # [B] -> [B, 1] -> [maxk, B]
46         # compare every topk's model prediction with the ground truth & give credit if any matches the ground truth
47         correct = (y_pred == target_resaped) # [maxk, B] were for each example we know which topk prediction matched truth
48         # original: correct = pred.eq(target.view(1, -1).expand_as(pred))
49
50         # -- get topk accuracy
51         list_topk_accs = [] # idx is topk1, topk2, ... etc
52         for k in topk:
53             # get tensor of which topk answer was right
54             ind_which_topk_matched_truth = correct[:k] # [maxk, B] -> [k, B]
55             # flatten it to help compute if we got it correct for each example in batch
56             flattened_indicator_which_topk_matched_truth = ind_which_topk_matched_truth.reshape(-1).float() # [k, B] -> [kB]
57             # get if we got it right for any of our top k prediction for each example in batch
58             tot_correct_topk = flattened_indicator_which_topk_matched_truth.float().sum(dim=0, keepdim=True) # [kB] -> [1]
59             # compute topk accuracy - the accuracy of the model's ability to get it right within it's top k guesses/preds
60             topk_acc = tot_correct_topk / batch_size # topk accuracy for entire batch
61             list_topk_accs.append(topk_acc)
62         return list_topk_accs # list of topk accuracies for entire batch [topk1, topk2, ... etc]
63

```

Figure 13 Top-k accuracy function (evaluate.py)

```

64 def evaluate(model):
65     # Given a model, load weights from model_path and perform evaluation.
66     data = get_dataloaders()["valid"]
67     total_data = len(data.dataset)
68     num_batch = len(data)
69     train_iter = iter(data)
70     accuracies = {
71         "top1": 0,
72         "top3": 0,
73         "top5": 0,
74     }
75     for i in tqdm(range(num_batch), desc=f"Evaluating: "):
76         bimg, blabel = next(train_iter)
77         bimg, blabel = bimg.to(DEVICE), blabel.to(DEVICE)
78         B = blabel.shape[0]
79         prediction = model(bimg)
80         top1, top3, top5 = accuracy(prediction, blabel, topk = (1,3,5))
81         accuracies["top1"] += int(top1.item()) * B
82         accuracies["top3"] += int(top3.item()) * B
83         accuracies["top5"] += int(top5.item()) * B
84     accuracies = {key: round(value/total_data, 4) for key, value in accuracies.items()}
85     return accuracies
86

```

Figure 14 Evaluation of a model (evaluate.py)

Finally, in the evaluate.py file, we accept model directory name and evaluate on all saved models, then store the model accuracies into a JSON file for future references (Figure 15). For instance, running `[python .\evaluate.py -d model/ReCAM_20e-5_sigmoid]` will evaluate all epochs model, and Figure 16 shows the performance of the ReCAM model trained with relu hybrid normalization method on epoch 10 to 15.

```

87 if __name__ == "__main__":
88     parser = argparse.ArgumentParser()
89     parser.add_argument("-d", "--model-dir", type=str)
90     # Model type will be inferred by training logs
91     args = parser.parse_args()
92     model_dir = args.model_dir
93     # Infer model type
94     model = None
95     with open(os.path.join(model_dir, "training_logs.txt"), "r") as f:
96         first_line = f.readline()
97         print(first_line)
98         if "[CAM]" in first_line:
99             model = baseCAM()
100         elif "[ReCAM]" in first_line:
101             if "[sigmoid]" in first_line:
102                 model = ReCAM("sigmoid")
103             elif "[relu]" in first_line:
104                 model = ReCAM("relu")
105             elif "[minmax]" in first_line:
106                 model = ReCAM("minmax")
107     model.eval()
108     model.to(DEVICE)
109     # Initialize json file
110     eval_out = os.path.join(model_dir, "eval.json")
111     # Start evaluating
112     evals = {}
113     for ind in range(100):
114         model_path = os.path.join(model_dir, f"epoch_{ind+1}.pth")
115         if not os.path.isfile(model_path):
116             continue
117         print(f"On model from epoch {ind+1}...")
118         model = partial_load_state_dict(model, model_path)
119         accuracies = evaluate(model)
120         evals[f"epoch_{ind+1}"] = copy(accuracies)
121     with open(eval_out, "w") as f:
122         json.dump(evals, f, indent=4)

```

Figure 15 Evaluate model then save the accuracies (evaluate.py)

```

model > ReCAM_20e-5_relu > {} eval.json > {} epoch_10
44     "top3": 0.9735,
45     "top5": 0.9882
46 },
47 "epoch_10": {
48     "top1": 0.8706,
49     "top3": 0.9676,
50     "top5": 0.9882
51 },
52 "epoch_11": {
53     "top1": 0.8765,
54     "top3": 0.9706,
55     "top5": 0.9882
56 },
57 "epoch_12": {
58     "top1": 0.8912,
59     "top3": 0.9765,
60     "top5": 0.9853
61 },
62 "epoch_14": {
63     "top1": 0.8971,
64     "top3": 0.9794,
65     "top5": 0.9882
66 },
67 "epoch_15": {
68     "top1": 0.8912,
69     "top3": 0.9794,
70     "top5": 0.9882
71 }
72 }

```

Figure 16 Accuracies on the ReCAM ReLU hybrid model with learning rate set as 0.0002.

Finally, we can perform inference on the model to generate CAMs and overlay them on original images (Figure 17, Figure 18, Figure 19). User must provide model directory, model type, normalization method (only applies to CAM and ReCAM) and an optional seed for reproducibility. The reason for using seed is because we will always randomly choose 12 images from validation set for visualization purpose, hence choosing seed can make the choice of these 12 images be fixed. To make CAM more visible, we blackout the activations below 0.3, then multiply all values by 1.5 before clamping the values back to 0 to 1.

```

102 def inference_workflow(model_path, model_type, normalize_by, save_as, image_title, seed=-1):
103     if model_type in ["CAM", "SingleLayerCAM", "LayerCAM"]:
104         model = baseCAM(normalize_by).to(DEVICE)
105     elif model_type == "ReCAM":
106         model = ReCAM(normalize_by).to(DEVICE)
107     model = partial_load_state_dict(model, model_path).to(DEVICE)
108     model.eval()
109     batch_input, batch_label = get_random_batch(valid_dataset, seed=seed)
110     B = batch_input.shape[0]
111     batch_input = batch_input.to(DEVICE)
112     predictions = model(batch_input)
113     # Create comparison labels
114     info = create_comparison(batch_label, predictions)
115
116     # Test CAM
117     if model_type == "CAM":
118         cams = model.get_cam(batch_input)
119     elif model_type == "ReCAM":
120         cams = model.get_recam(batch_input)
121     # elif model_type == "SingleLayerCAM":
122     #     cams = model.get_gradient_cam(batch_input)
123     elif model_type == "LayerCAM":
124         cams = model.get_detail_cam(batch_input)
125     # Post-process CAMs for better visualization
126     cams = postprocess(cams)
127     # Convert to pil then overlap heatmaps
128     inputs = unnormalize(batch_input)
129     overlapped = []
130     for ind in range(B):
131         # print(inputs[ind].min(), inputs[ind].max())
132         # print(heatmaps[ind].min(), heatmaps[ind].max())
133         img = TF.to_pil_image(inputs[ind])
134         h_img = TF.to_pil_image(cams[ind]) # Emphasize heatmap
135         res = Image.blend(img, h_img, 0.5)
136         res = transforms.ToTensor()(res)
137         overlapped.append(torch.clone(res))
138     overlapped = torch.stack(overlapped)
139     save_as_grids(overlapped, info, image_title, save_as)
140     save_as_compact(overlapped, save_as.replace(".jpg", "_compact.jpg"))
141     save_as_compact(overlapped, save_as.replace(".jpg", "_strip.jpg"), nrow=1)
142     print(f"Model inference result saved as [{os.path.abspath(save_as)}].")
143

```

Figure 17 Inference workflow (inference.py)

```

87 def postprocess(cams):
88     cams[cams<0.3] = 0
89     cams = torch.clamp(1.5*cams, max=1)
90     return cams

```

Figure 18 CAM postprocessing (inference.py)

```

144 if __name__ == "__main__":
145     os.makedirs("./output/", exist_ok=True)
146     parser = argparse.ArgumentParser()
147     parser.add_argument("-m", "--model-path", type=str)
148     parser.add_argument("-t", "--model-type", type=str, default="CAM") # Can also be ReCAM
149     parser.add_argument("-n", "--normalize-by", type=str, default="minmax")
150     parser.add_argument("-s", "--seed", type=int, default=-1)
151     args = parser.parse_args()
152     timestamp = datetime.datetime.now().strftime("%d-%m-%Y_%H%M%S")
153     image_output = os.path.join("./output/", f"{args.model_type}_{timestamp}.jpg")
154     # Only use valid dataset
155     d = FlowerDataset()
156     strain, stest, sval = load_splits(split=1)
157     valid_dataset = Subset(d, sval)
158     # Set image title
159     image_title = f"[{args.model_type}] Model"
160     if args.model_type != "LayerCAM":
161         # Specify normalizing method for non-LayerCAM model
162         image_title += f", normalized by [{args.normalize_by}]"
163     # Start
164     inference_workflow(args.model_path, args.model_type,
165                       normalize_by = args.normalize_by, save_as = image_output,
166                       image_title = image_title, seed = args.seed)
167
168
169

```

Figure 19 Accepting user input (inference.py)

From the code from inference workflow, we store the model CAM output in 3 versions: With labels, compact grid with shape 3 by 4, and compact strip. For example, the output below will be generate when running the command [python .\inference.py -m model\CAM_10e-5\epoch_22.pth -t LayerCAM -s 1090] (Figure 20). We used the strip format to compare different CAM models in Figure 8 (For this, we provide commands in the [inference_commands.bat] file from our github repository).

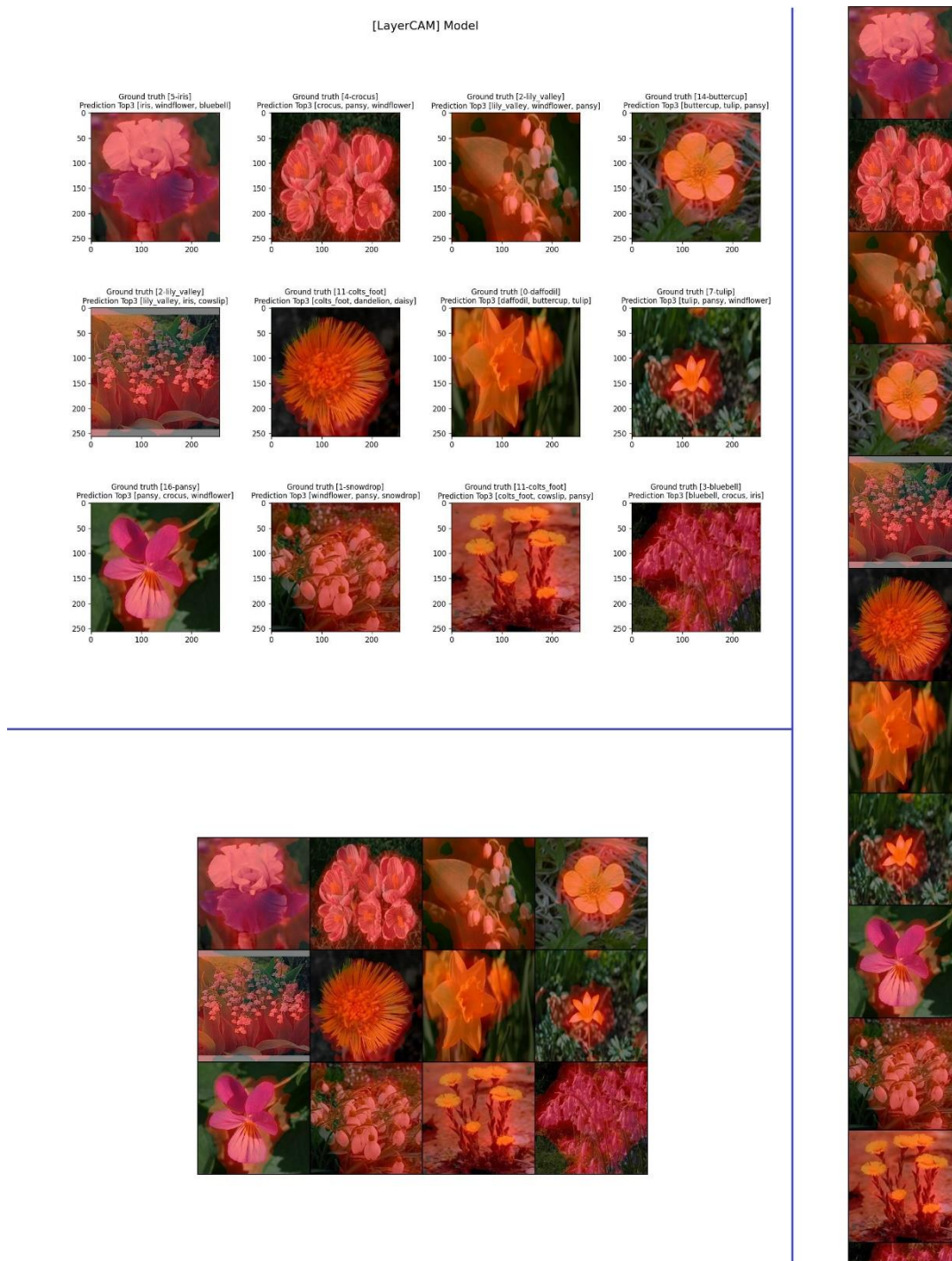


Figure 20 Inference output (inference.py). Top-left: CAM with top-3 classification results. Bottom-left: CAM only. Right: CAM only, in strip form.