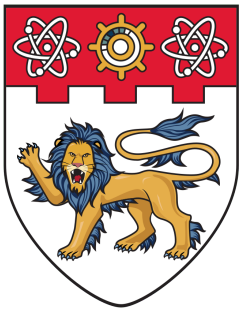

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

SCSE22-0205

Visualizing Interpretations of Deep Neural Networks

Author	Ta Quynh Nga
Matriculation Number	U1920601L
Supervisor	A/P Li Boyang
Examiner	Ms Liu Siyuan

Submitted in Partial Fulfilment of the Requirements for the
Degree of Bachelor of Science in Data Science and Artificial Intelligence
of the Nanyang Technological University

School of Computer Science and Engineering

2023

Abstract

The evolution of Convolutional Neural Networks and new approaches like Vision Transformers has led to better performance in computer vision. However, deep neural networks lack transparency and interpretability, leading to consequences in critical applications. Visualizing deep neural network interpretations can provide insights into decision-making, identify biases and errors, and reveal potential limitations in the model or training data. This area of research is significant for enhancing the transparency, interpretability, and trustworthiness of deep neural networks and facilitating their application in critical domains. This project aims to create a web application to facilitate the interpretation of the ConvNeXt model, a state-of-the-art convolutional neural network. The application implements three techniques: Maximally activating image patches, Feature attribution visualisation with SmoothGrad, and Adversarial perturbation visualisation with SmoothGrad. Maximally activating image patches help users understand what patterns maximally activate a channel in a layer. Feature attribution visualisation with SmoothGrad highlights the pixels that are most influential for the model's prediction. Adversarial perturbation visualisation with SmoothGrad allows users to explore how the model reacts when the input image is perturbed. The results of experimentation of interpretability techniques on the ConvNeXt model will also be discussed in this report.

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Associate Professor Li Boyang, for his invaluable guidance, encouragement, and support throughout this project. His expertise and insights have been instrumental in shaping the direction of this work and ensuring its success.

I also extend my thanks to my examiner, Ms Liu Siyuan, for her time and effort in reviewing this project.

I would like to thank my family and friends for their constant support and encouragement. I am also grateful to all those who provided feedback and suggestions during the course of this project.

Finally, I would like to thank the school for providing me with valuable lessons and knowledge that have helped me to develop my skills and capabilities.

Table of Contents

List of Figures

1 Introduction

1.1. Background

Convolutional Neural Networks (ConvNets) have revolutionized computer vision since their inception in 2010, leading to significant advancements in accuracy, efficiency, and scalability. Some of the most representative ConvNets models include VGGNet, Inception, ResNe(X)t, DenseNet, MobileNet, EfficientNet, and RegNet, each designed to address specific challenges in image recognition (Liu et al., 2022).

Despite the success of ConvNets, the field of computer vision continued to evolve, leading to the development of Vision Transformers (ViT) in 2020. As a transformer model, ViT showed remarkable potential in achieving high accuracy with larger model and dataset sizes. However, ViT faced challenges in generic computer vision tasks such as object detection and semantic segmentation (Liu et al., 2022).

The emergence of Swin Transformer (Liu et al., 2021) demonstrated the value of transformers as a generic vision backbone, outperforming ConvNets and achieving outstanding performance in a wide range of computer vision tasks. Nonetheless, the success of this approach was primarily attributed to Transformers' inherent superiority rather than the intrinsic inductive biases of convolutions.

In 2022, Liu et al. proposed a pure convolutional model dubbed ConvNeXt, which was designed by modernizing a standard ResNet (He et al., 2016) towards the design of Vision Transformers. ConvNeXt demonstrated remarkable performance, surpassing even Vision Transformers in certain computer vision tasks (Liu et al., 2022).

The rapid evolution of ConvNets and the emergence of new approaches such as Vision Transformers and ConvNeXt highlight the continuous pursuit of better performance in computer vision. This evolution is expected to bring about new breakthroughs and advancements in the field, pushing the limits of what is possible in computer vision tasks. However, the lack of transparency and interpretability in deep neural networks like image classification models can have significant consequences, particularly in critical applications such as healthcare and autonomous vehicles. In these applications, the decisions made by the neural network models must be explainable, trustworthy, and free from biases.

Visualizations of deep neural network interpretations can improve the transparency and interpretability of these models by providing insights into how the model arrived at its decision. By visualizing deep neural network interpretations, we can gain a better understanding of which features or patterns in the data the model is focusing on and how it is combining them to make decisions. Moreover, interpretating of deep neural networks can help researchers to identify biases and errors in the model, see which features the model is overemphasizing or underemphasizing, which can reveal potential biases or limitations in the training data or the model architecture. Therefore, this area of research has significant importance in advancing the development and application of deep neural networks, enhancing the transparency, interpretability, and trustworthiness of these models, and facilitating their application in critical domains.

1.2. Literature Review: Interpretability Techniques

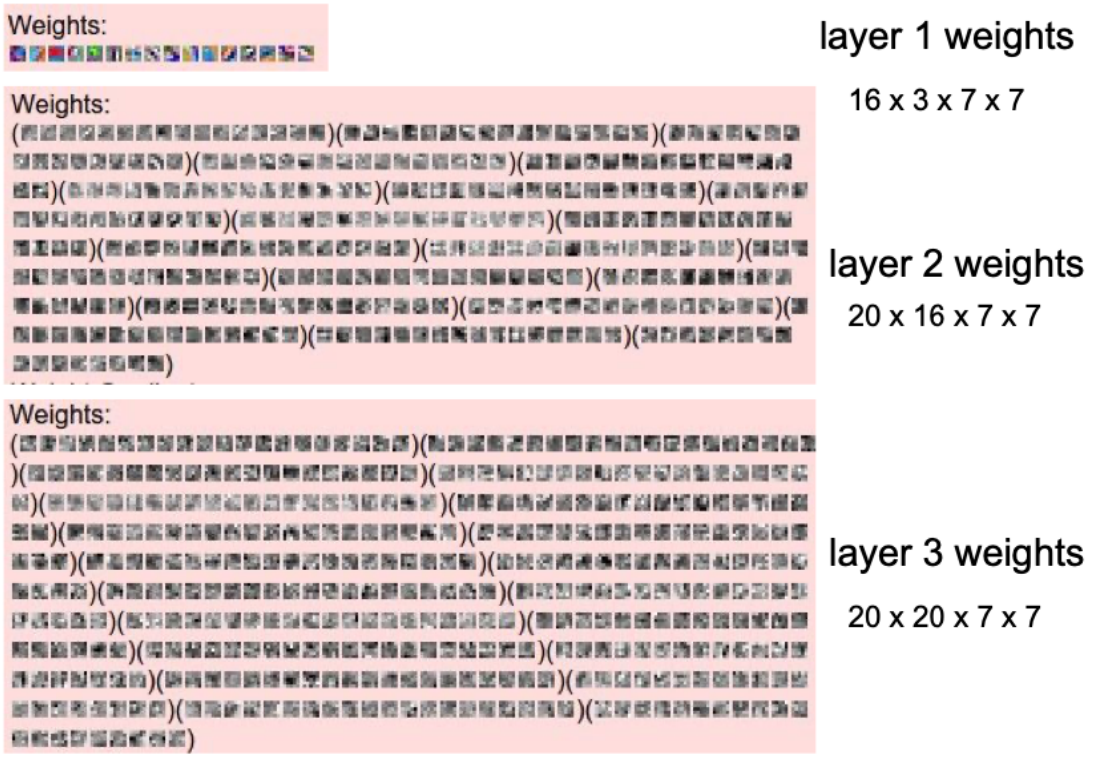

Figure 1: Visualize the filters/kernels (raw weights) of layer 1 (top), layer 2 (middle) and layer 3's weights (bottom) of CIFAR-10. Taken from ConvNetJS CIFAR-10 demo

Figure 2: L2 nearest neighbors (from second columns afterwards) in feature space of each test image (first column). Via Krizhevsky, Sutskever, & Hinton, 2012.

Figure 3: Map 4096-dimensional vectors to 2D using t-SNE ("T-SNE Visualization of CNN Codes," n.d.)

Among the basic interpretability techniques, visualizing raw weights, nearest neighbors, and maximally activating patches are commonly used. Visualizing raw weights can help us visualize filters at layers of the learned network by mapping every value of weights into grayscale colors (see Figure 1), while nearest neighbors analysis (see Figure 2 and Figure 3) can help us understand how the network groups similar input data together. Additionally, visualizing maximally activating image patches (see Figure 4) can help us identify the parts of an input image that are most important for that filter to make decisions. These basic interpretability techniques can be

used to gain insights into the behavior of deep neural networks and guide the development of more advanced interpretation methods.

Figure 4: Visualization of patterns learned by layer conv6 (top) and layer conv9 (bottom) of the network trained on ImageNet. Each row corresponds to one filter. Via Springenberg, Dosovitskiy, Brox, and Riedmiller (2015)

More advanced interpretation techniques involve computing saliency maps to identify regions of an image that are most important for classification. This can be done through occlusion techniques or gradient-

based attribution such as Deconvolution (Zeiler & Fergus, 2014), Guided back-propagation (Springenberg et al., 2015), Grad-cam (Selvaraju et al., 2016, see Figure 5). In real-world scenarios, these techniques often seem to identify regions that are meaningful to human eyes, according to Smilkov et al. (2017). However, the sensitivity maps generated by these techniques can also be visually noisy, highlighting seemingly random pixels that do not contribute much to the final decision. To address this issue, Smilkov et al. (2017) have developed a method called SmoothGrad, which aims to reduce the visual noise in saliency maps (see Figure 6). This method has been shown to produce saliency maps that are easier to interpret, while still identifying the key regions that contribute to the final decision.

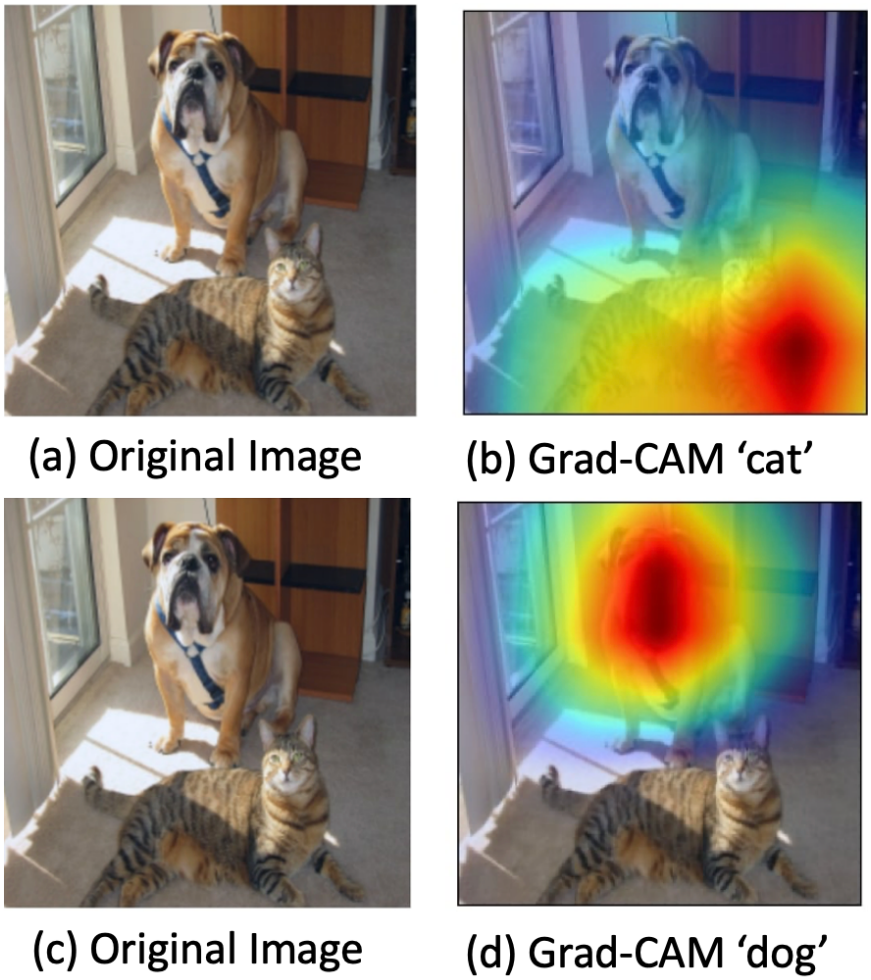

Figure 5: (a, c): original image with cat and dog. (b, d): Grad-CAM on 'cat' and 'dog' respectively. Via Selvaraju et al. (2017)

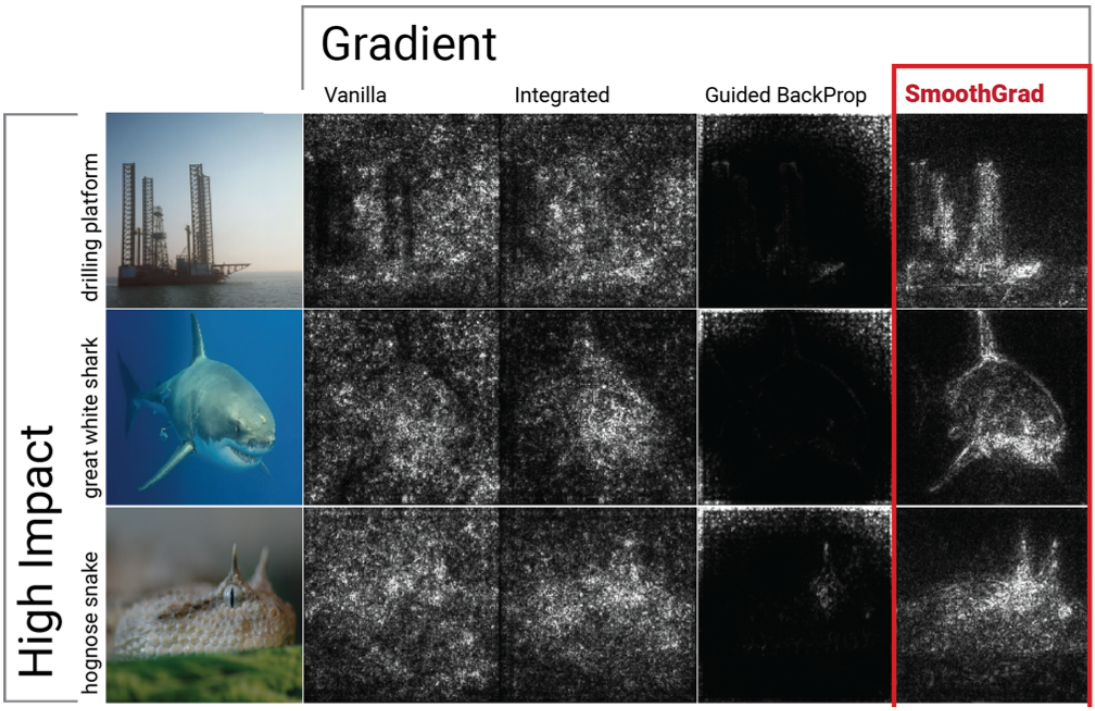

Figure 6: Qualitative evaluation of different methods including Vanilla, Integrated, Guided BackProp and SmoothGrad. Via Smilkov, Thorat, Kim, Viégas, and Wattenberg (2017)

In addition to feature attribution techniques, data attribution methods like HyDRA can also play a significant role in interpreting deep neural networks. HyDRA (Chen et al., 2021), which stands for Hypergradient for Data Relevance Analysis, is designed to provide insights into how individual data points affect a model's predictions. By analyzing the full trajectory of a training example, HyDRA can help determine the influence of that example on the model's decision, shedding light on the role of individual data points in shaping the model's behavior. This method can be particularly helpful in improving the interpretability of deep learning models, by providing a more complete understanding of the factors that contribute to their decisions.

1.3. Objective and scope

This project aims to interpret ConvNeXt model by utilizing two techniques: visualizing maximally activating patches and SmoothGrad saliency maps. The maximally activating image patches method facilitating the inquiry of

"What patterns maximally activated this filter (channel) in this layer?" would be discussed in Section 3.1 below. Meanwhile, the SmoothGrad method facilitating the inquiry of "Which features are responsible for the current prediction?" would be discussed in Section 3.2.

In addition to visualizing interpretations of ConvNeXt, we also employed SmoothGrad saliency maps on ResNet (He et al., 2016) and MobileNet (Howard et al., 2017). This allowed for a comprehensive understanding of the interpretability of ConvNeXt in comparison to other models in the computer vision domain.

Furthermore, we aimed to explore the vulnerability of ConvNeXt to adversarial attacks through the application of Smoothgrad visualization (Section 3.3). Finally, we developed a web application[1][2] for the public that allows for easy visualization of the interpretations of ConvNeXt and investigate the model further.

Due to resource and time constraints, this project focuses on the tiny version of ConvNeXt ("Hugging Face," 2022a), which was trained on ImageNet-1k ("Datasets at Hugging Face," n.d.) at a resolution of 224x224. For the sake of comparison, ResNet and MobileNet models of the same size are considered as well, which are ResNet-50 ("Hugging Face," 2022b) and MobileNetV2 ("PyTorch," n.d.).

Moreover, the current version of the web application enables users to apply visualization techniques solely on the ImageNet-1k validation set, which contains 50,000 images, and user-defined images are not yet supported.

[1] https://huggingface.co/spaces/taquynhnga/CNNs-interpretation-visualization
[2] https://github.com/taquynhnga2001/CNNs-interpretation-visualization

2 Project Work

2.1. Project schedule

The project was conducted throughout the academic year 2022-2023, starting from August 2022 and ending in May 2023. The project schedule is illustrated in the Gantt chart as shown in Figure 7 to Figure 11 below.

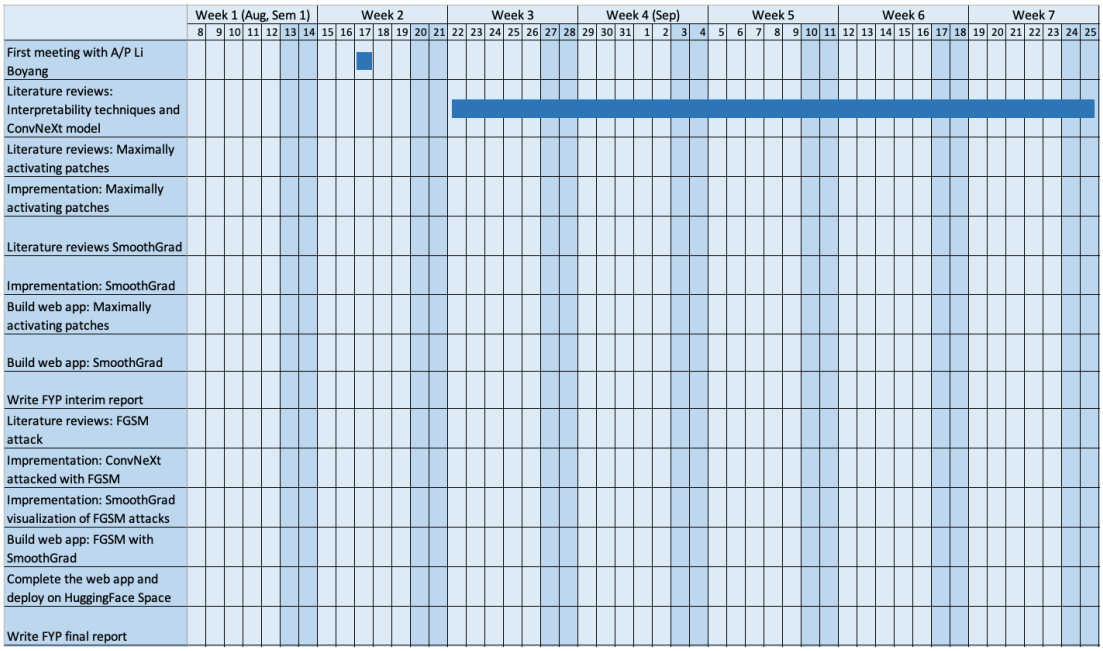

Figure 7: Gantt chart of project schedule from Week 1 Semester 1 to Week 7 Semester 1

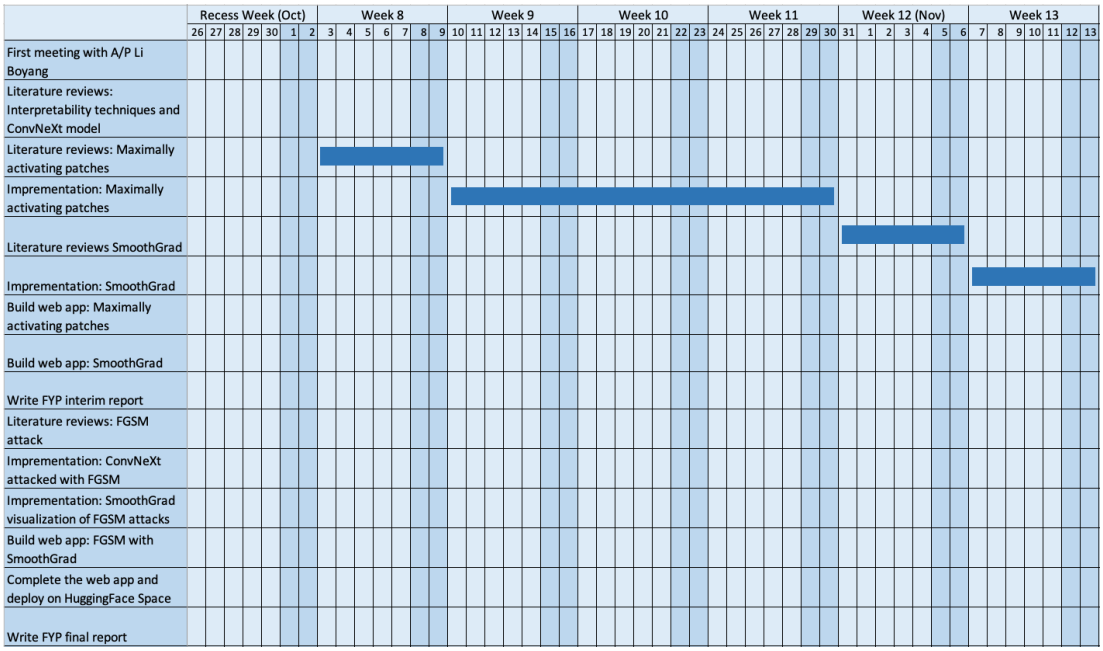

Figure 8: Gantt chart of project schedule from Recess week Semester 1 to Week 13 Semester 1

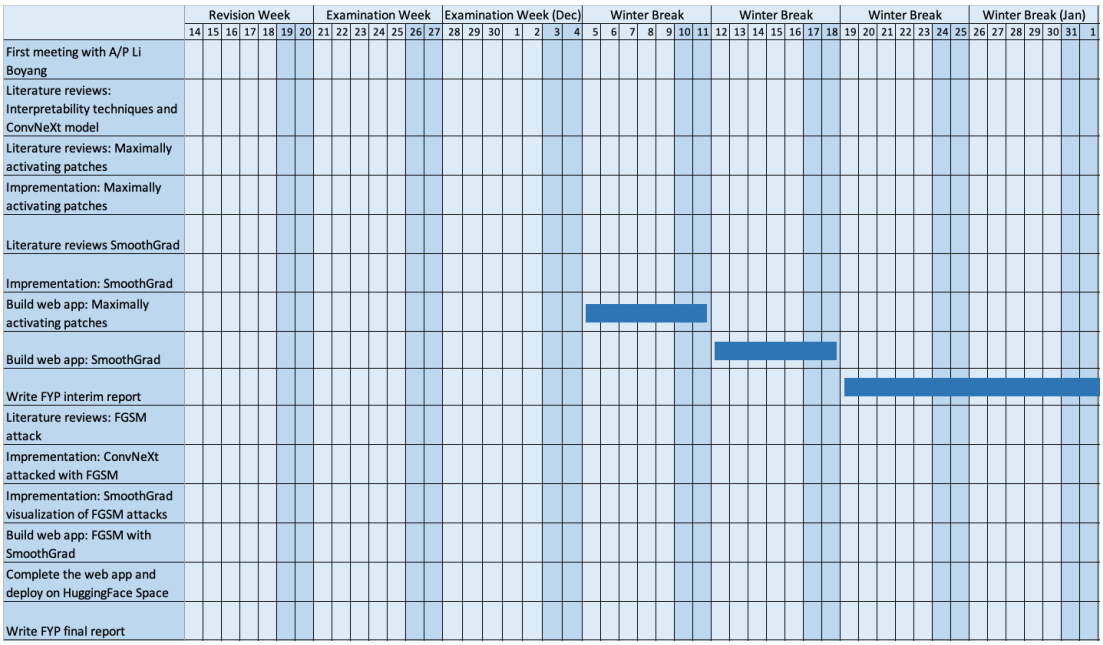

Figure 9: Gantt chart of project schedule from Revision week Semester 1 to Winter break

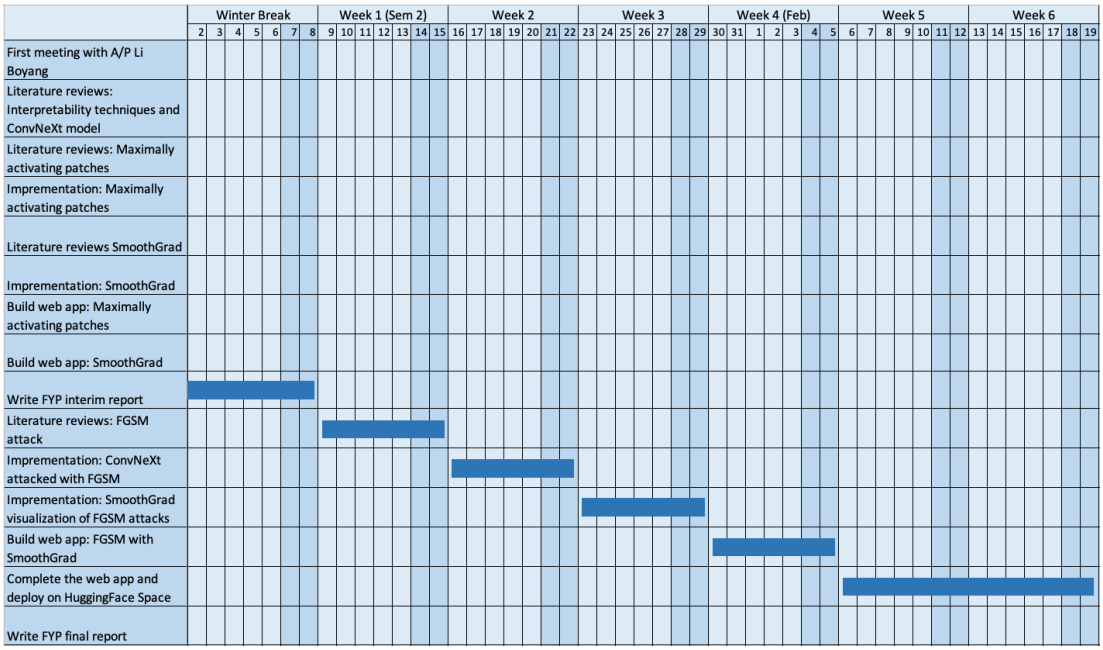

Figure 10: Gantt chart of project schedule from Winter break to Week 6 Semester 2

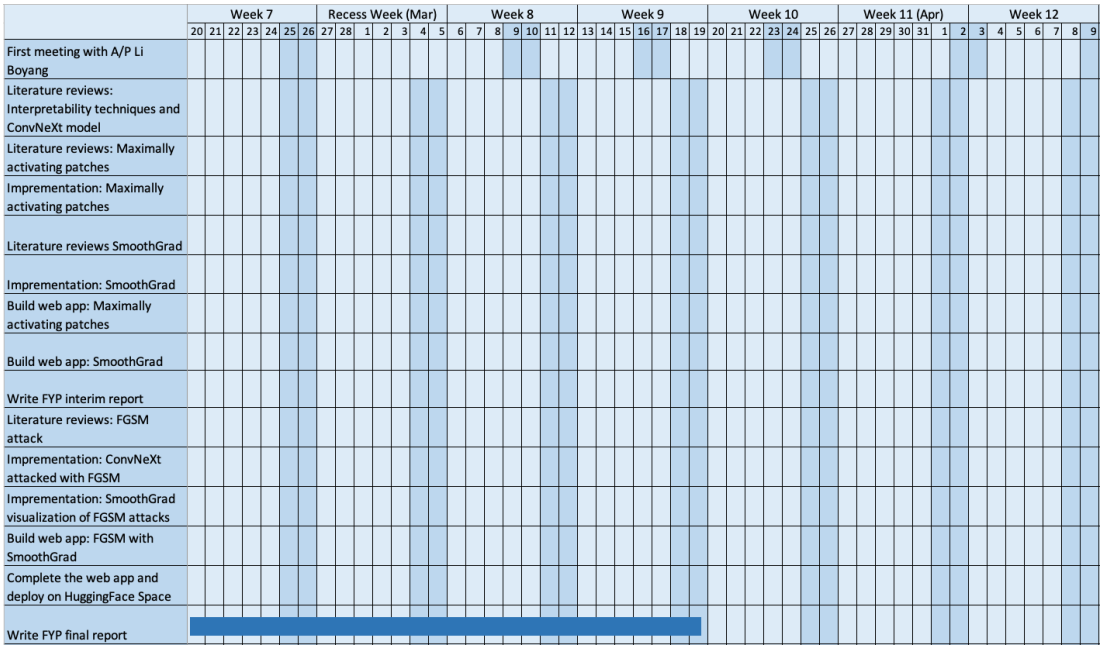

Figure 11: Gantt chart of project schedule from Week 7 Semester 2 to Week 12 Semester 2

2.2. Resources

The project was conducted on a variety of resources to enable the efficient execution and deployment of the visualization techniques and web application. The initial development and testing were performed on Kaggle ("Kaggle: Your Machine Learning and Data Science Community," n.d.),

utilizing both CPU and GPU resources to infer and evaluate the ConvNeXt model. The web application was built using Streamlit ("Streamlit - the Fastest Way to Build and Share Data Apps," n.d.), a Python framework for creating interactive web applications, providing a user-friendly interface for users to explore the visualizations. Finally, the web application was deployed on HuggingFace Spaces ("Spaces - Hugging Face," n.d.), which provided a virtual environment with 16GB RAM, 2 CPU cores, and 50GB of (not persistent) disk space for easy access and use by the public.

3 Implementation

This section will provide a detailed description of the implementation of interpretability techniques implemented in the project. Specifically, Section 3.1 will delve into the Maximally Activating Image Patches visualization method, while Section 3.2 will focus on Feature Attribution Visualization with SmoothGrad. Lastly, Section 3.3 will cover Adversarial Perturbation Visualization with SmoothGrad. Each section will cover the motivation behind the technique, how it was implemented, and instructions on how to use it on the web app.

3.1. Maximally activating image patches

The motivation behind the Maximally Activating Image Patches visualization technique is to investigate the specific patterns or features that maximally activate a filter or channel in a given layer of a deep neural network. In other words, the technique aims to answer the question of "what patterns maximally activated this channel?" by visualizing the most activating image patches. This approach provides insight into the learned representations and feature selectivity of the network, allowing for a more fine-grained understanding of the inner workings of the model. Through the identification of maximally activating patches, we can gain understanding into the types of patterns or features that the model has learned to attend to in order to make its predictions. This information can then be used to guide model improvement or to validate existing hypotheses about the model's behaviour.

The goal of this technique is to find the top-k image patches that have the highest activations in a given channel. The steps of the algorithm are outlined in Algorithm 1 below.

Algorithm 1: Find top-k image patches maximize activation of a given channel C

Input: k, channel C, image set, batch size

Output: top-k image patches

1. Generate batches of image from image set with specific batch size.
2. Feed the image batches iteratively, store the top-k activations of channel C until the current batch and store their coordinates of the activation (x, y) in the feature map.

   activations_list = empty list
   For each batch in batches:
   - 2.1. **Feed** the batch into the model.
   - 2.2. **Add** the activations of channel C of this batch to activations_list and store their coordinates (x, y) in feature map.
   - 2.3. **Sort** the activations_list and save only top-k activations item.

3. Calculate the receptive fields of each item in **activations_list** from their coordinates (x, y) in feature map. Receptive fields are specified by the coordinates of the top left (x1, y1) and bottom right (x2, y2) pixels in the original image.

Algorithm 1: Find maximally activating image patches

The Algorithm 1 above finds the image patches that maximize activation of a given channel in a particular layer. Feeding images into the model in step 2.1 requires the process saving the activation values, therefore, we used the $register\_forward\_hook(hook)$ function[3] to save the activation values in each interation (see code snippets in Appendix A in Figure 27).

The sorting in step 2.3 helps to reduce the extra memory for storing numerous redundant the activation values not in top-k in the entire image set. However, in practice, feeding 50,000 images into the ConvNeXt model, sorting, and then calculating the receptive fields of top-k activation patches

[3] https://pytorch.org/docs/stable/generated/torch.nn.Module.html#torch.nn.Module.register_forward_hook

is an expensive process. Thus, to ensure a seamless user experience on our web application, we pre-computed the output of the aforementioned process for all channels in all layers. This allows users to select a specific layer and value of k, and the top-k maximally activating patches for all channels in the layer will be readily displayed on the screen, eliminating the long waiting time while running the process for a specific channel.

In step 3 of our algorithm, we employed a modified version of the method presented in Hien's (2018) Medium blog to calculate the receptive field and obtain the coordinates of the maximally activating patches in the original image. The algorithm involved determining the center of the receptive field and its size in the original image, as depicted in Figure 12 and Figure 13.

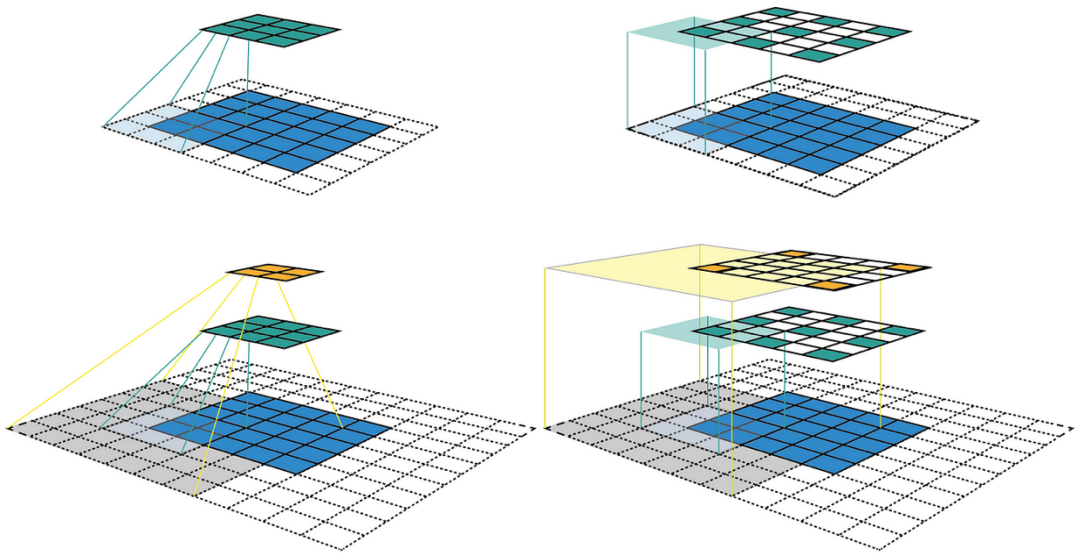

Figure 12: "Two ways to visualize CNN feature maps. In all cases, we uses the convolution C with kernel size k = 3x3, padding size p = 1x1, stride s = 2x2. (Top row) Applying the convolution on a 5x5 input map to produce the 3x3 green feature map. (Bottom row) Applying the same convolution on top of the green feature map to produce the 2x2 orange feature map. (Left column) The common way to visualize a CNN feature map. Only looking at the feature map, we do not know where a feature is looking at (the center location of its receptive field) and how big is that region (its receptive field size). It will be impossible to keep track of the receptive field information in a deep CNN. (Right column) The fixed-sized CNN feature map visualization, where the size of each feature map is fixed, and the feature is located at the center of its receptive field." Via Hien, D. H. (2018).

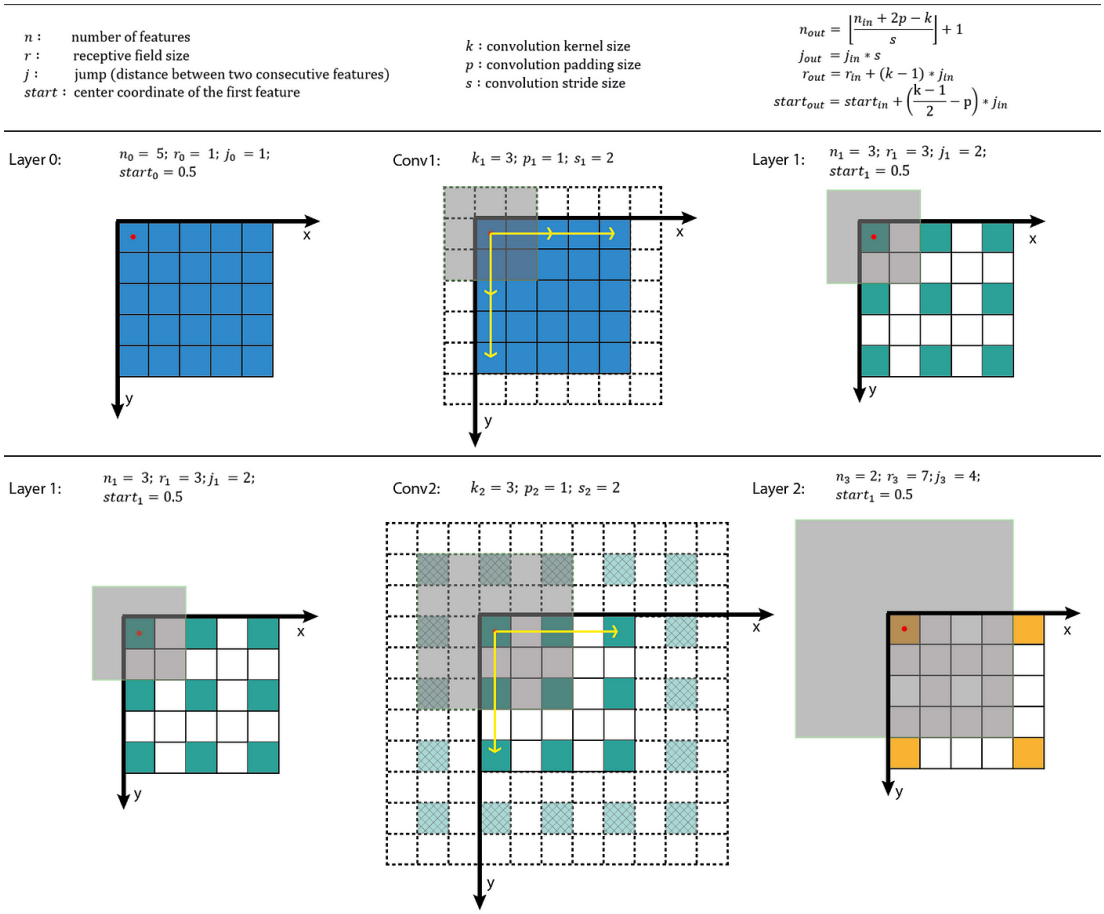

Figure 13: "Applying the receptive field calculation on the example given in Figure 12. The first row shows the notations and general equations, while the second and the last row shows the process of applying it to calculate the receptive field of the output layer given the input layer information". Via Hien, D. H. (2018).

In this project, we utilized DOT language to generate a visual representation of the ConvNeXt architecture (see Figure 28 for code snippet) on the web application. DOT ("DOT Language," 2022) is a graph description language that is commonly used for creating visual representations of graph structures. The generated DOT file was then converted into an SVG format (see more in Appendix A Figure 29 or access the web application to see the real sized version), which allowed us to add event listeners using JavaScript. Specifically, we added event listeners to the layer buttons on the SVG item so that users could click on a specific layer and see the top-*k* maximally activating patches for all channels in that layer. This interactive feature provides users with an intuitive and user-

friendly way of exploring and interpreting the ConvNeXt model (see Figure 14).

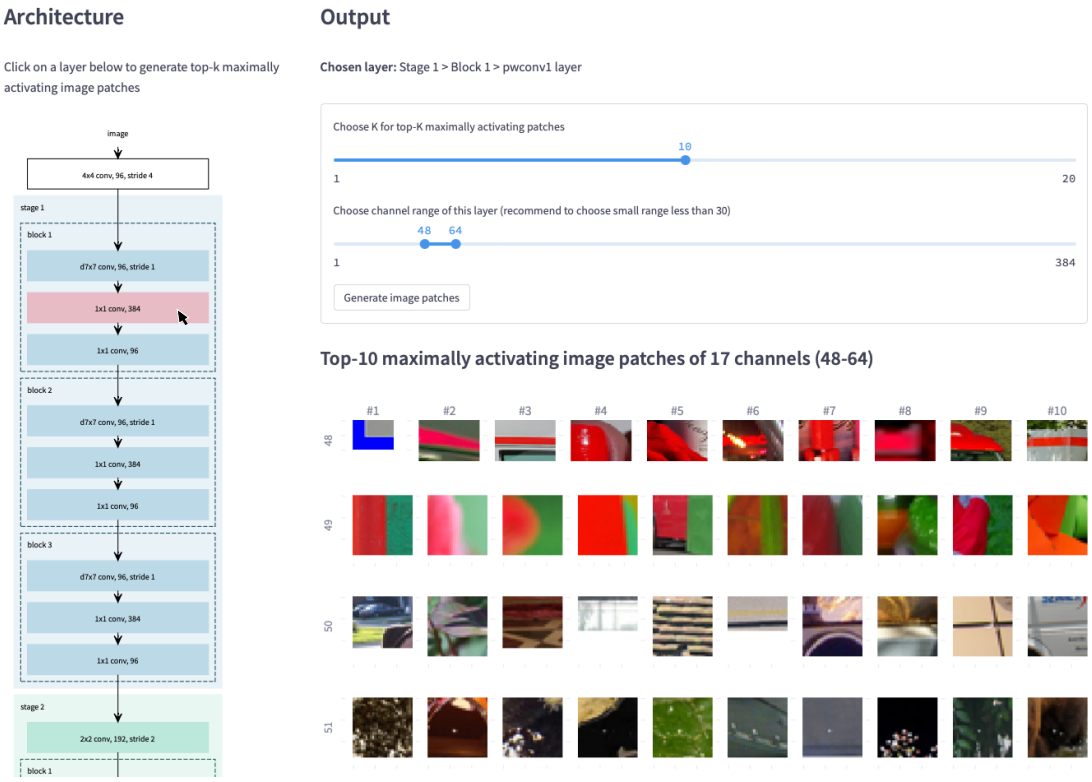

Figure 14: User Interface of Maximally activating patches technique of the web app

Figure 15: Getting information of the image patch (ranking value, activation value, class label, class id and image id) by hovering it

To use the Maximally Activating Image Patches visualization technique on the web application, users are required to select a specific layer on the left-hand side, where the ConvNeXt architecture is displayed. Subsequently, the user is prompted to choose the preferred number of patches, k, and select the desired channels in the selected layer to exhibit the top-k maximally activating patches. The output format of the resulting visualization is

similar to the representation shown in Figure 14. Further information pertaining to specific image patches can be obtained by hovering over them (see Figure 15). For more additional demonstration, please refer to the following video at https://youtu.be/yvpdxmxK3pY.

3.2. Feature attribution visualization with SmoothGrad

In the field of machine learning, it is often useful to identify the key input features, such as pixels in the case of images, that influence a model's prediction. In situations where the model makes an incorrect prediction, it becomes crucial to determine which features contributed to the error. To accomplish this, a feature importance mask can be generated, which is a grayscale image with the same dimensions as the original image. Each pixel's brightness in the mask corresponds to the importance of that feature in the model's prediction.

Several approaches exist to calculate an image sensitivity map for a particular prediction. One common technique is to utilize the gradient of a class prediction neuron in relation to the input pixels, indicating how the prediction is affected by small changes in pixel values. In other words, the techniques aim to answer the question of "which features are responsible for the current prediction of the model?" by visualizing the saliency maps. However, this method often generates a noisy mask. To address this issue, the SmoothGrad technique, proposed by Daniel et al. in their work titled "SmoothGrad: Removing noise by adding noise," adds Gaussian noise to multiple copies of the image and averages the resulting gradients to produce a smoother and less noisy mask.

In this project, we utilized the 'torchvex' package[4], which incorporates numerous saliency mapping techniques, including Class Activation Mapping (CAM), Gradient-weighted Class Activation Mapping (Grad-CAM), Simple Gradient, and SmoothGrad. To implement the SmoothGrad method into our ConvNeXt model, we made certain modifications to the existing code (see Figure 30). In addition, we also implemented the SmoothGrad technique on ResNet and MobileNet models to compare their effectiveness with the ConvNeXt model.

After generating the SmoothGrad saliency maps using the torchvex package for a selected image, we further enhance the interpretability of the generated maps by visualizing them in grayscale and overlaying them on top of the original image. To achieve this, we utilize the OpenCV[5] package, which provides a comprehensive suite of computer vision and image processing functions (see code snippet in Figure 31). The grayscale map highlights the relative importance of each pixel in the input image for the model's prediction, while the overlay allows for intuitive visualization of the specific regions that contribute to the prediction. The resulting visualizations provide a powerful tool for interpreting the behaviour of the model and identifying potential areas for improvement (see Figure 16).

[4] https://github.com/vlue-c/Visual-Explanation-Methods-PyTorch
[5] https://pypi.org/project/opencv-python/

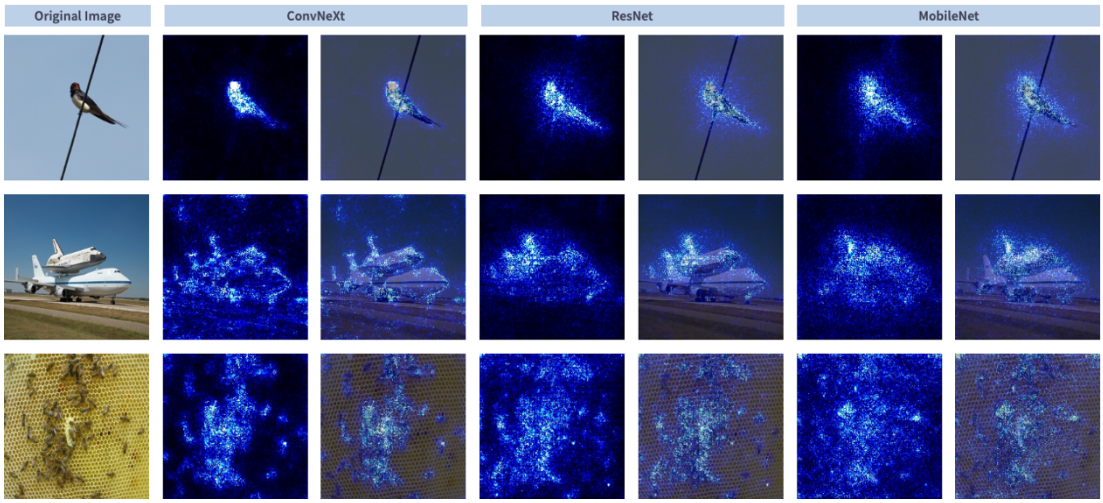

Figure 16: SmoothGrad maps interpretations of ConvNeXt, ResNet and MobileNet

In order to utilize the SmoothGrad visualisation technique on the web application, the user is required to input specific parameters (see Figure 17).

a) Firstly, the user must select the model(s) they wish to use. The available models include ConvNeXt, ResNet, and MobileNet, all of which have a similar number of parameters.
b) Secondly, the user must choose the type of Image set to use, either a User-defined set or a Random set. If the User-defined set is selected, the user must enter a list of image IDs separated by commas, such as '0,1,4,7' which corresponds to valid image IDs found in the ImageNet1k dataset. On the other hand, if the Random set is selected, the user needs to select the desired number of random images to display.
c) Once the user has entered the necessary parameters, the SmoothGrad visualisation technique will generate saliency maps for the chosen images, overlaying the map on the original image in grayscale.

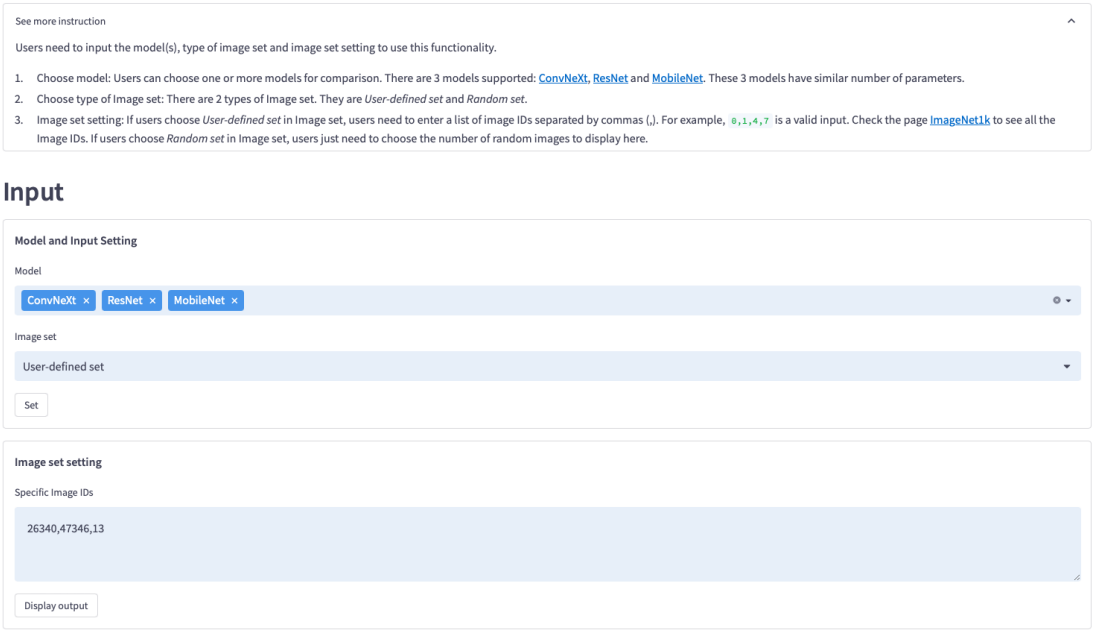

Figure 17: Instruction and Input fields to use SmoothGrad technique in the web app

For more additional demonstration, please refer to the following video at https://youtu.be/usek6JsmxCE.

3.3. Adversarial perturbation visualization with SmoothGrad

In the field of machine learning, adversarial examples are inputs that are deliberately designed to mislead neural networks into misclassifying a given input. Despite being indistinguishable by humans, these examples can cause the network to fail to recognize the actual image content. One of the most popular adversarial attacks is the Fast Gradient Sign Method (FGSM) attack, which is a white box attack that seeks to ensure misclassification by exploiting the gradient information of the targeted model. In a white box attack, the attacker has complete access to the model being attacked.

The FGSM attack is one of the earliest and most effective adversarial attacks, as described by Goodfellow et al. in their seminal work on Explaining and Harnessing Adversarial Examples. The attack utilizes the

gradients that neural networks use to learn, but instead of adjusting the weights based on the backpropagated gradients to minimize loss, it adjusts the input data to maximize the loss using the gradient of the loss with respect to the input data.

In this project, we sought to investigate the vulnerability of the ConvNeXt model to adversarial attacks by utilizing the SmoothGrad visualization technique. By generating saliency maps of adversarial examples crafted using the FGSM attack, we were able to visualize the regions of the input image that were responsible for the model's misclassification and answered the question: if we perform adversarial perturbation on an image, how does it affect the interpretations generated? (see code snippet in Figure 32 in Appendix C for FGSM implementation).

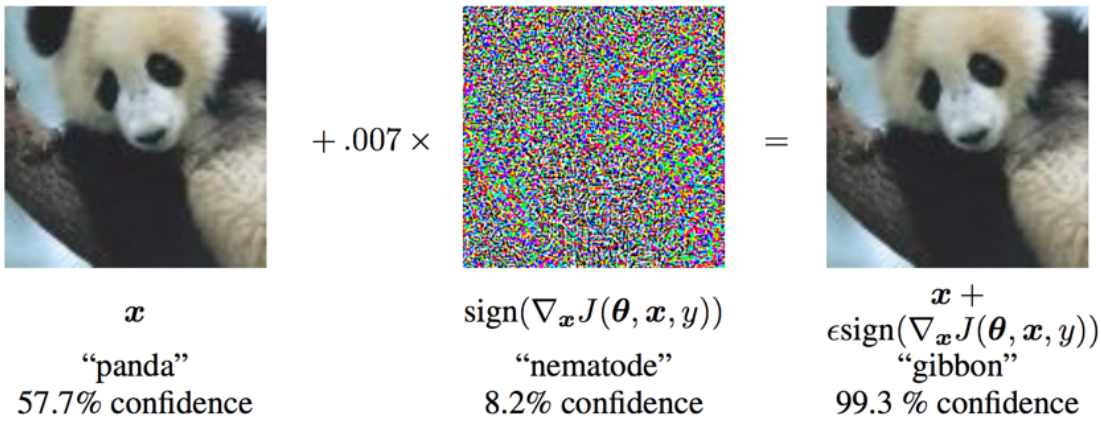

Figure 18: epsilon = 0.007 was used for the attack. Via Goodfellow, Shlens, and Szegedy (2015).

In the FGSM attack, *epsilon* represents the amount of perturbation added to the original image (see Figure 18). The larger the *epsilon*, the more noticeable the perturbation becomes in the resulting image. Adding a larger perturbation may also lead to the degradation of the model's accuracy, making the attack more effective in terms of causing misclassification. The choice of *epsilon* is therefore a trade-off between the strength of the attack and the visual quality of the resulting perturbed image. A small *epsilon* may result in a less noticeable perturbation but may

also be ineffective in fooling the model, while a large *epsilon* may result in a highly perturbed image but may also be easily detected by human observers. Therefore, it is important for users to choose an appropriate epsilon value based on their specific needs and goals.

Our web application provides options for users to choose a specific *epsilon* value or let the algorithm find the smallest *epsilon* automatically (if capable). The algorithm iteratively attacks the model through a set of *epsilon* values, increasing the value by a step variable after each iteration, until it finds the smallest *epsilon* value that can successfully cause the model to misclassify the image (see Figure 19) or stops when it reaches the maximum value of *epsilon* in the searching value sets (see Figure 21).

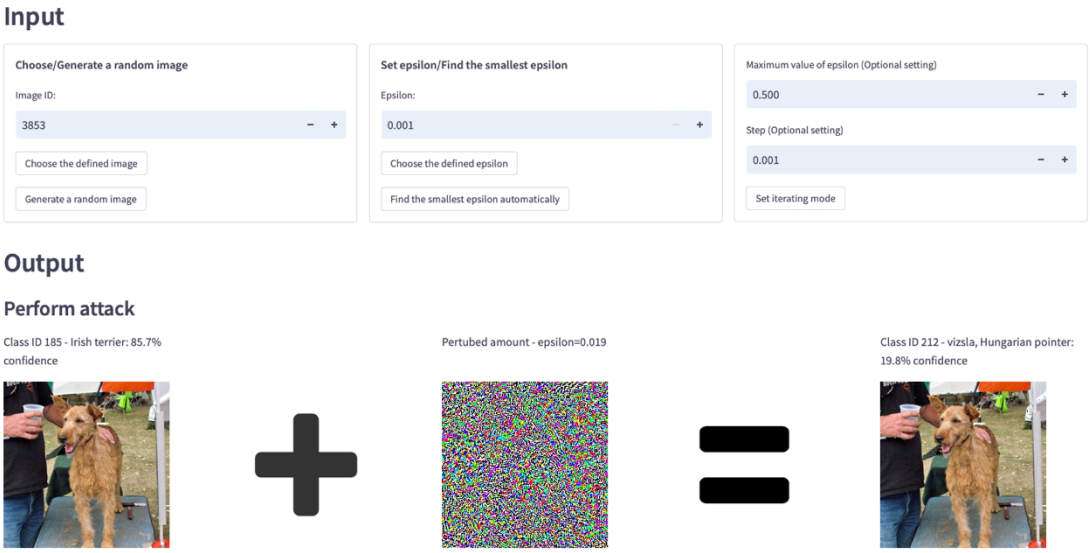

Figure 19: FGSM succeeded in attack the model at epsilon=0.019 after finding the smallest epsilon automatically

After successfully attacking the model using the chosen *epsilon* value, the web application will display the SmoothGrad visualisation generated for both the original image and the perturbed image (see Figure 20). This will allow users to compare the two images and observe the differences in the saliency maps for the two images. This comparison will provide insights into how the ConvNeXt model's interpretation changes due to adversarial perturbations.

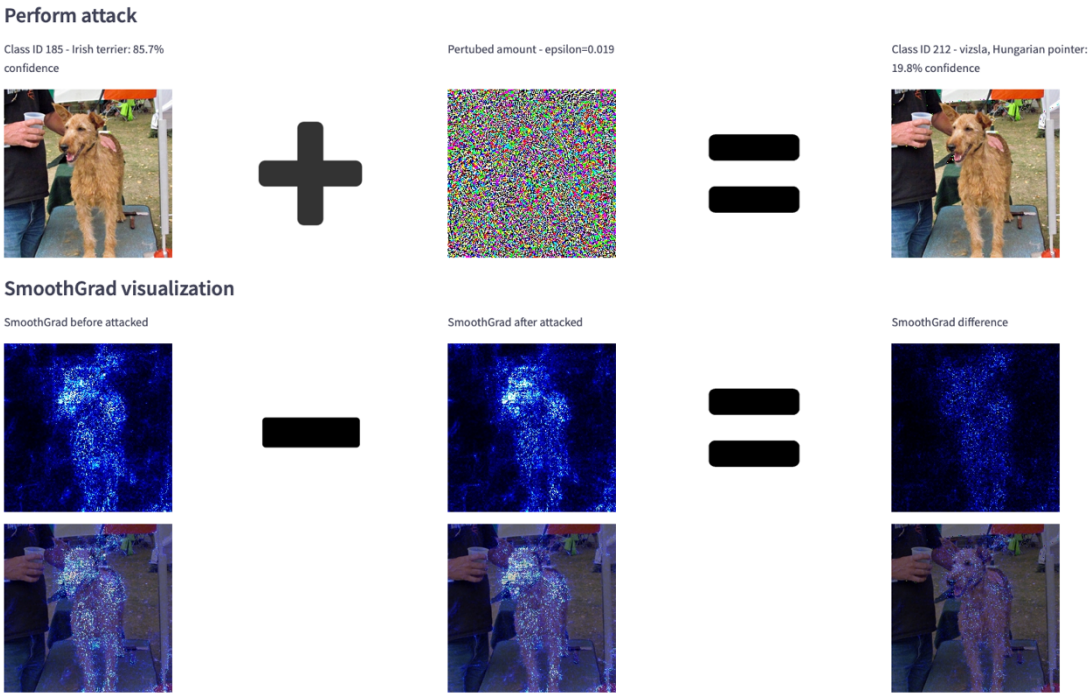

Figure 20: SmoothGrad visualization on perturbed image

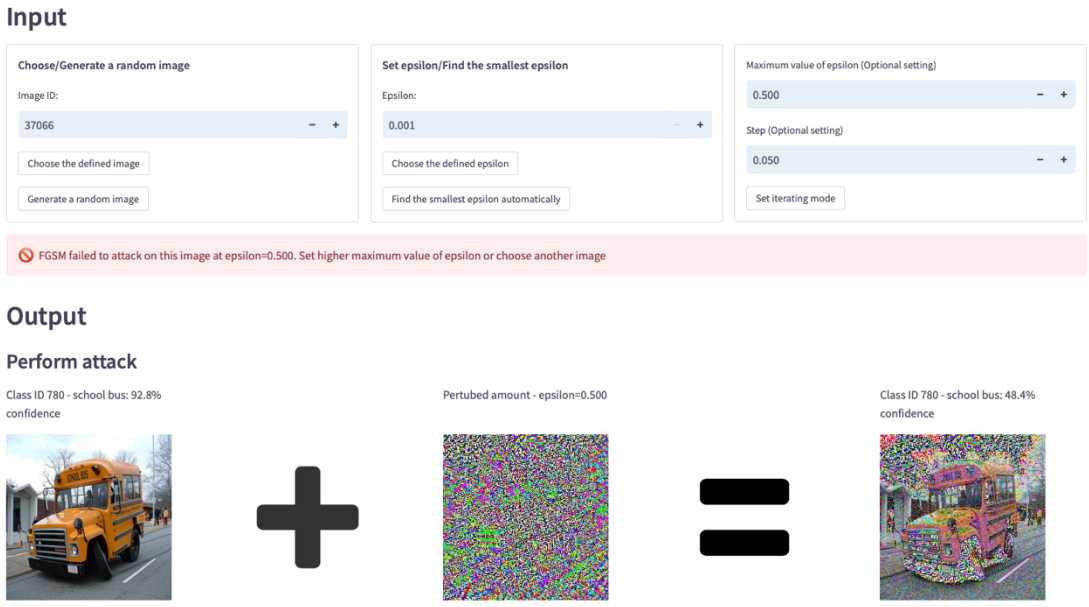

Figure 21: FGSM failed to attack on the image at the epsilon=0.5. The perturbed image at this epsilon is noticeable to human eyes

In general, to use the technique in the web application, users must follow these steps:

a) Choosing image: Users have the option to either choose a specific image by entering the Image ID and clicking on the “Choose the defined image” button or generate a random image by clicking on

the “Generate a random image” button. If the chosen or generated image is misclassified by ConvNeXt, users will need to select another image (see Figure 22).

b) Choosing *epsilon*: Users can select a specific *epsilon* value by entering it and clicking on the “Choose the defined epsilon” button. Alternatively, users can choose to let the algorithm find the smallest epsilon value automatically by clicking on the “Find the smallest epsilon automatically” button. The underlying algorithm will iterate through a set of epsilon values in ascending order until it reaches the *maximum value of epsilon*. After each iteration, the *epsilon* value will increase by an amount equal to the *step* variable. Users have the option to change the default values of these two variables as per their preference.

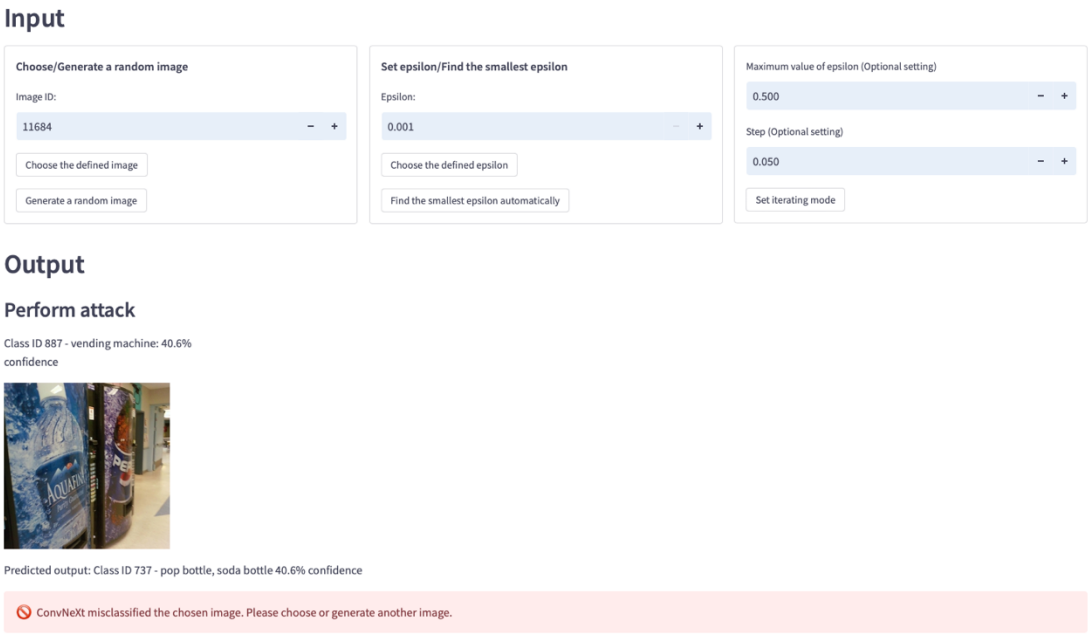

Figure 22: ConvNeXt misclassified the chosen image. Users need to choose another one.

For more additional demonstration, please refer to the following video at https://youtu.be/c2gi0rWGiFU.

4 Results and Discussion

The maximally activating image patches technique was used to explore the features learned by the ConvNeXt model. This technique is useful for understanding what aspects of the input images a particular layer is detecting. By finding the image patches that maximally activate each channel in a given layer, we can gain insights into the types of features that the layer is detecting (see Figure 23, Figure 24, Figure 25 and Figure 26).

Our results showed that as we moved up the layers of the ConvNeXt model, the patches that maximally activate the feature maps become more complex and resemble higher-level semantics, such as objects rather than simple parts. This is consistent with previous research, such as the study by Zhou et al. (2015) which demonstrated that object detectors emerge in deep scene CNNs.

Furthermore, our findings are in line with the work of Zeiler and Fergus (2014), who used similar visualization techniques to understand the features learned by a convolutional neural network. They found that the early layers of the network learn simple features such as edges and corners, while the later layers learn more complex features like object parts and even whole objects.

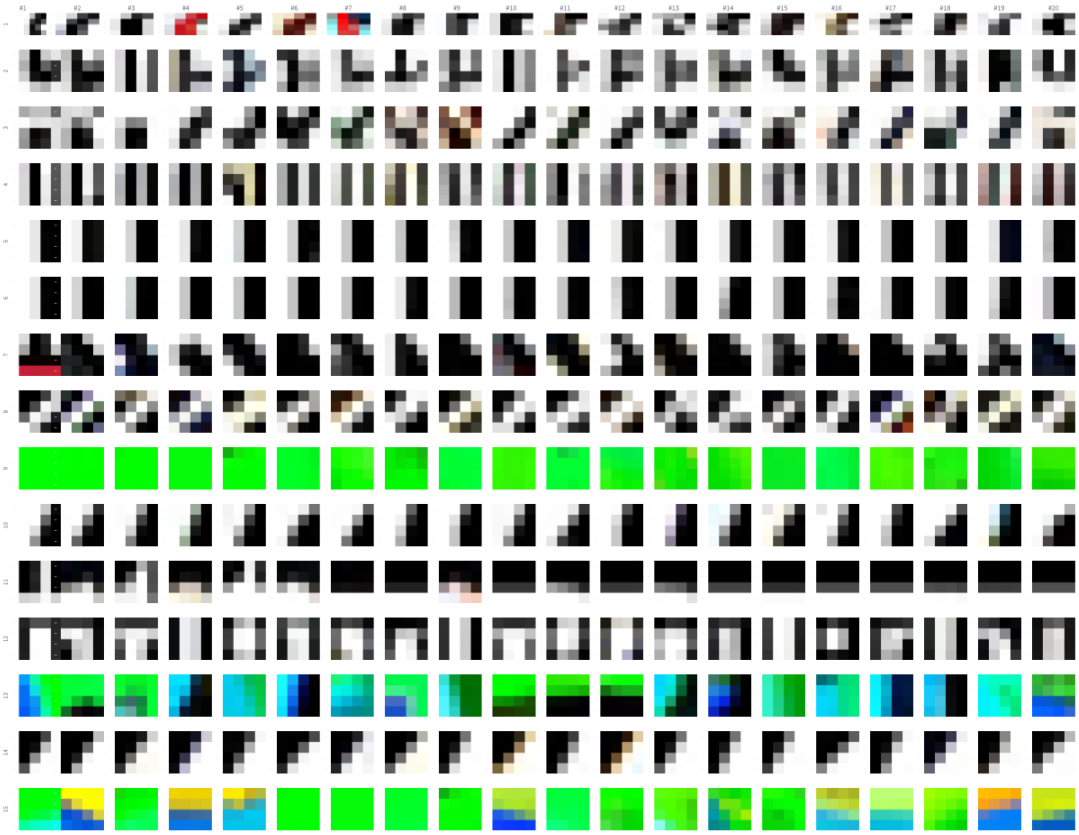

Figure 23: Top-20 maximally activating patches of channels 1-15 in Patchify layer

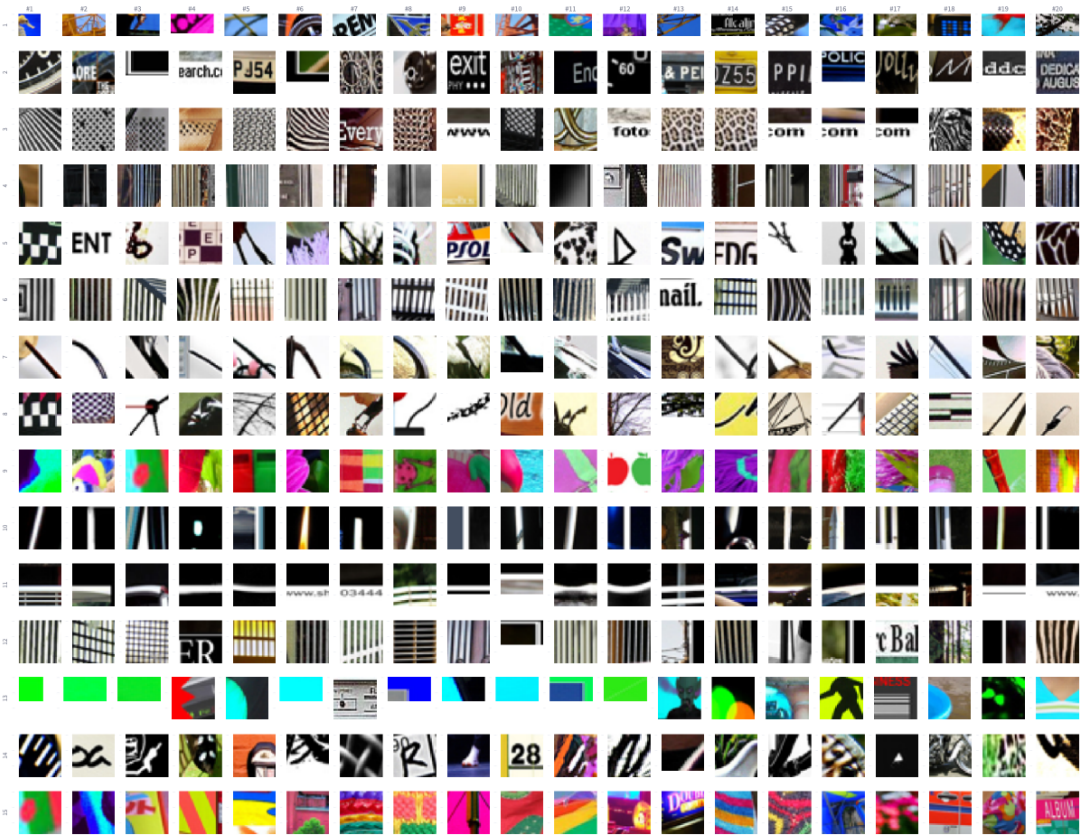

Figure 24: Top-20 maximally activating patches of channels 1-15 in dwconv layer in Block 1 of Stage 1

Figure 25: Top-10 maximally activating patches of channels 1-10 in dwconv layer in Block 1 of Stage 2

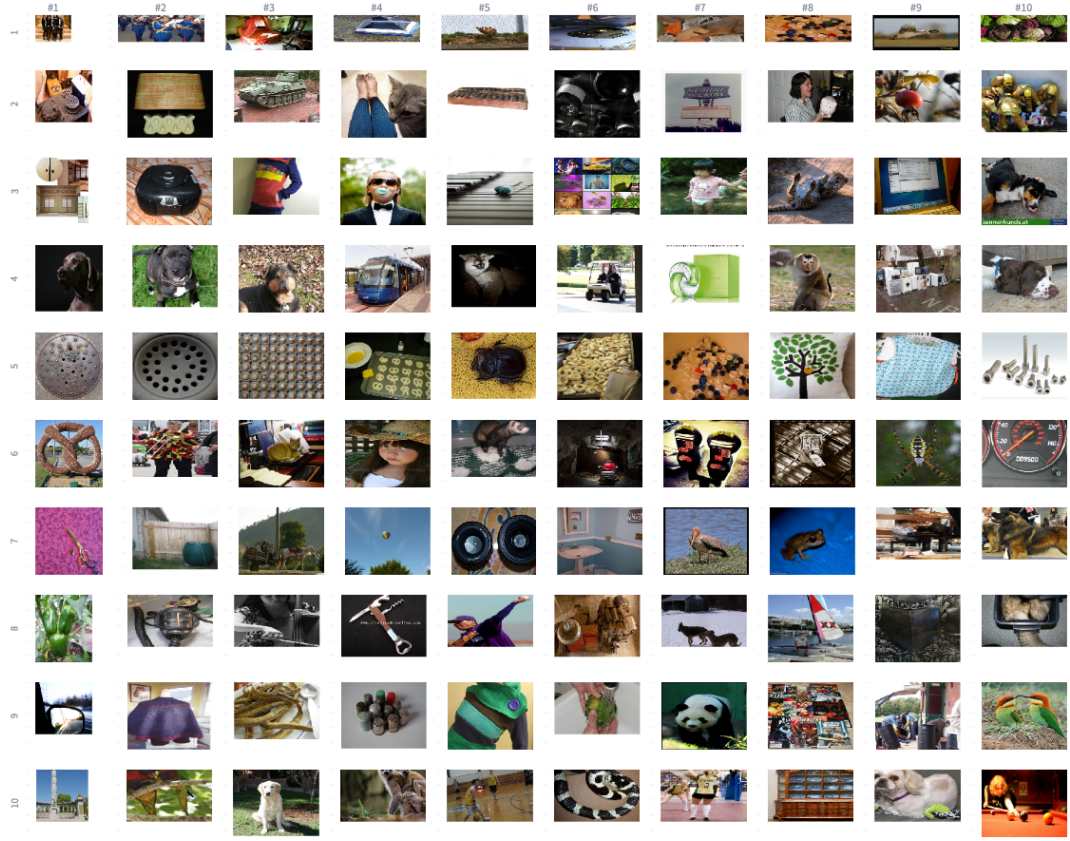

Figure 26: Top-10 maximally activating patches of channels 1-10 of dwconv layer in Block 1 of Stage 3

On the other hand, experiments of the feature attribution technique with SmoothGrad was implemented to compare the relative importance of pixels in ConvNeXt, ResNet, and MobileNet. The results showed that ConvNeXt focused on more accurate pixels when making predictions compared to ResNet and MobileNet (see Figure 16). This finding is indicative of the outperformance of ConvNeXt over the other two models. The higher accuracy in predicting which pixels are more important in the input image suggests that ConvNeXt has learned more robust and discriminative features for object recognition.

5 Conclusion and Future Works

In conclusion, this project aimed to facilitate researchers in interpreting ConvNeXt model by developing a web application that provides three visualization techniques: Maximally activating image patches, Feature attribution with SmoothGrad, and Adversarial perturbation with SmoothGrad. The Maximally activating image patches technique allows to interpret the learned features in ConvNeXt by highlighting the image patches that trigger the highest activation of the given channel. Meanwhile, Feature attribution with SmoothGrad reveals which pixels contribute the most to the model's prediction. Adversarial perturbation with SmoothGrad allows users to explore how adversarial attacks affect the model's interpretation. These techniques provide insights into how the ConvNeXt model processes and interprets images and can aid in model improvement and understanding.

For future work, several potential improvements and expansions to the project could be considered. One possibility is to integrate additional interpretability techniques, such as more saliency maps and data attribution techniques, to provide researchers with more insights into how ConvNeXt and other models are making their predictions.

Another potential area for expansion is to add support for additional models beyond ConvNeXt, allowing for a broader range of deep learning architectures to be analyzed. This would enable researchers to use the same web application to interpret a variety of models, potentially streamlining their workflow and increasing efficiency.

In addition, the project could be improved to allow users to upload their own image sets for analysis, rather than being limited to the pre-defined

set included with the application. This would increase the flexibility and usefulness of the tool for researchers working with their own data.

Finally, in order to improve the speed of processing and provide more powerful hardware resources, the project could be deployed on a server with GPU support. This would enable faster analysis and more advanced interpretability techniques, allowing researchers to gain even deeper insights into the workings of deep learning models.

References

Chen, Y., Li, B., Yu, H., Wu, P., & Miao, C. (2021). HyDRA: Hypergradient Data Relevance Analysis for Interpreting Deep Neural Networks. *Proceedings of the . . . AAAI Conference on Artificial Intelligence*, *35*(8), 7081–7089. https://doi.org/10.1609/aaai.v35i8.16871

Datasets at Hugging Face. (n.d.). Retrieved from https://huggingface.co/datasets/imagenet-1k

DOT Language. (2022, October 4). Retrieved from https://graphviz.org/doc/info/lang.html

Goodfellow, I., Shlens, J., & Szegedy, C. (2015). Explaining and Harnessing Adversarial Examples. International Conference on Learning Representations. Retrieved from https://ai.google/research/pubs/pub43405

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep Residual Learning for Image Recognition. Computer Vision and Pattern Recognition. https://doi.org/10.1109/cvpr.2016.90

Hien, D. H. (2018, June 20). A guide to receptive field arithmetic for Convolutional Neural Networks. Retrieved from https://blog.mlreview.com/a-guide-to-receptive-field-arithmetic-for-convolutional-neural-networks-e0f514068807

Howard, A. W., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., . . . Adam, H. (2017). MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv (Cornell University). Retrieved from http://export.arxiv.org/pdf/1704.04861

Hugging Face. (2022a, November 3). Retrieved from https://huggingface.co/facebook/convnext-tiny-224

Hugging Face. (2022b, November 3). Retrieved from https://huggingface.co/microsoft/resnet-50

Kaggle: Your Machine Learning and Data Science Community. (n.d.). Retrieved from https://www.kaggle.com/

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. Neural Information Processing Systems, 25, 1097–1105. Retrieved from http://books.nips.cc/papers/files/nips25/NIPS2012_0534.pdf

Liu, Z., Lin, Y., Cao, Y., Qiu, J., Wei, Y., Zhang, Z. G., . . . Guo, B. (2021). Swin Transformer: Hierarchical Vision Transformer using Shifted Windows. arXiv (Cornell University). https://doi.org/10.1109/iccv48922.2021.00986

Liu, Z., Mao, H., Wu, C., Feichtenhofer, C., Darrell, T., & Xie, S. (2022). A ConvNet for the 2020s. 2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). https://doi.org/10.1109/cvpr52688.2022.01167

PyTorch. (n.d.). Retrieved from https://pytorch.org/hub/pytorch_vision_mobilenet_v2/

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. International Conference on Computer Vision. https://doi.org/10.1109/iccv.2017.74

Selvaraju, R. R., Das, A., Vedantam, R., Cogswell, M., Parikh, D., & Batra, D. (2016). Grad-CAM: Why did you say that? arXiv (Cornell University). Retrieved from http://arxiv.org/pdf/1611.07450.pdf

Smilkov, D., Thorat, N., Kim, B., Viégas, F. B., & Wattenberg, M. (2017). SmoothGrad: removing noise by adding noise. arXiv (Cornell University). Retrieved from https://arxiv.org/pdf/1706.03825.pdf

Spaces - Hugging Face. (n.d.). Retrieved from https://huggingface.co/spaces

Springenberg, J. T., Dosovitskiy, A., Brox, T., & Riedmiller, M. (2015). Striving for Simplicity: The All Convolutional Net. International Conference on Learning Representations. Retrieved from https://arxiv.org/pdf/1412.6806

Streamlit - The fastest way to build and share data apps. (n.d.). Retrieved from https://streamlit.io/

t-SNE visualization of CNN codes. (n.d.). Retrieved from https://cs.stanford.edu/people/karpathy/cnnembed/

Zeiler, M. D., & Fergus, R. (2014). Visualizing and Understanding Convolutional Networks. Lecture Notes in Computer Science, 818–833. https://doi.org/10.1007/978-3-319-10590-1_53

Zhou, B., Khosla, A., Lapedriza, A., Oliva, A., & Torralba, A. (2015). Object Detectors Emerge in Deep Scene CNNs. International Conference on Learning Representations. Retrieved from http://places.csail.mit.edu/slide_iclr2015.pdf

Appendix A

```python
def get_activation(name):
    """
    activation[layer_name] = tensor of size (top_k, C, 4) where
        dim 0: top_k
        dim 1: channel
        dim 2: [activation value, x, y, index of original image]
    """
    dim_channel = 1
    if '.layernorm' in name or '.pwconv1' in name or '.pwconv2' in name:
        dim_channel = 3
    global start_index

    def hook(model, input, output):
        global coor_max_
        output = output.detach().cpu()

        coor_max = get_coor_max(output, start_index, dim_channel=dim_channel)

        if name in activation:
            activation[name] = np.concatenate((activation[name], coor_max),axis=0)
        else:
            activation[name] = coor_max

        ind, sorted_activation = top_k_activation(activation[name], top_k=TOP_K)
        activation[name] = sorted_activation
    return hook
def register_forward_hook_model(model):
    # embeddings
    model.embeddings.patch_embeddings.register_forward_hook(get_activation('embeddings.patch_embeddings'))

    # encoder stages
    no_blocks_each_stage = [3, 3, 9, 3]
    for stage, no_blocks in enumerate(no_blocks_each_stage):
        # downsampling layer
        if stage == 0:
            pass
        else:
            model.encoder.stages[stage].downsampling_layer[1].register_forward_hook(
                get_activation(f'encoder.stages[{stage}].downsampling_layer[1]'))
        # layers (aka blocks)
        for block in range(no_blocks):
            model.encoder.stages[stage].layers[block].dwconv.register_forward_hook(
                get_activation(f'encoder.stages[{stage}].layers[{block}].dwconv'))
            model.encoder.stages[stage].layers[block].pwconv1.register_forward_hook(
                get_activation(f'encoder.stages[{stage}].layers[{block}].pwconv1'))
            model.encoder.stages[stage].layers[block].pwconv2.register_forward_hook(
                get_activation(f'encoder.stages[{stage}].layers[{block}].pwconv2'))

activation = OrderedDict()
register_forward_hook_model(model)
```

Figure 27: register_forward_hook() to save activation values and sort top-k avtivations

```
digraph convnext {
    graph [labeljust=left ranksep=0.2]
    node [fixedsize=true shape=box width=3]
    edge [arrowhead=vee]
    image [label=image shape=plaintext]
    embeddings_conv [label="4x4 conv, 96, stride 4"]
    image -> embeddings_conv
```

```
    subgraph "cluster stage 1" {
        graph [color="#e9f2f7" label="stage 1\l" style=filled]
        node [color="#bcd9e7" style=filled]
        subgraph "cluster stage 1 block 1" {
            graph [color="#0f4158" label="block 1\l" style=dashed]
            "stage 1 block 1 dwconv" [label="d7x7 conv, 96, stride 1"]
            "stage 1 block 1 pwconv1" [label="1x1 conv, 384"]
            "stage 1 block 1 pwconv2" [label="1x1 conv, 96"]
            "stage 1 block 1 dwconv" -> "stage 1 block 1 pwconv1"
            "stage 1 block 1 pwconv1" -> "stage 1 block 1 pwconv2"
        }
        subgraph "cluster stage 1 block 2" {
            graph [color="#0f4158" label="block 2\l" style=dashed]
            "stage 1 block 2 dwconv" [label="d7x7 conv, 96, stride 1"]
            "stage 1 block 2 pwconv1" [label="1x1 conv, 384"]
            "stage 1 block 2 pwconv2" [label="1x1 conv, 96"]
            "stage 1 block 2 dwconv" -> "stage 1 block 2 pwconv1"
            "stage 1 block 2 pwconv1" -> "stage 1 block 2 pwconv2"
        }
        subgraph "cluster stage 1 block 3" {
            graph [color="#0f4158" label="block 3\l" style=dashed]
            "stage 1 block 3 dwconv" [label="d7x7 conv, 96, stride 1"]
            "stage 1 block 3 pwconv1" [label="1x1 conv, 384"]
            "stage 1 block 3 pwconv2" [label="1x1 conv, 96"]
            "stage 1 block 3 dwconv" -> "stage 1 block 3 pwconv1"
            "stage 1 block 3 pwconv1" -> "stage 1 block 3 pwconv2"
        }
        "stage 1 block 1 pwconv2" -> "stage 1 block 2 dwconv"
        "stage 1 block 2 pwconv2" -> "stage 1 block 3 dwconv"
    }
    subgraph "cluster stage 2" {
        graph [color="#e9f7f3" label="stage 2\l" style=filled]
        node [color="#bce7db" style=filled]
        "stage 2 downsampling" [label="2x2 conv, 192, stride 2"]
        subgraph "cluster stage 2 block 1" {
            graph [color="#0f5844" label="block 1\l" style=dashed]
            "stage 2 block 1 dwconv" [label="d7x7 conv, 192, stride 1"]
            "stage 2 block 1 pwconv1" [label="1x1 conv, 768"]
            "stage 2 block 1 pwconv2" [label="1x1 conv, 192"]
            "stage 2 block 1 dwconv" -> "stage 2 block 1 pwconv1"
            "stage 2 block 1 pwconv1" -> "stage 2 block 1 pwconv2"
        }
        subgraph "cluster stage 2 block 2" {
            graph [color="#0f5844" label="block 2\l" style=dashed]
            "stage 2 block 2 dwconv" [label="d7x7 conv, 192, stride 1"]
            "stage 2 block 2 pwconv1" [label="1x1 conv, 768"]
            "stage 2 block 2 pwconv2" [label="1x1 conv, 192"]
            "stage 2 block 2 dwconv" -> "stage 2 block 2 pwconv1"
            "stage 2 block 2 pwconv1" -> "stage 2 block 2 pwconv2"
```

```
        }
        subgraph "cluster stage 2 block 3" {
            graph [color="#0f5844" label="block 3\l" style=dashed]
            "stage 2 block 3 dwconv" [label="d7x7 conv, 192, stride 1"]
            "stage 2 block 3 pwconv1" [label="1x1 conv, 768"]
            "stage 2 block 3 pwconv2" [label="1x1 conv, 192"]
            "stage 2 block 3 dwconv" -> "stage 2 block 3 pwconv1"
            "stage 2 block 3 pwconv1" -> "stage 2 block 3 pwconv2"
        }
        "stage 2 downsampling" -> "stage 2 block 1 dwconv"
        "stage 2 block 1 pwconv2" -> "stage 2 block 2 dwconv"
        "stage 2 block 2 pwconv2" -> "stage 2 block 3 dwconv"
    }
    subgraph "cluster stage 3" {
        graph [color="#e9f2f7" label="stage 3\l" style=filled]
        node [color="#bcd9e7" style=filled]
        "stage 3 downsampling" [label="2x2 conv, 384, stride 2"]
        subgraph "cluster stage 3 block 1" {
            graph [color="#0f4158" label="block 1\l" style=dashed]
            "stage 3 block 1 dwconv" [label="d7x7 conv, 384, stride 1"]
            "stage 3 block 1 pwconv1" [label="1x1 conv, 1536"]
            "stage 3 block 1 pwconv2" [label="1x1 conv, 384"]
            "stage 3 block 1 dwconv" -> "stage 3 block 1 pwconv1"
            "stage 3 block 1 pwconv1" -> "stage 3 block 1 pwconv2"
        }
        subgraph "cluster stage 3 block 2" {
            graph [color="#0f4158" label="block 2\l" style=dashed]
            "stage 3 block 2 dwconv" [label="d7x7 conv, 384, stride 1"]
            "stage 3 block 2 pwconv1" [label="1x1 conv, 1536"]
            "stage 3 block 2 pwconv2" [label="1x1 conv, 384"]
            "stage 3 block 2 dwconv" -> "stage 3 block 2 pwconv1"
            "stage 3 block 2 pwconv1" -> "stage 3 block 2 pwconv2"
        }
        subgraph "cluster stage 3 block 3" {
            graph [color="#0f4158" label="block 3\l" style=dashed]
            "stage 3 block 3 dwconv" [label="d7x7 conv, 384, stride 1"]
            "stage 3 block 3 pwconv1" [label="1x1 conv, 1536"]
            "stage 3 block 3 pwconv2" [label="1x1 conv, 384"]
            "stage 3 block 3 dwconv" -> "stage 3 block 3 pwconv1"
            "stage 3 block 3 pwconv1" -> "stage 3 block 3 pwconv2"
        }
        subgraph "cluster stage 3 block 4" {
            graph [color="#0f4158" label="block 4\l" style=dashed]
            "stage 3 block 4 dwconv" [label="d7x7 conv, 384, stride 1"]
            "stage 3 block 4 pwconv1" [label="1x1 conv, 1536"]
            "stage 3 block 4 pwconv2" [label="1x1 conv, 384"]
            "stage 3 block 4 dwconv" -> "stage 3 block 4 pwconv1"
            "stage 3 block 4 pwconv1" -> "stage 3 block 4 pwconv2"
        }
```

```
        subgraph "cluster stage 3 block 5" {
            graph [color="#0f4158" label="block 5\l" style=dashed]
            "stage 3 block 5 dwconv" [label="d7x7 conv, 384, stride 1"]
            "stage 3 block 5 pwconv1" [label="1x1 conv, 1536"]
            "stage 3 block 5 pwconv2" [label="1x1 conv, 384"]
            "stage 3 block 5 dwconv" -> "stage 3 block 5 pwconv1"
            "stage 3 block 5 pwconv1" -> "stage 3 block 5 pwconv2"
        }
        subgraph "cluster stage 3 block 6" {
            graph [color="#0f4158" label="block 6\l" style=dashed]
            "stage 3 block 6 dwconv" [label="d7x7 conv, 384, stride 1"]
            "stage 3 block 6 pwconv1" [label="1x1 conv, 1536"]
            "stage 3 block 6 pwconv2" [label="1x1 conv, 384"]
            "stage 3 block 6 dwconv" -> "stage 3 block 6 pwconv1"
            "stage 3 block 6 pwconv1" -> "stage 3 block 6 pwconv2"
        }
        subgraph "cluster stage 3 block 7" {
            graph [color="#0f4158" label="block 7\l" style=dashed]
            "stage 3 block 7 dwconv" [label="d7x7 conv, 384, stride 1"]
            "stage 3 block 7 pwconv1" [label="1x1 conv, 1536"]
            "stage 3 block 7 pwconv2" [label="1x1 conv, 384"]
            "stage 3 block 7 dwconv" -> "stage 3 block 7 pwconv1"
            "stage 3 block 7 pwconv1" -> "stage 3 block 7 pwconv2"
        }
        subgraph "cluster stage 3 block 8" {
            graph [color="#0f4158" label="block 8\l" style=dashed]
            "stage 3 block 8 dwconv" [label="d7x7 conv, 384, stride 1"]
            "stage 3 block 8 pwconv1" [label="1x1 conv, 1536"]
            "stage 3 block 8 pwconv2" [label="1x1 conv, 384"]
            "stage 3 block 8 dwconv" -> "stage 3 block 8 pwconv1"
            "stage 3 block 8 pwconv1" -> "stage 3 block 8 pwconv2"
        }
        subgraph "cluster stage 3 block 9" {
            graph [color="#0f4158" label="block 9\l" style=dashed]
            "stage 3 block 9 dwconv" [label="d7x7 conv, 384, stride 1"]
            "stage 3 block 9 pwconv1" [label="1x1 conv, 1536"]
            "stage 3 block 9 pwconv2" [label="1x1 conv, 384"]
            "stage 3 block 9 dwconv" -> "stage 3 block 9 pwconv1"
            "stage 3 block 9 pwconv1" -> "stage 3 block 9 pwconv2"
        }
        "stage 3 downsampling" -> "stage 3 block 1 dwconv"
        "stage 3 block 1 pwconv2" -> "stage 3 block 2 dwconv"
        "stage 3 block 2 pwconv2" -> "stage 3 block 3 dwconv"
        "stage 3 block 3 pwconv2" -> "stage 3 block 4 dwconv"
        "stage 3 block 4 pwconv2" -> "stage 3 block 5 dwconv"
        "stage 3 block 5 pwconv2" -> "stage 3 block 6 dwconv"
        "stage 3 block 6 pwconv2" -> "stage 3 block 7 dwconv"
        "stage 3 block 7 pwconv2" -> "stage 3 block 8 dwconv"
        "stage 3 block 8 pwconv2" -> "stage 3 block 9 dwconv"
```

```
    }
    subgraph "cluster stage 4" {
        graph [color="#e9f7f3" label="stage 4\l" style=filled]
        node [color="#bce7db" style=filled]
        "stage 4 downsampling" [label="2x2 conv, 768, stride 2"]
        subgraph "cluster stage 4 block 1" {
            graph [color="#0f5844" label="block 1\l" style=dashed]
            "stage 4 block 1 dwconv" [label="d7x7 conv, 768, stride 1"]
            "stage 4 block 1 pwconv1" [label="1x1 conv, 3072"]
            "stage 4 block 1 pwconv2" [label="1x1 conv, 768"]
            "stage 4 block 1 dwconv" -> "stage 4 block 1 pwconv1"
            "stage 4 block 1 pwconv1" -> "stage 4 block 1 pwconv2"
        }
        subgraph "cluster stage 4 block 2" {
            graph [color="#0f5844" label="block 2\l" style=dashed]
            "stage 4 block 2 dwconv" [label="d7x7 conv, 768, stride 1"]
            "stage 4 block 2 pwconv1" [label="1x1 conv, 3072"]
            "stage 4 block 2 pwconv2" [label="1x1 conv, 768"]
            "stage 4 block 2 dwconv" -> "stage 4 block 2 pwconv1"
            "stage 4 block 2 pwconv1" -> "stage 4 block 2 pwconv2"
        }
        subgraph "cluster stage 4 block 3" {
            graph [color="#0f5844" label="block 3\l" style=dashed]
            "stage 4 block 3 dwconv" [label="d7x7 conv, 768, stride 1"]
            "stage 4 block 3 pwconv1" [label="1x1 conv, 3072"]
            "stage 4 block 3 pwconv2" [label="1x1 conv, 768"]
            "stage 4 block 3 dwconv" -> "stage 4 block 3 pwconv1"
            "stage 4 block 3 pwconv1" -> "stage 4 block 3 pwconv2"
        }
        "stage 4 downsampling" -> "stage 4 block 1 dwconv"
        "stage 4 block 1 pwconv2" -> "stage 4 block 2 dwconv"
        "stage 4 block 2 pwconv2" -> "stage 4 block 3 dwconv"
    }
    "stage 1 block 3 pwconv2" -> "stage 2 downsampling"
    "stage 2 block 3 pwconv2" -> "stage 3 downsampling"
    "stage 3 block 9 pwconv2" -> "stage 4 downsampling"
    embeddings_conv -> "stage 1 block 1 dwconv"
    "stage 4 block 3 pwconv2" -> "output vector"
    "output vector" [label="output vector" shape=plaintext]
}
```

Figure 28: Code snippets in DOT language for ConvNeXt architecture

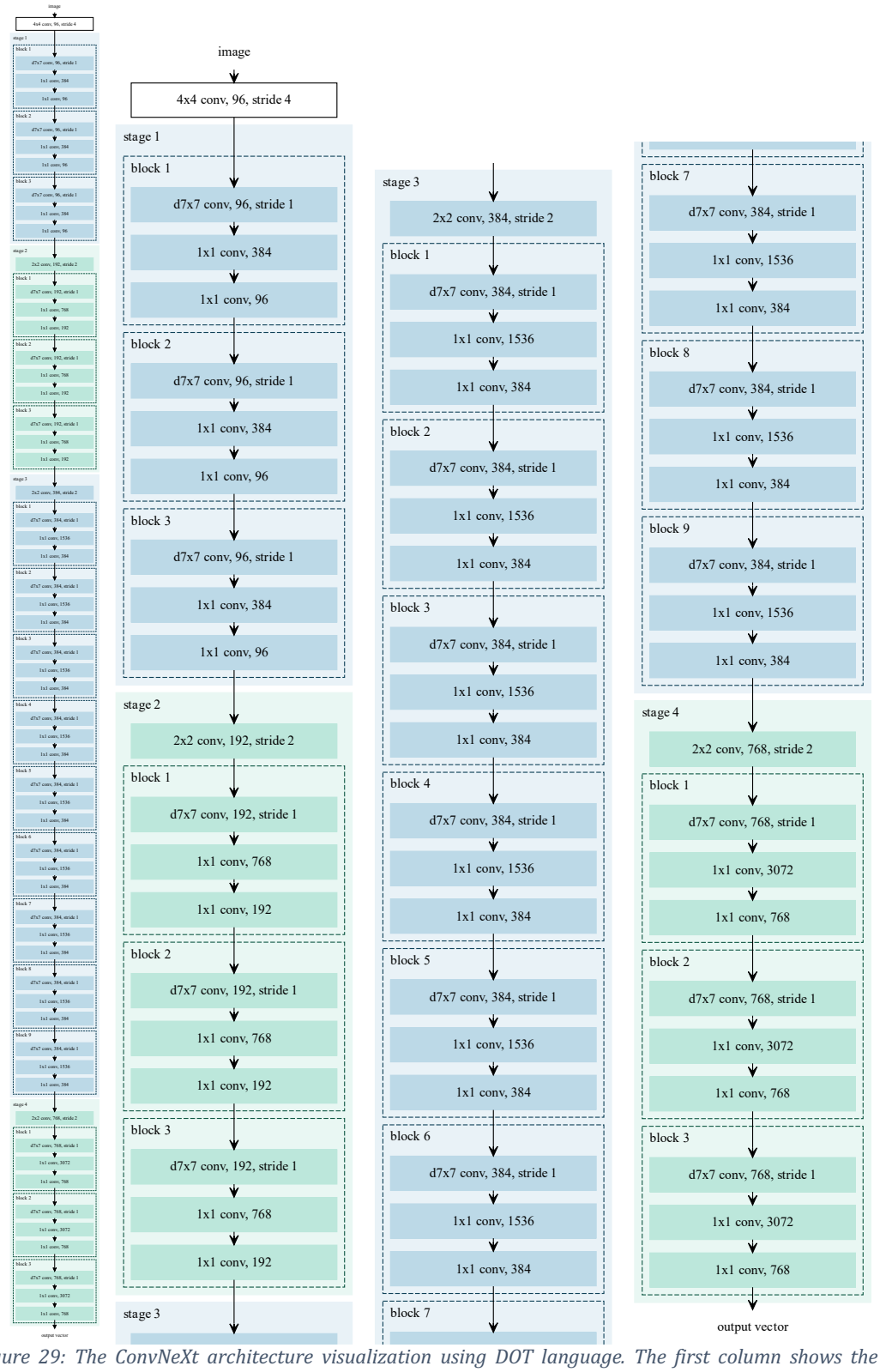

Figure 29: The ConvNeXt architecture visualization using DOT language. The first column shows the full architecture. The remaining columns zoom out the parts in the first column.

Appendix B

```
class SimpleGradient(ExplanationMethod):
    def __init__(self, model, create_graph=False,
                 preprocess=None, postprocess=None):
        super().__init__(model, preprocess, postprocess)
        self.create_graph = create_graph

    def predict(self, x):
        return self.model(x)

    @torch.enable_grad()
    def process(self, inputs, target):
        self.model.zero_grad()
        inputs.requires_grad_(True)

        out = self.model(inputs)
        out = out if type(out) == torch.Tensor else out.logits

        num_classes = out.size(-1)
        onehot = torch.zeros(inputs.size(0), num_classes,
*target.shape[1:])
        onehot = onehot.to(dtype=inputs.dtype, device=inputs.device)
        onehot.scatter_(1, target.unsqueeze(1), 1)

        grad, = torch.autograd.grad(
            (out*onehot).sum(), inputs, create_graph=self.create_graph
        )
        return grad


class SmoothGradient(ExplanationMethod):
    def __init__(self, model, stdev_spread=0.15, num_samples=25,
                 magnitude=True, batch_size=-1,
                 create_graph=False, preprocess=None, postprocess=None):
        super().__init__(model, preprocess, postprocess)
        self.stdev_spread = stdev_spread
        self.nsample = num_samples
        self.create_graph = create_graph
        self.magnitude = magnitude
        self.batch_size = batch_size
        if self.batch_size == -1:
            self.batch_size = self.nsample

        self._simgrad = SimpleGradient(model, create_graph)

    def process(self, inputs, target):
```

```
        self.model.zero_grad()

        maxima = inputs.flatten(1).max(-1)[0]
        minima = inputs.flatten(1).min(-1)[0]

        stdev = self.stdev_spread * (maxima - minima).cpu()
        stdev = stdev.view(inputs.size(0), 1, 1, 1).expand_as(inputs)
        stdev = stdev.unsqueeze(0).expand(self.nsample, *[-1]*4)
        noise = torch.normal(0, stdev)

        target_expanded = target.unsqueeze(0).cpu()
        target_expanded = target_expanded.expand(noise.size(0), -1)

        noiseloader = torch.utils.data.DataLoader(
            TensorDataset(noise, target_expanded),
batch_size=self.batch_size
        )

        total_gradients = torch.zeros_like(inputs)
        for noise, t_exp in noiseloader:
            inputs_w_noise = inputs.unsqueeze(0) + noise.to(inputs.device)
            inputs_w_noise = inputs_w_noise.view(-1, *inputs.shape[1:])
            gradients = self._simgrad(inputs_w_noise, t_exp.view(-1))
            gradients = gradients.view(self.batch_size, *inputs.shape)
            if self.magnitude:
                gradients = gradients.pow(2)
            total_gradients = total_gradients + gradients.sum(0)

        smoothed_gradient = total_gradients / self.nsample
        return smoothed_gradient


def feed_forward(model_name, image, model=None, feature_extractor=None):
    if model_name in ['ConvNeXt', 'ResNet']:
        inputs = feature_extractor(image, return_tensors="pt")
        logits = model(**inputs).logits
        prediction_class = logits.argmax(-1).item()
    else:
        transform_images = transforms.Compose([
        transforms.Resize(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
0.225])])
        input_tensor = transform_images(image)
        inputs = input_tensor.unsqueeze(0)

        output = model(inputs)
        prediction_class = output.argmax(-1).item()
    return inputs, prediction_class
```

```
def clip_gradient(gradient):
    gradient = gradient.abs().sum(1, keepdim=True)
    return clamp_quantile(gradient, q=0.99)

def fig2img(fig):
    """Convert a Matplotlib figure to a PIL Image and return it"""
    import io
    buf = io.BytesIO()
    fig.savefig(buf)
    buf.seek(0)
    img = Image.open(buf)
    return img

def generate_smoothgrad_mask(image, model_name, model=None,
feature_extractor=None, num_samples=25, return_mask=False):
    inputs, prediction_class = feed_forward(model_name, image, model,
feature_extractor)

    smoothgrad_gen = SmoothGradient(
        model, num_samples=num_samples, stdev_spread=0.1,
        magnitude=False, postprocess=clip_gradient)

    if type(inputs) != torch.Tensor:
        inputs = inputs['pixel_values']

    smoothgrad_mask = smoothgrad_gen(inputs, prediction_class)
    smoothgrad_mask = smoothgrad_mask[0].numpy()
    smoothgrad_mask = np.transpose(smoothgrad_mask, (1, 2, 0))

    image = np.asarray(image)
    heat_map_image = show_heatmap(smoothgrad_mask)
    masked_image = show_masked_image(smoothgrad_mask, image)

    if return_mask:
        return heat_map_image, masked_image, smoothgrad_mask
    else:
        return heat_map_image, masked_image
```

Figure 30: Code snippet for modifications in implementation of SmoothGrad

```
import PIL
from PIL import Image
import numpy as np
from matplotlib import pylab as P
import cv2

def show_masked_image(saliency_map, image):
    """
    Save saliency map on image.
```

```
    Args:
        image: Tensor of size (H,W,3)
        saliency_map: Tensor of size (H,W,1)
    """

    saliency_map = saliency_map - saliency_map.min()
    saliency_map = saliency_map / saliency_map.max()
    saliency_map = saliency_map.clip(0,1)
    saliency_map = np.uint8(saliency_map * 255)

    saliency_map = cv2.resize(saliency_map, (224,224))
    image = cv2.resize(image, (224, 224))

    # Apply JET colormap
    color_heatmap = cv2.applyColorMap(saliency_map, cv2.COLORMAP_HOT)

    # Blend image with heatmap
    img_with_heatmap = cv2.addWeighted(image, 0.4, color_heatmap, 0.6, 0)

    return img_with_heatmap
```

Figure 31: Code snippet for overlaying the SmoothGrad maps onto the original image

Appendix C

```
# FGSM attack code
def fgsm_attack(image, epsilon, data_grad):
    # Collect the element-wise sign of the data gradient and normalize it
    sign_data_grad = torch.gt(data_grad, 0).type(torch.FloatTensor) * 2.0 -
1.0
    perturbed_image = image + epsilon*sign_data_grad
    return perturbed_image

# perform attack on the model
def perform_attack(input_image, target, epsilon):
    model, feature_extractor = load_model("ConvNeXt")
    # preprocess input image
    inputs = feature_extractor(input_image, do_resize=False,
return_tensors="pt")['pixel_values']
    inputs.requires_grad = True

    # predict
    logits = model(inputs).logits
    prediction_prob = F.softmax(logits, dim=-1).max()
    prediction_class = logits.argmax(-1).item()
    prediction_label = model.config.id2label[prediction_class]

    # Calculate the loss
    loss = F.nll_loss(logits, torch.tensor([target]))

    # Zero all existing gradients
    model.zero_grad()

    # Calculate gradients of model in backward pass
    loss.backward()

    # Collect datagrad
    data_grad = inputs.grad.data

    # Call FGSM Attack
    perturbed_data = fgsm_attack(inputs, epsilon, data_grad)

    # Re-classify the perturbed image
    new_prediction = model(perturbed_data).logits
    new_pred_prob = F.softmax(new_prediction, dim=-1).max()
    new_pred_class = new_prediction.argmax(-1).item()
    new_pred_label = model.config.id2label[new_pred_class]

    return perturbed_data, new_pred_prob.item(), new_pred_class,
new_pred_label
```

```
def find_smallest_epsilon(input_image, target):
    epsilons = [i*0.001 for i in range(1000)]

    for epsilon in epsilons:
        perturbed_data, new_prob, new_id, new_label =
perform_attack(input_image, target, epsilon)
        if new_id != target:
            return perturbed_data, new_prob, new_id, new_label, epsilon
    return None
```

Figure 32: Code snippet for FGSM attack and algorithm to find the smallest epsilon if necessary