



**QUEEN'S
UNIVERSITY
BELFAST**

Multi-task learning of objects and parts

A dissertation submitted in partial fulfilment of

the requirements for the degree of

BACHELOR OF SCIENCE/ENGINEERING* in Computer Science

in

The Queen's University of Belfast

by

Hailin Weng

15/04/2024

SCHOOL OF ELECTRONICS, ELECTRICAL ENGINEERING and COMPUTER SCIENCE

CSC3002 – COMPUTER SCIENCE PROJECT

Dissertation Cover Sheet

A signed and completed cover sheet must accompany the submission of the Software Engineering dissertation submitted for assessment.

Work submitted without a cover sheet will **NOT** be marked.

Student Name:	Hailin Weng	Student Number:	40381868
Project Title:	P25 Multi-task learning of objects and parts		
Supervisor:	Dr Barry Devereux		

Declaration of Academic Integrity

Before submitting your dissertation please check that the submission:

1. Has a full bibliography attached laid out according to the guidelines specified in the Student Project Handbook
2. Contains full acknowledgement of all secondary sources used (paper-based and electronic)
3. Does not exceed the specified page limit
4. Is clearly presented and proof-read
5. Is submitted on, or before, the specified or agreed due date. Late submissions will only be accepted in exceptional circumstances or where a deferment has been granted in advance.

By submitting your dissertation you declare that you have completed the tutorial on plagiarism at <http://www.qub.ac.uk/cite2write/introduction5.html> and are aware that it is an academic offence to plagiarise. You declare that the submission is your own original work. No part of it has been submitted for any other assignment and you have acknowledged all written and electronic sources used.

6. If selected as an exemplar, I agree to allow my dissertation to be used as a sample for future students. (Please delete this if you do not agree.)

Student's signature

Hailin Weng

Date of submission

15/04/2024

Contents

Acknowledgements	4
Abstract	5
1.0 Introduction and Problem Area	5
2.0 System Requirements and Specification	6
2.1 Solution Description	6
2.2 Property Norms Infilling	7
2.3 Mapping between CSLB and THINGS	8
2.4 Concept Classification	9
2.5 Feature Classification	10
2.6 System Requirements	11
3.0 Design	11
3.1 AlexNet Architecture	11
3.2 Unified Model Architecture	12
3.3 Concept Model Architecture	13
3.4 Feature Model Architecture	13
3.5 Convolutional Layer	14
3.6 Fully Connected Layer	15
3.7 Loss Function	15
3.8 Metrics	17
3.9 Error, Event and Exception Handling	18
4.0 Implementation	19
4.1 Implementation Descriptions: Languages and Environments	19
4.2 Software Libraries	19
4.3 Important Functions / Algorithms	21
5.0 Testing	28
5.1 AlexNet Architecture	28
5.2 Functionality Testing: Custom Datasets	31
5.3 Predict	33
5.4 RSM & Second Order RSM	35
6.0 System Evaluation and Experimental Results	38

6.1 CNN's Performance in Object Concepts Classification	38
6.2 CNN's Performance in Object Features Classification	39
6.3 RSA: Difference in Models.....	42
References	44

Acknowledgements

At the end of my undergraduate career, I want to express my deepest gratitude to my supervisor, Dr. Barry Devereux. His guidance and insightful feedback helped me a lot, and his support has been the foundation of my growth and understanding in the field of Computer Vision. Under his supervision, I learned to delve deeper into this project and to fall in love with Computer Vision. It's an honour for me to be his student.

I am deeply grateful to Professors Fei-Fei Li, Andrew Ng and Hung-Yi Lee for their inspiring courses. It's not an exaggeration to say that I'm a huge beneficiary of their courses. I don't think I can acquire such knowledge from AI without them.

I sincerely thank Kaixi. She is so wise and kindness, and it was her help that helped me to regain my confidence and get through this difficult time.

To my family and friends, I want to express my grateful for your warm emotional support and encouragement. Your belief in me has been the driving force behind my perseverance, especially during the rigours of my final year. In times of doubt, your love and constant motivation pushed me forward. I am really thankful for all that you have done for me.

I am also grateful to the pioneering researchers. Their scholarly works have paved the path, so that I have had the privilege to follow. As an undergraduate, it has been an honour and a formidable challenge for me to build upon their foundational research and contribute to the field of Computer Vision.

With a deep sense of self-awareness, I am grateful for the hard work and commitment that have driven me throughout this journey. Starting as a beginner with little understanding of the project at hand, I have tirelessly pursued knowledge and skills, finally culminating in the completion of my project and dissertation. This milestone is a testament to the challenging yet rewarding pursuit of knowledge and personal growth that I have undertaken.

This experience has been truly transformative, not only teaching me about the field of study, but also about resilience, curiosity, and the relentless pursuit of excellence. I will carry these qualities with me as I move forward into the next stage of my life, and eternally grateful for every lesson I learned, every challenge I faced, and every support I got on this remarkable academic expedition.

Abstract

With the development of hardware, computer vision has become increasingly popular in the field of machine learning. In this paper, I present a comparative analysis of three models which were trained independently. The first model is called the "**Unified model**", which can perform both conceptual classification and feature classification of images. The "**Concept model**", on the other hand, is only for concept classification, while the "**Feature model**" only classifies semantic features. Each concept is exclusive. As an example, an image of a dog will be classified only as a dog. In contrast, features are non-exclusive, and a single image may possess multiple real features. For instance, a cat may have features such as "*has_a_tail*," "*is_biotic*," "*does_eat*," and "*is_cute*."

1.0 Introduction and Problem Area

In the field of computational vision, researchers aim to develop machines with vision capabilities similar to humans. Advances in this field have been made possible through the use of deep learning methodologies, particularly Convolutional Neural Networks (CNNs). These networks have achieved remarkable success in image classification tasks by identifying objects through predefined labels, like AlexNet, VGG, and ResNet. However, these models have a major limitation in that they lack the ability to understand the intrinsic attributes or properties of objects, also known as property norms. Understanding these properties is crucial in a wide range of applications such as advanced image search engines, augmented reality, and robotics.

This challenge arises when machines need to identify shared properties across objects, such as recognizing that both "**EAGLES**" and "**WOLVES**" have "**EYES**." Existing models are unable to identify common properties across various object categories due to their concept-centric training. They lack the granularity required to achieve this task.

This limitation hinders the application of existing computational vision models in scenarios where understanding shared properties of objects could significantly enhance machine perception. Recent research suggests a more nuanced approach, which involves training models on property norms alongside traditional object labels, could offer a more comprehensive understanding of image content. Such an approach would facilitate a richer interpretation of visual data.

The project aims to develop a Convolutional Neural Network (CNN) model which is trained not only on object concept labels but also on semantic feature labels derived from the CSLB property norms. The goal is to address the gap identified in existing computational vision models. In simple terms, an object can only be recognized as a unique concept, such as a cat, ice cream, bass, or car. However, an object can have many semantic features simultaneously, for example, a cat has features like "*has_a_tail*," "*is_biotic*," "*is_cute*," etc. There may also be potential correlations between these features. For instance, all creatures must eat, and all vehicles are made of metal.

The following table shows some semantic features of the two concepts mentioned above, cat and car, as examples:

Concepts	Features				
Cat	does_run	does_move	has_eyes	is_biotic	is_cute
Car	does_carry_transport	does_move	has_four_wheels	is_a_vehicle	is_useful

The feature data used in this research was taken from the Centre for Speech, Language and the Brain (CSLB) Concept Property Norms (Devereux et al., 2013). The data collected was based on the features given in the property norms and was intended to enable further research into semantic features.

One limitation of the dataset is that the features were collected from volunteers, who may not have been able to note every obvious trait for each concept. For example, the feature of "breathes" is applicable to every animal on the list, but it is not mentioned for all of them. This could be because the individuals who collected the data only recorded properties that they could articulate in words, and not every feature was noted. It's important to note that people's mental representation of features is richer than what they can express verbally. That's why I'm infilling the dataset in this project.

The questions, requirements and objectives of this research project are as follows:

- Extend and evaluate deep convolutional neural network models for vision based on the AlexNet architecture. One of them is a unified model, which predicts concepts of objects as well as semantic properties. The other two are independent conceptual and feature models.
- Analyse the performance and internal representation of the models. Compare the unified model with two separate models and compare the activation patterns of the three models.

In conclusion, this project aims to shed light on the limitations of current computer vision models and propose a way to create a better understanding of visual data. It does this by focusing on integrating property norms with conventional object concept labels in training CNN models. The findings and methodologies of this project can serve as a foundation for future research, encouraging a deeper exploration of the synthesis of object recognition and property norm identification in computational vision. The research hopes to influence the ongoing discourse in computer vision by highlighting the importance of a comprehensive understanding of images beyond traditional label-centric approaches.

2.0 System Requirements and Specification

2.1 Solution Description

As the field of artificial intelligence continues to evolve, projects require substantial computational power, demanding high performance from systems. In my project, I have an Intel i9-13900K CPU and an NVIDIA RTX 4090 GPU, which meet the requirements. Therefore, I chose to utilize PyCharm for running in local, debugging, and analyzing my project instead of Google Colab. PyCharm offers several advantages, including robust code analysis features, a user-friendly interface, and seamless integration with various version control systems. These features make it an ideal environment for complex programming tasks, particularly in AI development where efficiency and reliability are paramount.

Neural networks and computer vision represent a broad area of research within deep learning, a field that has seen rapid advancements over the last decade, producing many exemplary cases of image classification. Single-concept classification issues are a popular topic within the field of computer vision. However, my project explores the differences between concept classification (a multi-class problem) and attribute recognition (a multi-label problem). I have employed the classical AlexNet architecture as the foundation for my convolutional neural network, which allows for the evaluation of predictions in scenarios with multiple possible outcomes and the application of various scientific methods to calculate these results.

An effective strategy to address this challenge involves constructing three distinct CNN models for comparative analysis: a Unified Model capable of both single-concept classification and attribute recognition, a Concepts Model focused solely on single-concept classification, and a Features Model dedicated to attribute recognition. Unlike traditional single-concept classification, attribute recognition is trained based on the semantic properties of images to predict potential semantic features, rather than training based on category recognition to predict a single category. Ultimately, a comprehensive analysis of the predictions made by the Unified Model, Concepts Model, and Features Model will be conducted, shedding light on their effectiveness and the nuances between these approaches in handling complex visual data.

2.2 Property Norms Infilling

Before training a predictive model, it is essential to augment the **CSLB Property Norms** dataset to enhance its accuracy. The primary issue with this dataset, derived from features matrices filled out by volunteers, is the frequent lack of comprehensive feature information. Often, participants contribute only the data they deem valuable. For example, within the Property Norms dataset, only six animal types might be marked with the feature "capable of breathing," although this attribute is applicable to all animal concepts.

To address this limitation, I employed the ChatGPT (GPT-4) API to programmatically determine 'yes' or 'no' responses for all features within the Property Norms dataset. This method allows for the creation of a more extensive and accurate set of real features. Additionally, the Property Norms dataset contains 638 concepts and 2,725 features, which results in approximately two million possible concept-feature pair combinations. Testing all these combinations is impractical. However, it is feasible to test for features that are true for one concept but false for a similar one. For instance, the Property Norms dataset may list "*has_a_neck*" as a feature for giraffes but not for elephants, despite both sharing other attributes like having legs and being mammals. Thus, concept similarity can be measured using cosine similarity of concept feature vectors, information stored in the file *cosine_wTax_pf5_long.csv*.

The script *infilling.py* implements the augmentation of the Property Norms using the following algorithm:

Starting with the binarized (many-hot encoded) **CSLB property norms** (P_0), where 1 indicates the presence of a feature and 0 its absence.

Adding all features of P_0 to a new dataset (P_1).

Computing cosine similarity for every pair of concepts in P_0 .

For the N most similar concept pairs:

- a. Identifying pairs of concepts (c_1 and c_2).
- b. Selecting a feature from c_1 that is absent in c_2 :

i. Testing the feature with ChatGPT and adding it to c_2 in P_1 if the answer is yes.

c. Selecting a feature from c_2 that is absent in c_1 :

i. Testing the feature with ChatGPT and adding it to c_1 in P_1 if the answer is yes.

This iterative process can be repeated for P_1 , further expanding the dataset.

This approach enhances the completeness and accuracy of the property norms datasets, contributing significantly to the development of more robust computational vision models. As a result, we have generated an updated feature matrix, stored in the file *updated_feature_matrix.csv*.

2.3 Mapping between CSLB and THINGS

Following the Property Norms Infilling, we have the feature matrix, *updated_feature_matrix.csv*, ready for use. Regarding to the image dataset, I selected the widely applied **THINGS** image dataset, which includes 1,854 concepts and more than 26,000 images. However, the challenge remains to match the concept names from the Property Norms with those in **THINGS**.

The file *THINGS.csv* contains the names of each concept in **THINGS** along with the corresponding image folder names. By utilizing both *updated_feature_matrix.csv* and *THINGS.csv*, we aim to map the concepts between these two datasets effectively.

Due to the geographical and linguistic variations—**CSLB Property Norms** being from the UK and **THINGS** from the US—differences in British and American English terminology can lead to distinct lexical representations for the same concept. To bridge this gap, I employed resources like *ConceptNet*, *WordNet*, and *Datamuse* to acquire potential synonym mappings between the datasets. It is also crucial to include direct lexical matches in this mapping.

By integrating synonym and direct mappings, we obtained a preliminary map, stored in *namemap.json*.

However, this initial mapping might still include some problematic associations, such as slang synonyms that are inappropriate in formal English contexts. Therefore, I implemented the use of the ChatGPT-4 API to refine these mappings.

In the script *refinenamemap.py*, I employed the ChatGPT-4 API to scrutinize the preliminary *namemap.json*, resulting in the refined *updated_namemap.json*. This new mapping file eliminates inappropriate mappings to better align with formal English usage contexts.

Additionally, since the same term can represent different concepts in various contexts, the **THINGS** dataset accounts for this by appending numbers to concept image folder names to differentiate them. For example, 'Chicken' can signify raw chicken sold in supermarkets as '*Chicken1*' and live chickens as '*Chicken2*'. In the **CSLB Property Norms**, 'Chicken' features '*does_cluck*' with a value of 1, indicating a live chicken, hence it corresponds to '*Chicken2*' in **THINGS**. Given the rarity of such scenarios and the necessity to examine images and features for accurate matching, I opted for manual review for this step.

Thus, we achieved the final mapping file, *updated_namemap_verified.json*, which contains 473 key-value

pairs, with the value being the name from **CSLB Property Norms** and the key from **THINGS**.

After obtaining the mapping file, we must filter the relevant images from the complete **THINGS** image dataset (*object_images*) based on the entries in the mapping file.

The script *filter.py* performs this task by searching the *object_images* for subfolders matching the key names in *updated_namemap_verified.json*, copying them to a directory called *filtered_images*, and renaming the subfolders to correspond with the value names for ease of future use with the feature matrix:

filtered_images/apple/xxx.jpg

filtered_images/cat/xxx.jpg

This approach not only ensures that the images are systematically organized but also simplifies their integration with the feature matrix for further analysis or model training.

2.4 Concept Classification

Concept classification tasks require models to train on images from various concept categories and then make predictions using test images. In this project, I employed the AlexNet architectural framework, training from scratch with the newly assembled dataset comprising the THINGS image dataset and CSLB Property Norms dataset. Due to the limited number of images per concept category in the THINGS dataset—typically only a dozen or so, which is significantly fewer than the thousands per category found in the ImageNet dataset originally used with AlexNet—my model ultimately achieved a lower accuracy rate. However, the primary focus of this project is not to optimize accuracy or perform fine-tuning but to analyze and compare differences in activation patterns across three distinct models. This aspect will be explored in detail using Representational Similarity Matrices (RSMs) in subsequent analyses.

In concepts classification, where each image is associated with a single category (as demonstrated by AlexNet's label predictions for eight ImageNet images, where the probabilities of the top five labels are displayed below each image, ranked from highest to lowest), feature classification allows for the identification of multiple attributes in a single image. If the correct label is among these top attributes, its probability bar is displayed in red. Given the multi-label nature of feature classification, the probability distributions are not constrained to sum to one, reflecting the potential for multiple features per image.



[Figure 1: Eight ISVRC-2010 test images and labels considered most probable by the original AlexNet Model]

2.5 Feature Classification

The principle of feature classification is quite similar to that of concept classification, as it involves training models on images to derive an optimal model, which is then used for predictions with test images. However, while concept classification deals with an exclusive multi-class problem where each image is assigned to a single category, feature classification is a multi-label task where an image may exhibit multiple attributes simultaneously. This necessitates differences in the architecture of convolutional neural networks, their connectivity, and the loss functions used, which will be detailed in the subsequent sections of the model analysis.

For training feature classification, we utilize the `updated_feature_matrix.csv`, serving as our feature matrix. This matrix is used in conjunction with a custom dataset, the filtered THINGS image dataset. The dataset is initialized with an object called **CustomImageDataset** and data are loaded through `DataLoader` within a cross-validation loop during training sessions.

In the application scenario of feature classification, convolutional neural networks perform comprehensive analyses based on the set of features each image possesses. However, unlike concept classification, feature classification allows each image to possess multiple feature labels. For example, in concept classification, it might only be necessary to identify that an image depicts a cat. However, in feature classification, the goal is to identify many features of the cat, such as *does_eat*, *does_meow*, *has_a_tail*, *is_a_mammal*, which are objective features, and even more subjective features like *does_cute* or *is_fast*.

The challenge then arises: How can a model deduce these common-sense features and accurately predict an object's characteristics?

A viable approach is to train the model using the concept-to-feature mappings from the **updated_feature_matrix.csv** along with the given image dataset.

The original architecture of AlexNet was designed for solving a problem where there is only one correct class label per image. However, it is not suitable for multi-label feature classification tasks. To address this, I have modified the connection layers of the model and adapted the loss function to enable multi-label outputs.

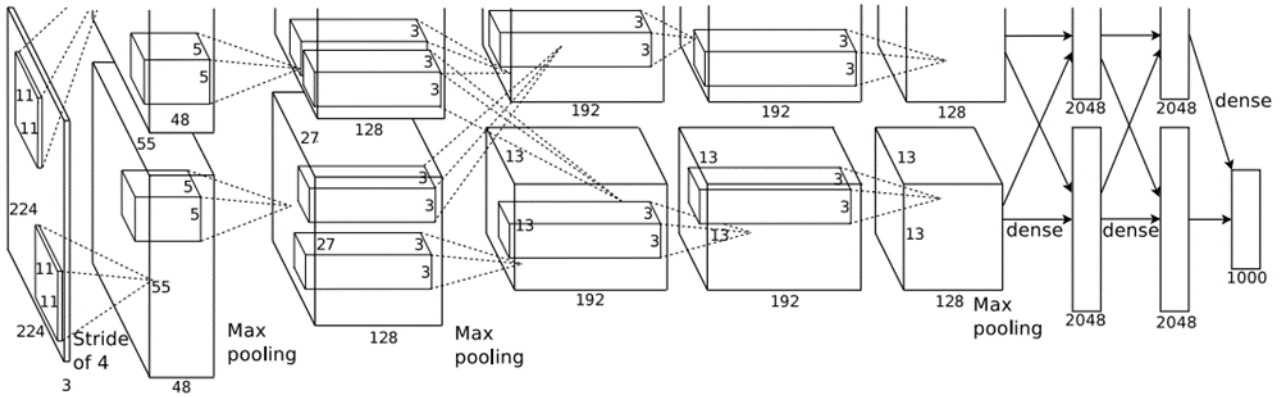
By updating AlexNet for feature classification, the model can now identify the presence of multiple features in a single image. This enhancement enhances the model's predictive versatility and makes it more practical for complex real-world tasks where multiple characteristics need to be identified simultaneously.

2.6 System Requirements

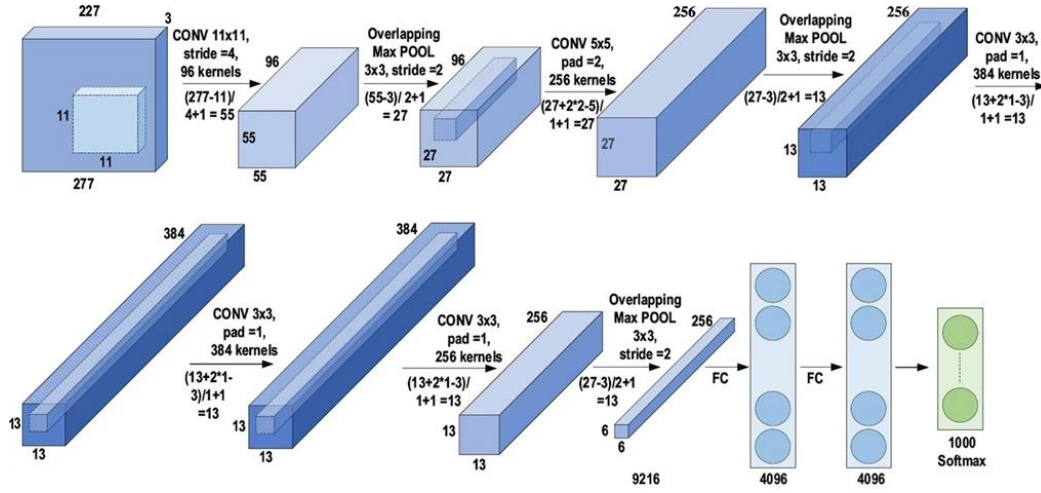
- The Unified Model should be able to predict both concept categories mapped between **CSLB** and **THINGS** and identify multiple features of objects from the **CSLB** feature matrix. This model can be trained on a set of images for both multi-class classification (concept classification) and multi-label classification (feature classification).
- The Concept Classification Model should be able to concept categories mapped between **CSLB** and **THINGS**. This model is designed to train on multi-class classification (concept classification) using a set of images and to visualize the accuracy of given examples post-training.
- The Feature Classification Model targets the features listed in the **CSLB** feature matrix, and it should be able to identify multiple features of objects.
- An in-depth study and analysis of significant layers within these three different models are conducted, with a focus on comparing their differences, especially regarding activation patterns. To facilitate this analysis, Representational Similarity Matrices (RSMs) will be utilized.

3.0 Design

3.1 AlexNet Architecture



[Figure 2: AlexNet Architecture]

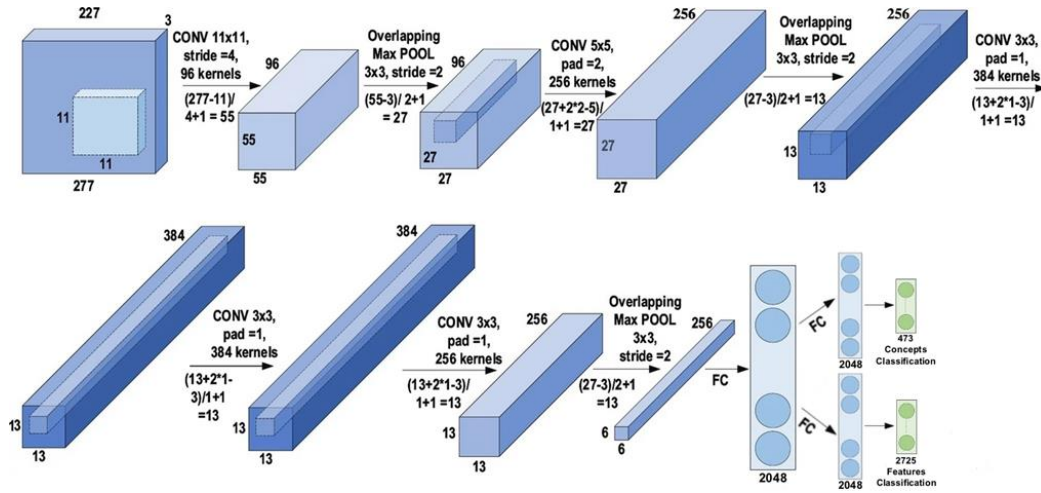


[Figure 3: One Part of AlexNet Architecture]

The original architecture of AlexNet, as illustrated, was uniquely designed to be trained across two Nvidia GeForce GTX 580 GPUs, which facilitated the parallel processing of its network layers. This setup divided the network into two identical parts, each running on one GPU; however, for understanding purposes, only one segment needs to be considered as shown in the diagram.

In this project, I have adopted the foundational structure of AlexNet but have implemented modifications tailored to the specific needs of this project, particularly in adapting to the differing number of outputs. While AlexNet was primarily engineered for a fixed number of classes (1,000 classes from the ImageNet challenge), the adaptation in this project involves accommodating the unique demands of both concept classification and feature classification.

3.2 Unified Model Architecture

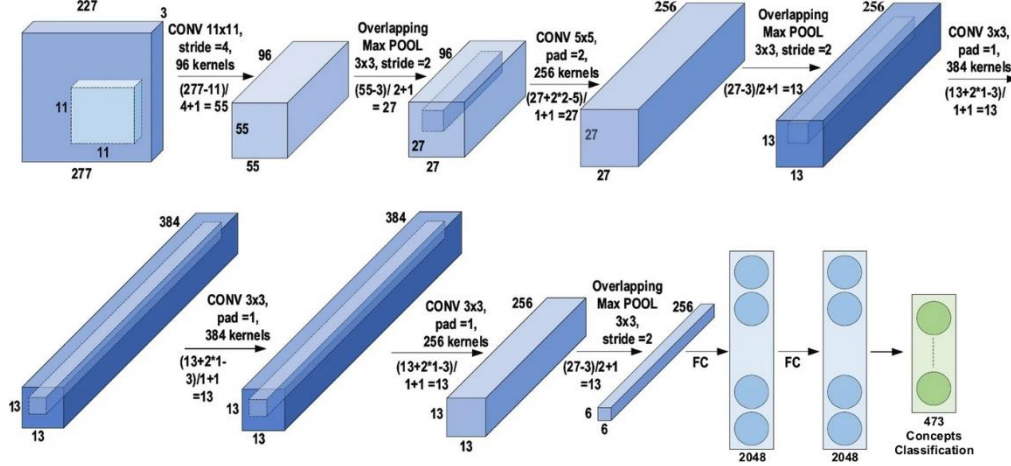


[Figure 4: Unified Model Architecture]

In the unified model designed to handle both concept classification and feature classification, the architecture bifurcates after the first fully connected (FC) layer into two separate FC layers, each dedicated to a specific type of output. One of these layers is designated for concept classification, outputting results for a total of 473 concept categories. The other layer addresses feature classification, with outputs tailored for 2,725 distinct attribute categories.

This approach allows the model to efficiently multitask, classifying concept categories and identifying semantic features.

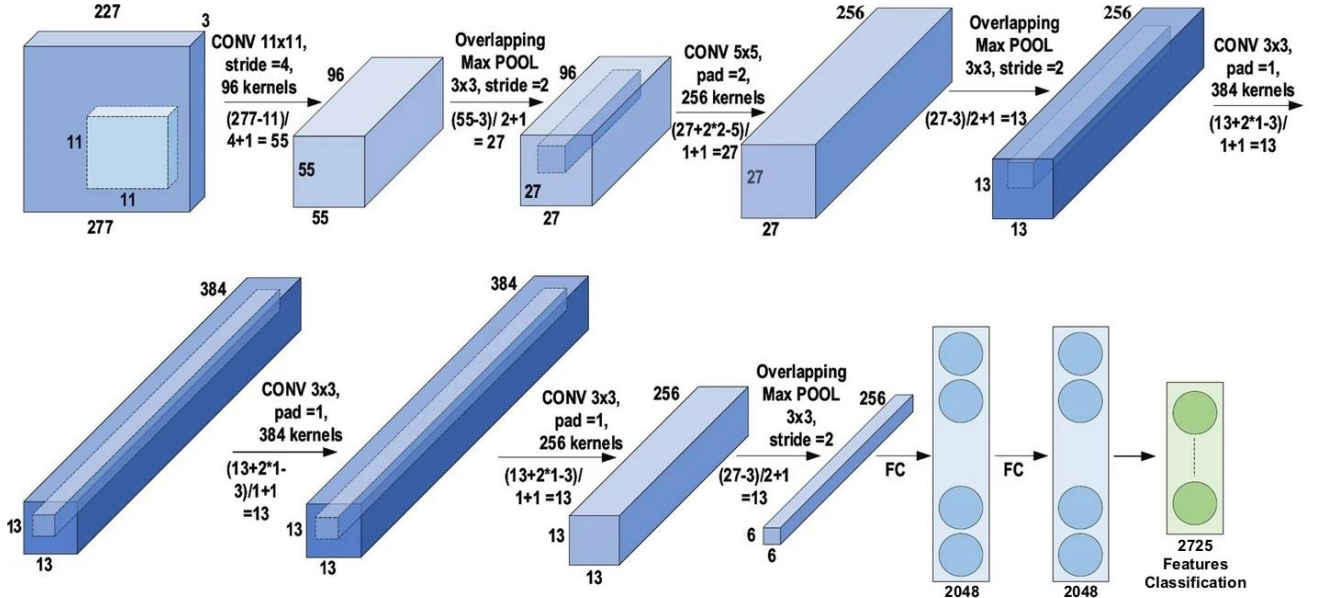
3.3 Concept Model Architecture



[Figure 5: Concept Model Architecture]

The separate concept classification model focuses on categorizing images into concept categories. To align the model's architecture with the specific requirements of concept classification, I adjusted the second fully connected (FC) layer to output the exact number of concept categories, which is 473. This adjustment is crucial to ensure that the output layer precisely matches the number of concept categories in this dataset.

3.4 Feature Model Architecture

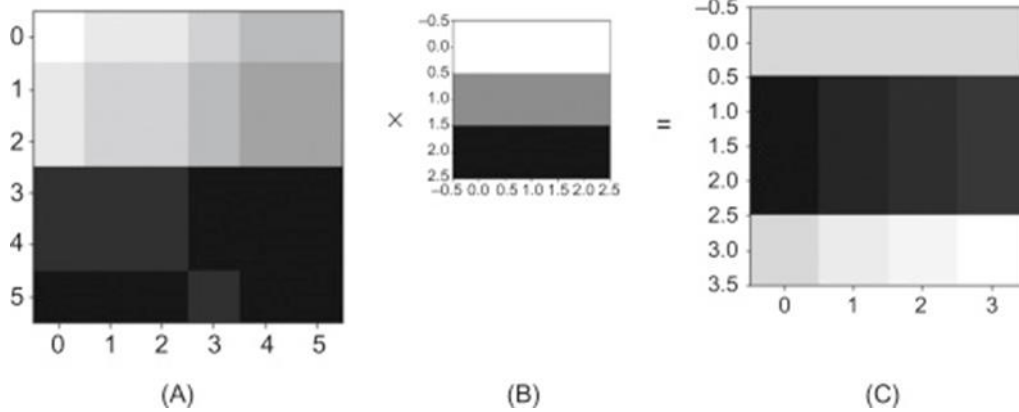


[Figure 6: Feature Model Architecture]

In the separate feature classification model, the sole objective is to identify multiple features of each image. Therefore, I adjusted the second fully connected (FC) layer specifically for this task by setting the output size to match the total number of feature categories (2,725). This modification allows the model to output a

prediction for each feature, facilitating a multi-label classification approach.

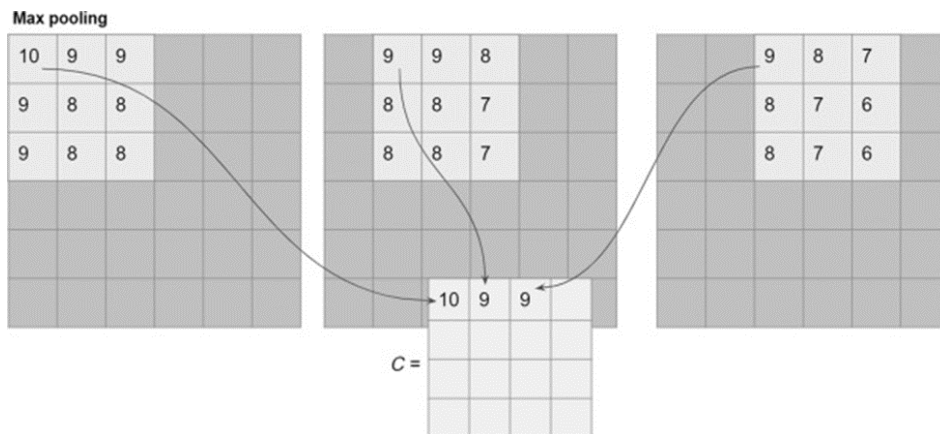
3.5 Convolutional Layer



[Figure 7: Convolutional Layer]

The models' architecture consists of five convolutional layers that form the basis for extracting image features. These layers use filters to convolve the input images, encapsulating the feature information into smaller matrices called feature maps. The filters can recognize various image attributes, such as vertical lines, horizontal lines, and edges. After convolution, we get a series of feature maps where higher activation values indicate a strong presence of the targeted features in the image.

Convolutional neural networks (CNNs) have layers equipped with filters that are arranged in a stacked manner. Each subsequent layer computes its output based on the feature maps provided by the previous layer. In this way, the network captures complex features that are combinations of simpler patterns detected by earlier layers. As the number of layers increases, the depth of the network also increases, enabling it to detect more abstract and sophisticated features. However, deeper networks with more filters can introduce challenges such as overfitting and unmanageable weight complexity.

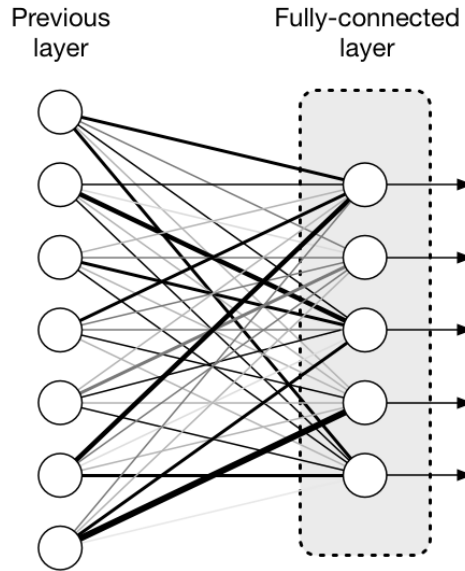


[Figure 8: Max Pooling]

To address certain issues in data representation, we use a technique called max pooling. This technique works by scanning the feature map and keeping the highest activation within a given area, which reduces the representation's size. This helps to highlight the most important features, decrease the data's dimensionality, and improve the network's ability to handle input variations and noise. As a result, not only does the

computational load decrease but the models' ability to generalize from training data to new, unseen datasets also increases. This makes it more practical and effective in real-world applications.

3.6 Fully Connected Layer



[Figure 9: Fully Connected Layer]

In the architecture of the network, the last two or three layers are fully connected layers, which come after the initial five convolutional layers dedicated to feature recognition within the images. These fully connected layers have access to all the activations across the network and compute their own set of weights. This capability allows them to interpret which category the extracted features by the convolutional layers belong to. Essentially, the fully connected layers integrate the learned features, providing high-level reasoning about the content of the images.

Each neuron in a fully connected layer is connected to all the activations from the previous layer, transforming localized and abstracted feature data from the convolutional layers into final class scores.

Convolutional neural networks use a structured approach, where convolutional layers detect features and fully connected layers classify them. This approach enables the network to effectively learn hierarchical representations of complex images and improve its ability to categorize them accurately. Additionally, this architecture enhances the model's ability to generalize across different visual datasets.

3.7 Loss Function

In the training phase, I utilize the `nn.CrossEntropyLoss()` function and an integrated **Softmax** for the concept classification task. This ensures that the model learns to output the correct probability distribution. This loss function, CrossEntropyLoss (CELOSS), is one of the most commonly used loss functions in classification problems because it measures the difference between the probability distribution predicted by the model and the distribution of the true labels.

The cross-entropy loss function is based on the concept of cross-entropy in information theory and is used to

measure the difference between two probability distributions. In classification problems, it is used to measure the difference between the probability distribution p output by the model and the true distribution q , which is usually a uniquely hot coded vector. In multicategorization problems, the cross-entropy loss function is often used in conjunction with the **Softmax** activation function, which converts the raw outputs of the model (also known as logits) into probability distributions. The reason for choosing the cross-entropy loss function as the optimization objective in conceptual classification tasks is due to its unique properties. First, the cross-entropy loss function is able to force the model to learn a distribution that accurately reflects the probability of each category, which is crucial for the classification problem because it requires the model to not only correctly predict the categories, but also assign reasonable probabilities to each possible category. Second, the **Softmax** function used in conjunction with the cross-entropy loss function provides a nonlinear activation mechanism that helps the model capture complex relationships in the data and enhances its ability to learn complex patterns. In addition, the cross-entropy loss function provides a good gradient for the gradient descent algorithm, which is crucial for the training efficiency of the model, as a clear gradient helps the model to quickly converge to the optimal solution. At the same time, the cross-entropy loss function is scale invariant with respect to the size of the original values output by the model, which ensures the stability of the loss calculation. Finally, the cross-entropy loss function is designed to avoid the gradient vanishing problem, which ensures that the model can continuously and effectively learn during the training process. Therefore, the cross-entropy loss function is ideal for use in conceptual classification tasks, and it helps the model to continuously improve during the training process through these properties, ultimately achieving highly accurate classification performance.

As multi-label classification problem, the standard practice typically involves using a binary cross-entropy loss function (**BCELoss()** or **BCEWithLogitsLoss()**, which incorporates a sigmoid activation function). However, I have chosen to use the Mean Squared Error Loss (**MSELoss()**) as an experimental choice. This decision aims to optimize the squared differences between the predicted values and the actual labels. By using **MSELoss**, the model may potentially offer advantages in terms of prediction accuracy, particularly when absolute precision in prediction is crucial.

This experimental approach with **MSELoss** for multi-label tasks is a little unconventional, as **MSELoss** does not inherently manage the probabilistic nature of label predictions as **BCEWithLogitsLoss** does. Instead, it treats the problem as a regression. This could be beneficial in scenarios where the labels themselves are of continuous nature, or when fine-grained differences between predicted and actual values are significant. By focusing on minimizing the square of the error, the model can more finely tune the predicted outputs to closely align with the observed data. This is particularly valuable in complex attribute recognition scenarios where each attribute can vary independently, and it's not merely present or absent. Besides, in this feature classification task, labels are not totally independent, they are relevant to each other. I will discuss the selection of the loss function in more detail in the subsequent sections.

For example, consider a multi-label image categorization task where the labels represent different age groups such as "**child**", "**teenager**", "**adult**" and "**elderly**". These labels are not independent of each other, but have a recursive relationship, i.e., an image may be labeled as "**adult**" and "**elderly**" at the same time, but should not be labeled as "**child**" or "**teenager**". or "adolescent". In this case, the advantage of using **MSELoss** over **BCELoss** becomes apparent.

MSELoss better captures the recursive relationship between labels by minimizing the squared difference between the predicted and actual labels. The model learns that when an image is labeled as an "adult", it is more likely to be labeled as an "**elderly**" at the same time, while it is more likely to be labeled as "**child**" or "teenager". "teenager" is less likely to be labeled as such. In this way, the model is able to more accurately predict the age range of an image, rather than just categorizing it as a single label. In contrast, **BCELoss** treats each label as a separate binary classification problem and fails to capture the recursive relationships between labels. This may lead to inconsistent results in the model's predictions, such as labeling an image as both "child" and "elderly", which is not reasonable in practice.

Therefore, when there is a recursive relationship between classification labels, using **MSELoss** provides more accurate predictions than **BCELoss**. By fine-tuning the prediction output, the model is able to better capture the correlation between labels, thus improving the performance of the classification task.

3.8 Metrics

In the concept classification task, I used standard metrics - Loss rate and Accuracy - to evaluate the performance of the model. These metrics are well suited for evaluating how accurately the model categorizes images into their respective categories.

However, in the case of multi-label classification, it is not feasible to calculate the accuracy directly. In multi-label classification problems, each sample may belong to multiple categories at the same time, so the traditional accuracy rate metric is no longer applicable. Accuracy rate usually refers to the ratio of the number of samples correctly predicted by the model to the total number of samples, but in multi-label classification, a sample can have multiple correct labels, which leads to the fact that the Accuracy rate cannot fully reflect the performance of the model.

For example, in a multi-label classification problem with 2,000 categories, if most of the categorical labels are 0 (i.e., most of the samples do not belong to any of the categories), the use of accuracy as an evaluation metric may be misleading. Accuracy is the number of all correctly predicted labels divided by the total number of all predicted labels. In this case, a simple model that always predicts label 0 (i.e., the predicted samples do not belong to any category) may achieve a high accuracy rate, but this does not mean that the model performs well because it may not identify any positive class labels. Due to the complexity of multi-label categorization, using accuracy alone does not provide comprehensive information about model performance. Especially when the categories are unbalanced, the accuracy is biased towards the majority class. In the case of high category imbalance, the model may just learn to predict the most common labels and ignore the prediction of a few categories.

In order to better evaluate the performance of multi-label classification models, we usually use two metrics, Precision and Recall. Precision refers to the proportion of samples predicted by the model to be positive classes that are actually positive classes, while Recall refers to the proportion of samples that are actually positive classes that are correctly predicted by the model to be positive classes. Precision rate and Recall rate can provide a more comprehensive picture of the model's performance in multi-label classification problems, as they focus on the model's ability to predict positive classes and identify positive classes, respectively.

In summary, the Precision and Recall rates are more appropriate metrics to evaluate in a multi-label classification task because they can more comprehensively reflect the model's performance when dealing with samples with multiple labels. Whereas traditional accuracy metrics do not provide effective evaluation information in such tasks.

3.9 Error, Event and Exception Handling

Given the script's objective to process a dataset for neural network training, the error handling analysis focuses on the potential exceptions that could be raised during file and directory operations. The implemented error handling logic provides a robust framework that anticipates and addresses the primary issues that may arise in such a file system-centric task.

JSON File Processing:

The script commences by loading a `namemap` JSON file. This operation is encapsulated in a try-except block that handles `FileNotFoundError`, to account for the possibility that the specified `namemap` file might not be present at the given location, and `json.JSONDecodeError`, to ensure that the contents of the `namemap` file are in valid JSON format. The use of different exceptions for various error conditions enables specific and informative error messages. If an error condition is met, the script is terminated to prevent further issues.

Directory Operations:

After ensuring the JSON data is reliable, the script moves on to validate the presence of the target directory for filtered images. To prevent the script from crashing due to insufficient permissions for creating new directories, the `os.makedirs` function is enclosed within a try-except block that catches any ***PermissionError***. This approach helps the user to understand the exact nature of the problem and prevents the script from ending abruptly.

File and Directory Looping:

The script's primary function is to iterate through the subfolders within the dataset image directory and map each subfolder to a new directory based on the ***namemap***. This process is protected within a try-except block that handles both ***FileNotFoundError***, in case the source directory does not exist, and ***PermissionError***, in case the script lacks the necessary permissions to read the directory contents. By addressing these potential issues, the script ensures that the source data is available and accessible before attempting to process it.

File Copying Operations:

During the execution of the script for mapping and copying files, each copy operation is safeguarded by a ***try-except*** block that handles ***shutil.Error***. This is crucial to detect and handle any issues that may arise during the file copying process, such as hardware I/O errors, filesystem errors, or unexpected problems with the files themselves. In case of such exceptions, the script logs an error message that includes the source and destination of the failed copy operation. This ensures targeted troubleshooting without hindering the overall process.

Summary of Error Handling Strategy:

The script has a well-structured and comprehensive error handling strategy, which ensures that exceptions are gracefully managed without causing unexpected terminations or unhandled errors. This strategy contributes to

the script's reliability and user-friendliness, making it a resilient component in the dataset preparation stage for neural network training. The systematic approach to error handling demonstrates a thorough understanding of potential issues and showcases best practices in Python scripting for robust application development.

4.0 Implementation

4.1 Implementation Descriptions: Languages and Environments

Python is widely used in the field of artificial intelligence, and this project was implemented using Python. Python provides several machine learning libraries and packages that simplify many complex steps with pre-packaged functions.

In this project, I primarily used PyTorch, a specialized tensor library optimized for deep learning, which ranks as one of the most commonly used libraries in the deep learning sphere, alongside TensorFlow. I chose PyTorch over TensorFlow because it handles version compatibility issues with Keras more straightforwardly, which is frequently encountered with TensorFlow.

PyTorch is easily integrated with other Python packages and supports efficient memory management and GPU utilization. Utilizing GPUs can significantly reduce the training time for networks, which is an essential factor in deep learning. The PyTorch framework includes multiple components, such as the torch package, which provides a multi-dimensional tensor data structure and a suite of useful utility functions. For example, the torch package supports CUDA tensors, enabling computation on GPUs, providing the same functionalities as CPU tensors, but with higher efficiency. The statement `device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")` dynamically specifies the execution environment for tensor computations, automatically choosing between GPU or CPU based on system configuration. This approach to device management enhances code portability and flexibility, allowing the same code to run on different hardware setups without modification. When CUDA devices are available, tensors, and models leverage the GPU's parallel processing capabilities for efficient computation. If unavailable, they revert to CPU execution, ensuring compatibility and stability of the application. This use of an abstraction layer for device management is one of the key practices for efficient training of deep learning models.

4.2 Software Libraries

Package & Libraries	Description / Reasoning
Torch	Core library for tensor computation and deep learning, providing a flexible platform for research and development.
torch.nn	Provides modules and classes to create and train neural networks, encapsulating weights and computation.
torch.optim	Offers optimization algorithms like SGD and Adam for updating weights and biases during model training.
torch.utils.data.Dataset	An abstract class allowing custom data handling by

	defining how data items are fetched.
<code>torch.utils.data.DataLoader</code>	Combines a dataset and a sampler, providing an iterable over the given dataset with batching, sampling, and shuffling.
<code>torch.cuda.amp.GradScaler</code>	Manages automated mixed-precision scaling to maintain the balance between speedup and numerical stability.
Torchvision	A companion package for Torch that provides pre-trained models and common image transformations for computer vision.
<code>torchvision.transforms</code>	A collection of image transformations that can be composed into a pipeline to preprocess data before training.
SKLearn	A comprehensive machine learning library offering various tools for modeling and data analysis.
<code>sklearn.model_selection.StratifiedKFold</code>	Splits the data in a stratified manner, ensuring that each fold has a representative ratio of class samples.
<code>sklearn.metrics.mean_squared_error</code>	Measures the average squared difference between estimated values and the actual value, used in regression analysis.
<code>sklearn.metrics.pairwise.cosine_similarity</code>	Computes the cosine similarity between samples in n-dimensional space, useful for comparing vectorized data.
NumPy	Provides support for multi-dimensional arrays and matrices, along with a large collection of mathematical functions.
Pandas	Offers data structures and operations for manipulating numerical tables and time series, making data analysis more accessible.
Math	Supplies a wide array of mathematical functions, such as trigonometric calculations, to support complex calculations.
PIL (Python Imaging Library)	Enables opening, manipulating, and saving many different image file formats, vital for image preprocessing.
Matplotlib	A plotting library for creating static, interactive, and animated visualizations in Python.
Seaborn	A data visualization library based on Matplotlib that provides a high-level interface for drawing attractive and informative statistical graphics.
Random	Implements pseudo-random number generators and provides functions to generate random data.
Json	Enables encoding and decoding data in the JSON format, a lightweight data interchange format.

OS	Provides a portable way to use operating system-dependent functionalities like reading or writing to a filesystem.
Shutil	Offers high-level file operations such as copying and archiving.
OpenAI	Provides interfaces to OpenAI APIs or tools, which might be used for leveraging external AI models or functionalities.
Requests	Allows sending HTTP requests using Python, enabling communication with web services.
NLTK (Natural Language Toolkit)	A leading platform for building Python programs to work with human language data, essential for text processing and analysis.
nltk.corpus.wordnet	A lexical database for the English language that helps scripts understand word meanings and relationships.
UnitTest	The built-in Python framework for conducting automated tests to ensure code correctness.

These libraries provide a comprehensive toolkit for constructing, analyzing, and deploying machine learning models. They are specifically designed for tasks involving image processing, data manipulation, and pattern recognition in neural network architectures.

4.3 Important Functions / Algorithms

4.3.1 Custom Dataset: CustomImageDataset

The class inherits from *torch.utils.data.Dataset*, which is an abstract class representing a dataset and includes the methods `__init__`, `__getitem__`, and `__len__`.

`__init__` Method:

Commonalities:

- All models' `__init__` methods accept `img_dir` (image directory) and `transform` (image transformation operations) parameters.
- Each model iterates over the image directory to build a list containing image paths.

Differences:

- Unified Model:
 - Receives both `feature_vectors` and `namemap`, using them to construct a list that includes image paths, concept indices, and feature vectors.
 - Utilizes the keys of `namemap` to create a mapping from concepts to indices.
- Concept Classification Model:
 - Only receives `namemap`, used to create a mapping from concept names to indices.

- The list only stores image paths and concept names, not involving feature vectors.
- Feature Classification Model:
 - Receives `feature_vectors` and `namemap`, but only includes feature vectors for concepts that exist in `namemap`.
 - The list includes image paths, concept indices, and filtered feature vectors.

`__getitem__` Method:

Commonalities:

- All models' `__getitem__` methods read the image path from the list, open the image, and apply the image transformation if provided.

Differences:

- Unified Model:
 - Returns the image, concept label index, and feature vectors, supporting training for both concept and feature classification simultaneously.
- Concept Classification Model:
 - Only returns the image and concept label index, focusing on the task of concept classification.
- Feature Classification Model:
 - Returns the image, concept index, and feature vectors, focusing on classifying the attributes of the images.

These design differences reflect the specific needs of each model. The unified model needs to handle both concept and feature classification tasks, making its data structure and return values the most complex. The concept classification model and feature classification model are more specialized, dealing with concept recognition and feature recognition respectively, with relatively simpler data structures and return values that are precisely tailored to their specific tasks. This tailored design ensures efficient data handling and targeted model training.

4.3.2 Modified Training Process

4.3.2.1 Selection of the loss function

As previously mentioned, adjustments were made to the loss function during the training process to accommodate the functional requirements of the current project.

For the concept classification task, which is a multi-class classification, I utilized the standard cross-entropy loss function.

In PyTorch, ***CrossEntropyLoss*** is a common loss function for multi-class classification tasks. It effectively combines the ***LogSoftmax*** and ***NLLLoss*** (Negative Log Likelihood Loss) in one single class, offering an efficient and numerically stable computation method. Specifically, ***CrossEntropyLoss*** calculates the loss between the log probabilities of each class and the corresponding actual labels, enabling the model to accurately predict the class of each input sample.

Using CrossEntropyLoss has the following advantages in multi-class classification problems:

Probabilistic interpretation: cross entropy loss provides a probabilistic interpretation of the output. It encourages the model to output a probability distribution that closely matches the true distribution (represented by the uniquely hot coded labels).

1. **Gradient property:** cross-entropy loss has a good gradient property. It provides the model with a clear signal of the gradient, which helps the optimization process. When the model's prediction is correct, the gradient is smaller; when the prediction is wrong, the gradient is larger, which helps to learn faster.
2. **Handling unbalanced data:** Cross entropy loss can handle unbalanced datasets to some extent. As it works with probability, it can reduce the weight of overrepresented classes in the data and give more attention to a few classes.
3. **Robustness to Noise:** Cross entropy loss is robust to noise in the data. It is less sensitive to outliers or mislabeled examples than other loss functions (e.g., mean square error).
4. **Simplicity and efficiency:** By combining LogSoftmax and negative log-likelihood loss, CrossEntropyLoss simplifies the implementation and makes the computation more efficient because it reduces the number of necessary operations.
5. **Prevents overfitting:** by its very nature, CrossEntropyLoss encourages the model to remain confident in its predictions. This helps prevent overfitting because the model is encouraged to make more certain predictions that may fit the training data well, but do not generalize to unseen data.

The mathematical expression is as follows:

$$Loss(x, class) = -\log \left(\frac{\exp(x[class])}{\sum_j \exp(x[j])} \right) = -x[class] + \log \left(\sum_j \exp(x[j]) \right)$$

[Figure 10: Cross Entropy Loss Formula]

where C is the total number of classes, and y_i is the actual target label, where $y_i=1$ if the sample belongs to class i, and 0 otherwise (this is a one-hot encoded vector).

P_i is the probability predicted by the model that the sample belongs to class i, obtained by applying the softmax function to the logits (i.e., the raw prediction values from the model).

In practice, it's not necessary to explicitly apply the softmax function; CrossEntropyLoss takes care of this step automatically. Additionally, the function typically expects logits to be a two-dimensional tensor with the shape [N,C], where N is the number of samples in the batch, and C is the number of classes. The corresponding target labels should be a one-dimensional tensor of shape [N], containing the class indices for each sample.

Above all, using **CrossEntropyLoss** allows direct acquisition of the probability distribution of predicted categories from the model's final output layer (usually an unactivated linear layer, in this case, a fully connected layer) and calculates the error relative to the true labels, thus effectively training multi-class classifiers. This loss function is one of the standard choices for training deep

learning models for classification tasks due to its combination of probabilistic interpretation and automated optimization convenience.

For the feature recognition task, which is a multi-label classification, as previously mentioned, I did not follow the standard practice of using loss functions such as *BCELoss* or *BCEWithLogitsLoss* but instead used the *MSELoss* function. The following will analyze the advantages of using *MSELoss* over traditional *BCELoss* for the feature recognition application in this project:

- **The labels are not entirely independent but have some associations.**

The Binary Cross-Entropy (BCE) loss function essentially decomposes a multi-label classification problem into multiple independent binary classification tasks. Each label's classification is considered a separate binary task, with the BCE loss function evaluating the prediction for each label individually. When using *BCELoss*, each label is regarded as independent.

This loss function is very effective for traditional multi-label classification problems. However, in this project, the feature labels are not completely independent. Each feature label represents different characteristics of the image (such as "*has_a_tail*," "*is_biotic*," "*does_eat*," "*does_growl*," "*is_cute*," etc.). These labels are related to some extent and are not entirely independent. The relationship between the labels can be considered from several perspectives:

Semantic Relevance: Some labels naturally have semantic connections. For example, "*has_a_tail*" and "*is_biotic*" are often co-present because most entities with tails are also biological. Similarly, "*does_eat*" and "*is_biotic*" are highly related since all biological entities need to consume food.

Logical Dependency: The presence of certain labels may logically depend on the existence of other labels. For example, the feature "*does_growl*" is usually associated with animals, establishing a direct link with the "*is_biotic*" label.

Co-occurrence of Labels: In many cases, some labels tend to appear together, indicating some degree of dependency between them. For instance, an image describing an animal might simultaneously have the labels "*is_cute*" and "*has_a_tail*."

Therefore, although technically each label could independently represent a feature, in practical applications, these labels are often interrelated. This interconnection means that the dependencies and interactions between labels could have a significant impact on the model's training and prediction.

- ***MSELoss* outputs specific confidence levels, which are more intuitive.**

The use of Mean Squared Error (MSE) as a loss function allows the model to express confidence levels quantitatively for each feature label prediction. Unlike binary classification tasks where the output is typically a binary value indicating the presence or absence of a feature, MSE expresses this as a continuous value. This continuous nature of the output from *MSELoss* aligns with the inherent uncertainty and variability in multi-label feature recognition, providing a more granular

understanding of the model's predictions.

MSELoss does not assume independent label predictions. Given that feature labels in the context of this project show varying degrees of correlation and dependency, **MSELoss** caters to these nuances by treating the task more as a regression problem. It takes into account the relationships between features and adjusts the model's confidence levels accordingly.

The project uses a technique called **MSELoss** for multi-label classification. This approach considers the complex relationships between labels and provides a confidence score that represents the model's certainty in its predictions. By doing so, it creates a more accurate error gradient that can help improve the model's performance during training. This method is better suited to the complex nature of the data and can potentially lead to better model tuning.

The use of **MSELoss** in multi-label classification not only provides a more intuitive representation of confidence levels but also allows for a more nuanced evaluation of the model's performance. The continuous nature of the output from **MSELoss** enables the model to express varying degrees of certainty in its predictions, which can be particularly useful in scenarios where the correctness of a prediction can vary in significance.

For instance, in the context of the project, an image may contain an animal with a tail, but the tail may not be clearly visible or identifiable. The model's prediction for the "**has_a_tail**" feature may not be as confident as it would be for a feature like "**is_biotic**," which is a more fundamental characteristic of the entity in the image. **MSELoss** allows the model to express this uncertainty by providing a lower confidence score for the "**has_a_tail**" prediction compared to the "**is_biotic**" prediction.

Moreover, the use of **MSELoss** in a multi-label classification setting allows for a more robust evaluation of the model's performance. By considering the relationships between labels and providing a confidence score for each prediction, **MSELoss** can help identify areas where the model may be underperforming or where additional training data may be needed.

In conclusion, the use of **MSELoss** in multi-label classification provides a more intuitive and granular understanding of the model's predictions. It caters to the complex relationships between labels and enables the model to express varying degrees of certainty in its predictions. This approach can lead to better model tuning and improved performance in real-world scenarios.

4.3.2.2 K-fold Cross Validation

In order to evaluate the performance of our deep learning models in classifying multiple categories and labels, I employed the cross-validation technique. This method involves dividing the dataset into different sets for multiple rounds of training and validation. By doing so, I can reduce the chances of overfitting on a single sample and improve the accuracy of the model when predicting new data.

Initially, I used:

```
from sklearn.model_selection import KFold
```

However, due to the variability in the number of images per category in the **THINGS** dataset, this method resulted in an uneven distribution of images across categories in each fold.

Hence, I adjusted to:

```
from sklearn.model_selection import StratifiedKFold
```

StratifiedKFold ensures that each fold is a good representative of all the classes. It maintains a consistent distribution of categories across each training and validation set, making it better suited for handling datasets where classes are unevenly represented.

Dataset and Model Preparation

The dataset includes various concepts and features annotated through images. To perform a complex task with 473 concepts or 2,725 features (or both in the Unified Model), I utilized customized AlexNet models. To ensure the independence of validation, each model training begins from randomly initialized weights.

Cross-Validation Process

The dataset for this project is divided into five folds, as specified by the variable *num_folds*. For each fold, a portion of the data is randomly chosen as the validation set, while the rest is used for training. This ensures that every data point is used for both training and validation, which provides a comprehensive assessment of the model's performance.

Before using each fold, appropriate preprocessing is carried out, including image resizing and normalization, to satisfy the network's input requirements. The data loaders for training and validation sets are configured with different batch sizes and shuffling options.

Model Training and Validation

The models are trained using the *Adam* optimizer, with careful adjustments made to the learning rate and weight decay. During each training epoch, the model undergoes forward and backward propagation on the training set to compute the loss, which includes concept and feature losses. The models' performance is monitored by tracking loss values, accuracy, precision, recall and Root Mean Squared Error (RMSE).

Once each training cycle is complete, the models are switched to validation mode to evaluate their performance on the validation set. It calculates the corresponding loss, accuracy, and RMSE for feature predictions.

Performance Evaluation

After each fold in the training process, the results are recorded for further evaluation. This allows for the

calculation of average performance metrics, including average training loss, average validation loss, average training accuracy, and average validation accuracy, across all folds upon completion.

The evaluation process is rigorous and thorough, utilizing stratified cross-validation and detailed performance tracking. This ensures that the model's effectiveness in handling complex classification tasks in deep learning is validated impeccably.

4.4 Representational Similarity Matrix (RSM)

In order to analyse the way visual features are represented within deep neural network models, I utilized the Representational Similarity Matrix (RSM) techniques. This method measures the similarities between activation patterns of various stimuli instances, which helps in understanding how the model processes and abstracts visual information.

RSM for each model

Model and Data:

I used modified versions of AlexNet, and the models were trained on a wide-ranging dataset comprising images of objects possessing 473 concept labels and 2,725 semantic features. To perform the analysis, I loaded the models with trained weights and set them in evaluation mode to ensure that the activations reflected the learned features without any further adaptation.

Dataset Preparation:

Considering the impracticality of displaying all 473 concepts in the RSM, I randomly selected 40 concepts from the comprehensive naming map to ensure a diverse and representative visual category selection. Each concept corresponded to a specific directory name within our image storage structure, which simplified the retrieval process during data loading. Images were resized to 224x224 pixels, normalized, and converted into tensor format using PyTorch's transformation tools.

Activation Extraction and RSM Calculation:

As batches of images passed through the model, I extracted activations. For each layer from which activations were collected, a representational similarity matrix was calculated. This matrix was obtained by flattening the activation tensors into vectors and computing the cosine similarity between these vectors. Cosine similarity measures the cosine of the angle between two vectors, serving as a measure of their orientation similarity in multi-dimensional representational space.

RSM Visualization:

To analyze the relationship between different concepts in a given layer, a heatmap was generated for each RSM using Seaborn's heatmap function. The heatmap provided a visual representation of pairwise similarity scores between all selected concepts, and was labeled with concept names on both axes. Furthermore, the

RSM files were saved in graphical (PNG) and numerical (*NumPy* array) formats for future analysis.

Second-Order RSM

In order to better understand how different models or layers encode information in their representational spaces, I utilised a technique called Second-Order Representational Similarity Matrix (RSM). This method involves comparing the similarities between first-order RSMs of different models or layers, which provides a quantitative way of comparing the learned features across different architectures.

Data Preparation and Loading:

Initially, I defined paths for storing the RSM (Representational Similarity Matrix) files of each model. These paths include "unified" for the integrated model, "concepts" for the concept model, and "features" for the feature model. Then I used the "*load_rsms_from_directory*" function to load the RSM files of these models from the specified directories. The RSM files were pre-saved in NumPy format. This function reads all files in the directory that have the ".*np*y" file extension and loads them as NumPy arrays.

Second Order RSM Calculation:

After loading all first-order RSMs, we initialized a second-order RSM matrix, its dimensions based on the number of RSMs loaded. For each RSM of every model layer, we computed its cosine similarity with all other RSMs by comparing their upper triangular matrix elements (excluding the diagonal). Diagonal elements (i.e., a model compared with itself) were set to 1.0, indicating perfect similarity.

Visualization

Using seaborn and matplotlib, heatmaps were created to visualize these similarities. The heatmaps' horizontal and vertical labels represent combinations of different models and layers, allowing us to visually assess the similarities and differences between layers and models. This visualization aids in identifying layers or models with similar or significantly different encoding strategies or representational features.

5.0 Testing

5.1 AlexNet Architecture

To address the specific challenges of this project, modifications were made to the original AlexNet architecture, particularly to its last fully connected layer, to accommodate the input features from the preceding layers and output the required number of categories specific to our dataset. These adaptations ensure that the system requirements outlined in Section 2.6 are both testable and have been rigorously tested to confirm their effectiveness. Below are the detailed changes and verifications for each model variant:

Unified Model:

Modification: The original final fully connected layer was removed, and the penultimate fully connected layer was connected to two parallel fully connected layers to handle separate tasks simultaneously.

Implementation:

```
self.classifier_concepts = nn.Linear(2048, num_concepts)
self.classifier_features = nn.Linear(2048, num_features)
```

These layers output labels for the concepts and features categories, respectively.

Concept Model:

Modification: The output of the last fully connected layer was changed from the original AlexNet's 1000 categories to match the number of concepts categories in this project.

Implementation:

```
nn.Linear(2048, num_concepts),
```

Feature Model:

Modification: Similarly, the output of the last fully connected layer was adapted from AlexNet's standard 1000 classes to the number of features categories required by this project.

Implementation:

```
nn.Linear(2048, num_concepts),
```

Test

In our research, we implemented a structured testing procedure to verify the architectural modifications of the AlexNet model, customized for specific multi-class and multi-label classification tasks. Utilizing Python's unittest framework, we developed a test named TestAlexNetArchitecture to systematically assess and validate the modifications made to the model.

Key Components of the Test:

Model Initialization: The AlexNet model was instantiated with parameters to support 473 concept classes and 2725 feature classes.

Verification of Output Layers: The test confirmed the correct configuration of the output layers—ensuring they matched the predefined numbers of outputs for concepts and features (For separate concept model or feature model, it would be just concept or feature).

Architecture Inspection: The entire structure of the model was printed and inspected to ensure all components were correctly implemented according to the design specifications.

Outcome:

Unified Model:

```

AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 48, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(48, 128, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(128, 192, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(192, 192, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(192, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=4608, out_features=2048, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
  )
  (classifier_concepts): Linear(in_features=2048, out_features=473, bias=True)
  (classifier_features): Linear(in_features=2048, out_features=2725, bias=True)
)

```

[Figure 11: Unified Model Architecture Test]

Concept Model:

```

AlexNet(
  (concepts): Sequential(
    (0): Conv2d(3, 48, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(48, 128, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(128, 192, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(192, 192, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(192, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=4608, out_features=2048, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=2048, out_features=473, bias=True)
  )
)

```

[Figure 12: Concept Model Architecture Test]

Feature Model:

```

AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 48, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(48, 128, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(128, 192, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(192, 192, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(192, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=4608, out_features=2048, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=2048, out_features=2725, bias=True)
  )
)

```

[Figure 13: Feature Model Architecture Test]

The test successfully verified that the model was correctly assembled and functioned as intended, ensuring its suitability for the intended image recognition tasks. This validation process underscores the model's readiness for deployment in research applications, highlighting the effectiveness of our architectural modifications.

5.2 Functionality Testing: Custom Datasets

Given the storage of feature information in the *updated_feature_matrix.csv* file, I have developed and tested a custom data class to extract corresponding features for concepts. This class is derived from the Dataset abstract class and includes essential methods for retrieving sample items from the dataset and representing the total number of samples.

Example Code for Function Invocation:

```

# Initialize datasets
full_dataset = CustomImageDataset(
    img_dir='.././.././things_data/filtered_object_images',
    feature_vectors=feature_vectors, # This line is omitted in the concept model
    namemap=namemap,
    transform=data_transform["train"]
)

```

Once the dataset is initialized, visualization of image samples and their corresponding feature labels can be carried out.

Code for Providing a Random Index and Visualizing a Sample from the Feature Dataset:

```

# Function to visualize an image and its feature labels
def visualize_sample(image, feature_vector, feature_names):
    # Check if the image is a torch.Tensor and convert it for visualization

```

```

    if isinstance(image, torch.Tensor):
        # Convert from CxHxW to HxWxC for visualization
        image = image.permute(1, 2, 0).numpy() # Assuming image data is in the
range [0, 1]
    plt.imshow(image) # Display the image
    plt.title("Sample Image")
    plt.show()

    # Filter and display feature names where the feature vector is 1
    positive_features = [name for value, name in zip(feature_vector,
feature_names) if value == 1]
    print("Positive Features:", ", ".join(positive_features))

# Load feature names from the DataFrame (assuming it's globally accessible)
feature_names = feature_vectors.columns.tolist()

# Test __len__ method to print the total number of samples in the dataset
print("Total number of samples in the dataset:", len(dataset))

# Test __getitem__ to fetch and visualize a random sample
random_index = random.randint(0, len(dataset) - 1)
sample_image, sample_concept_label, sample_feature_vector = dataset[random_index]

# Since we are using ToTensor in transform, the image is already a tensor, we
directly pass it
visualize_sample(sample_image, sample_feature_vector.numpy(), feature_names)

# Print out the concept label index to verify it matches with the dataset
print("Concept label index for the sample:", sample_concept_label)

```

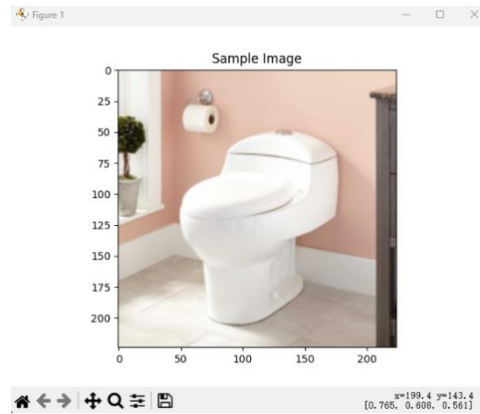
Outcome:

```

Total number of samples in the dataset: 6808
Positive Features: does_flush, has_a_bowl, has_a_cistern, has_a_flush, has_a_handle_handles, has_a_lid, has_a_seat_seats, has_water,
is_found_in_bathrooms, is_found_in_homes_houses, is_necessary_essential, is_sat_on, is_used_to_urinate_in, is_useful, is_where_you_poo,
is_white, made_of_ceramic_china_clay_porcelain, made_of_metal, made_of_plastic
Concept label index for the sample: 424

```

[Figure 14: Custom Datasets Test Result]



[Figure 15: Custom Datasets Test Sample Image]

5.3 Predict


During the prediction phase, for concept classification, the model's output logits are transformed into probabilities using the ***Softmax*** function. This conversion ensures that the probability of each category's prediction is clear and easy to interpret, making it more intuitive and user-friendly. Such an approach not only enhances the model's interpretability but also increases its practical applicability and accuracy in real-world scenarios.




To identify the top five most likely concepts, the model uses the ***torch.topk*** method. This method enables the model to output the probabilities of the most probable concepts with ease. This methodology is derived from the design of the original AlexNet, and it effectively highlights the model's ability to prioritize and display the most relevant classifications based on their likelihood.

In contrast, feature recognition predictions operate differently. Unlike concept classification, where an image is linked to a single definitive category, feature recognition can associate multiple features with a single image. Therefore, the output is not limited to the top few confidences. Instead, any feature with a confidence exceeding a specified threshold is considered a valid prediction. For this project, I set the threshold at ***0.3***, meaning any feature with confidence above this value is flagged as a correct prediction and included in the output.

Below are four instances illustrating the probability distributions for concept and feature classification predictions on an identical test image, as generated by the unified, concept, and feature models.

(When an object has too many features, the five with the highest probability would be selected for display.)

Image	File Name
	boot_01b.jpg

	cat_01b.jpg
	coffeemaker_01b.jpg
	truck_05s.jpg

Unified Model:

1.

Image	boot_01b.jpg				
Concepts	boots	skirt	cape	aubergine	telephone
Probabilities	0.3207	0.0348	0.0346	0.0285	0.0273
Features	is_useful	is_worn	is_big_large	is_long	is_heavy
Confidence	0.7167	0.4912	0.3315	0.3056	0.3045

2.

Image	cat_01b.jpg				
Concepts	doorknob	grenade	hawk	wine	rollerskate
Probabilities	0.1169	0.1036	0.0631	0.0433	0.0331
Features	is_an_animal	has_legs	is_big_large	is_dangerous	is_small
Confidence	0.9135	0.4007	0.3751	0.3350	0.3183

3.

Image	coffeemaker_01b.jpg				
Concepts	coffee_machine	bra	shield	blender	sofa
Probabilities	0.7711	0.0822	0.0425	0.0399	0.0314
Features	made_of_metal		made_of_plastic		
Confidence	0.4050		0.4048		

4.

Image	truck_05s.jpg				
Concepts	surfboard	bell	taxi	boat	ambulance
Probabilities	0.1832	0.0725	0.0536	0.0496	0.0290
Features	made_of_metal	is_an_animal	has_legs	does_eat	has_eyes
Confidence	0.6696	0.4996	0.4270	0.4172	0.4000

Concept Model:

1.

Image	boot_01b.jpg
-------	--------------

Concepts	boots	apron	skirt	tractor	jackets
Probabilities	0.2228	0.0428	0.0228	0.0172	0.0043

2.

Image	cat_01b.jpg				
Concepts	cat	doorknob	grenade	saxophone	owl
Probabilities	0.3752	0.0870	0.0636	0.0513	0.0435

3.

Image	coffeemaker_01b.jpg				
Concepts	coffee_machine	blender	kettle	orchid	bra
Probabilities	0.2604	0.0748	0.0427	0.0287	0.0282

4.

Image	truck_05s.jpg				
Concepts	truck	anchor	jeep	ambulance	chainsaw
Probabilities	0.2487	0.1089	0.1027	0.0482	0.0354

Feature Model:

1.

Image	boot_01b.jpg				
Features	is_long	does_protect	is_warm	made_of_leather	is_worn
Confidence	0.4873	0.4817	0.4509	0.4436	0.4378

2.

Image	cat_05s.jpg				
Features	is_an_animal	has_a_tail	does_eat	has_legs	has_fur_hair
Confidence	0.9886	0.9709	0.9422	0.9065	0.8852

3.

Image	coffeemaker_01b.jpg				
Features	made_of_metal				
Confidence	0.3143				

4.

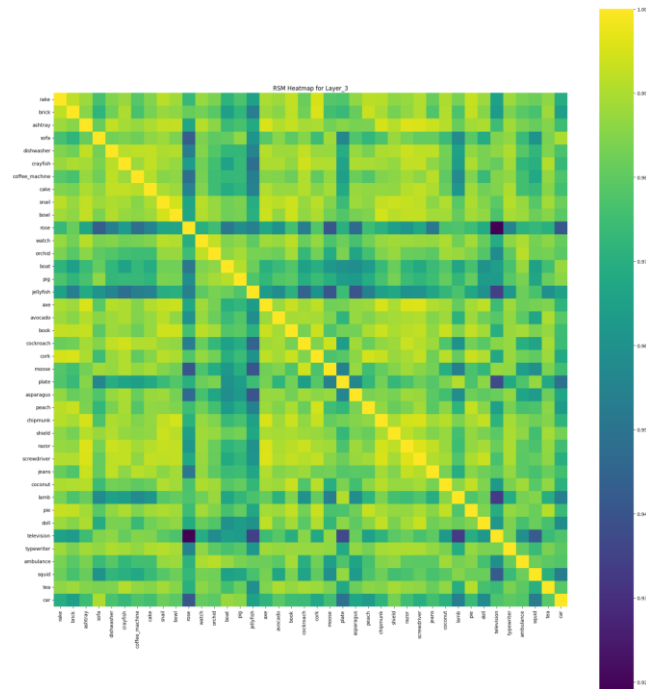
Image	truck_01b.jpg				
Features	has_wheels	made_of_metal	is_heavy	is_big_large	is_a_vehicle
Confidence	0.9816	0.8787	0.8729	0.8723	0.8576

From the results above, it is evident that all three models are capable of performing the corresponding classification predictions for the images, providing predictive probabilities/confidence for each potential category listed. To some extent, their predictions correspond with the actual labels.

5.4 RSM & Second Order RSM

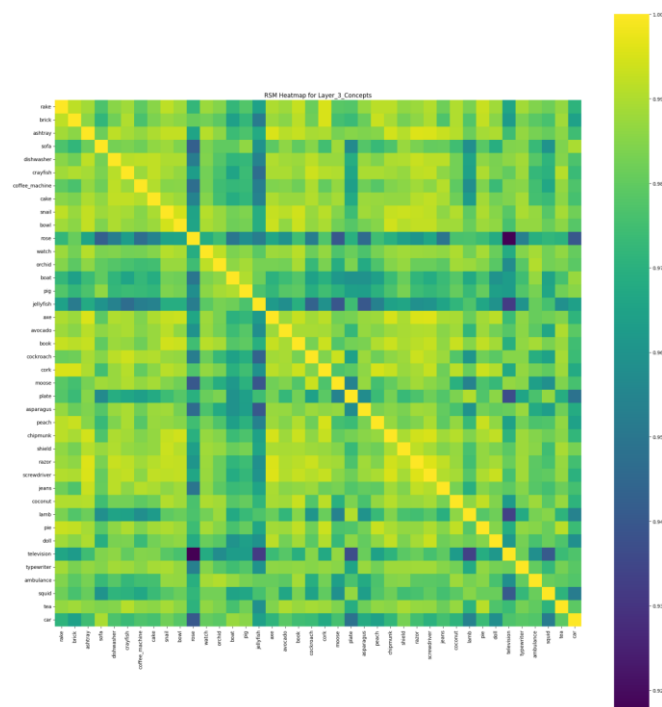
In Section 4.3.3, it is mentioned that a Representational Similarity Matrix (RSM) is generated for each layer of every model. Each RSM involves calculating and comparing the cosine similarities of activation patterns for a randomly selected set of N concepts (40 in this case).

Unified Model:



[Figure 16: Layer 3 RSM of Unified Model]

Concept Model:



[Figure 17: Layer 3 RSM of Concept Model]

Feature Model:

6.0 System Evaluation and Experimental Results

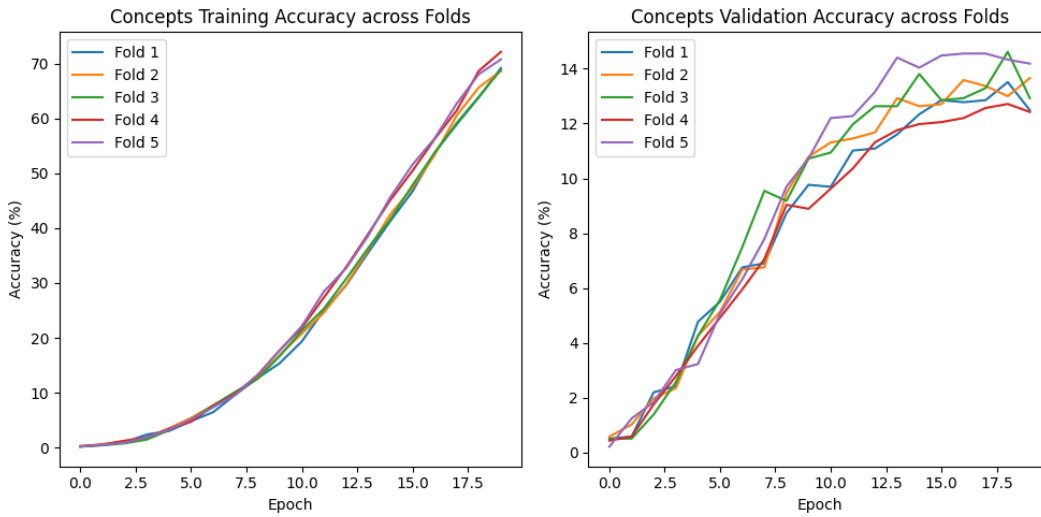
6.1 CNN's Performance in Object Concepts Classification

The test results indicate that the constructed CNN is capable of performing concept classification on images.

Since the total loss in the unified model is the sum of the losses from concept classification and feature classification, to facilitate the analysis of concept classification issues, this discussion will focus on the individual concept classification loss from the unified model, as well as the loss from the standalone concept model.

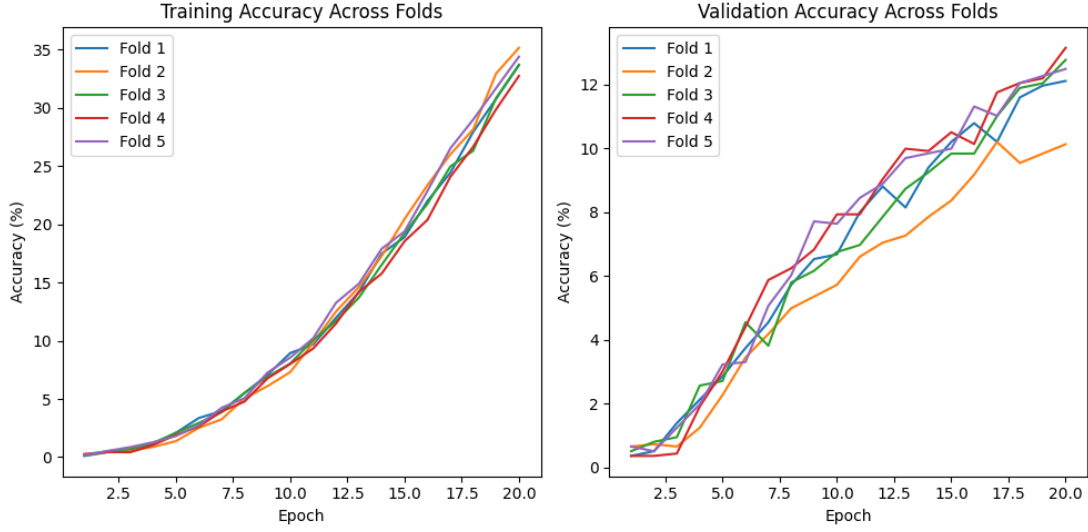
Below are displayed the loss curves for concept classification from the unified model and the concept model, as well as the accuracy curves for both models.

Unified Model:



[Figure 19: Concepts Accuracy Curves of Unified Model]

Concept Model:



[Figure 20: Accuracy Curves of Concept Model]

As can be seen from the test accuracy graph, the accuracy of the UNIFIED model is generally higher than that of the CONCEPT model. the accuracy of the CONCEPT model ranges from a high of around 12% to a low of only around 9%. The highest accuracy rate of the unified model can reach more than 14%, and the lowest accuracy rate is generally more than 12%. This suggests that the unified model can combine the advantages of concept prediction and feature prediction through multi-task learning to improve the model performance. Through multi-task learning, the UNIFIED model is able to learn both the conceptual features and the underlying features in the image, thus achieving better performance on the conceptual classification task. This type of learning helps the model to capture richer information and improve the understanding of image content. In addition, multi-task learning also promotes sharing and regularization of model parameters and reduces the risk of overfitting, resulting in better generalization ability on concept classification tasks.

It can also be observed from the test accuracy graph that the accuracy of the UNIFIED model shows a gradual increase with the training process and stabilizes after a certain number of iterations. This indicates that the model gradually learns the features of the image during the training process and is able to achieve better performance on the concept classification task. While the concept model shows some improvement in accuracy at the beginning of training, the improvement is smaller at the later stage, indicating that single-task learning has limited ability in capturing image features.

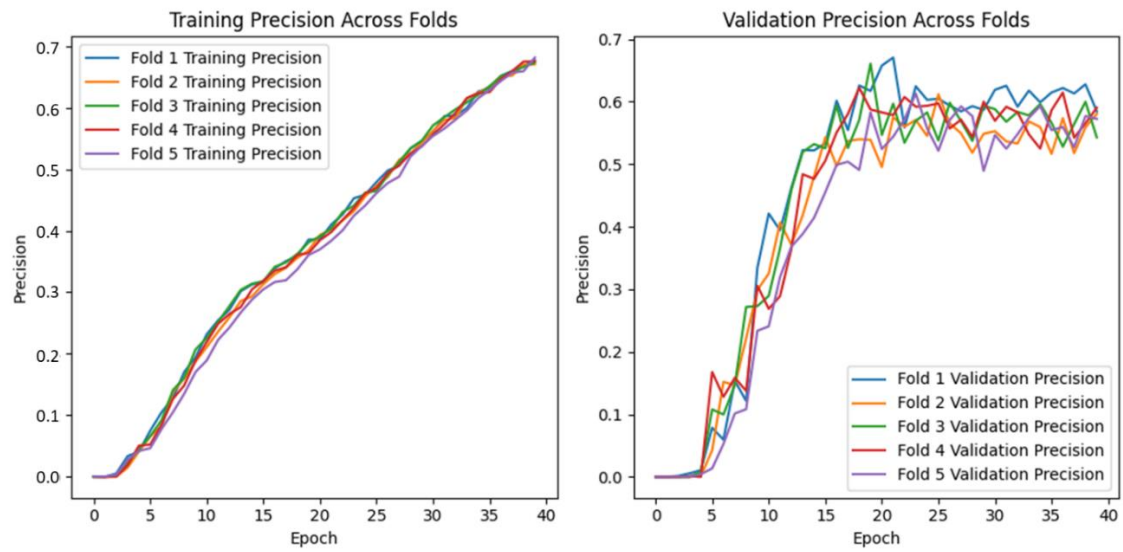
6.2 CNN's Performance in Object Features Classification

The test results indicate that the constructed CNN effectively recognizes certain semantic data in images.

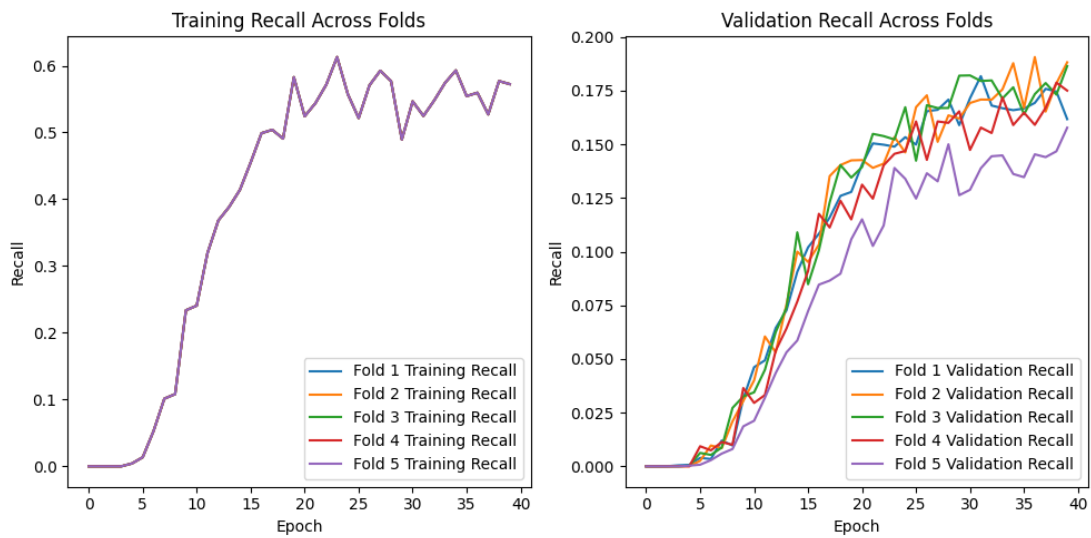
Since the total loss in the unified model comprises the sum of the losses from both concept classification and feature classification, to facilitate the analysis of the feature classification issues, this discussion will focus on the individual feature classification loss from the unified model as well as the loss from the standalone feature model.

Below are displayed the loss curves for feature classification from the unified model and the feature model, along with the Precision and Recall curves for both models.

Unified Model:

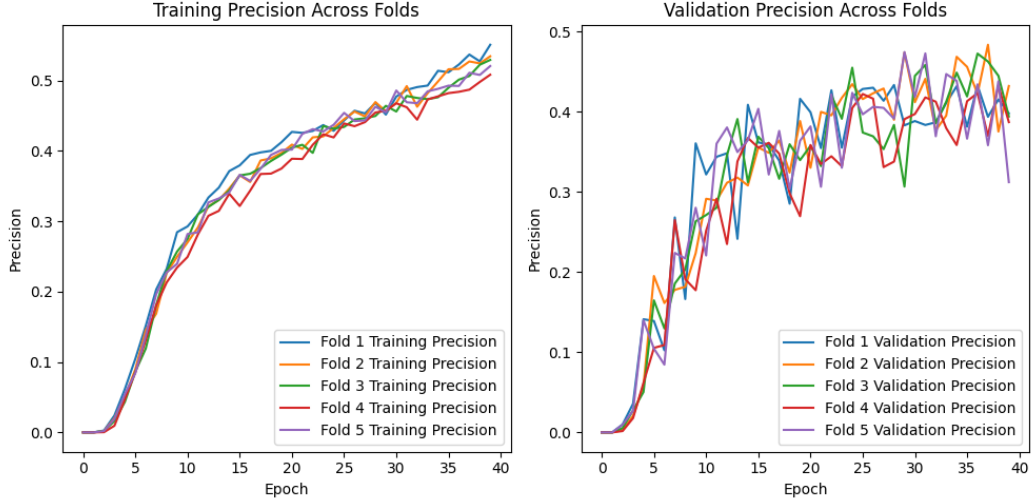


[Figure 21: Feature Precision Curves of Unified Model]

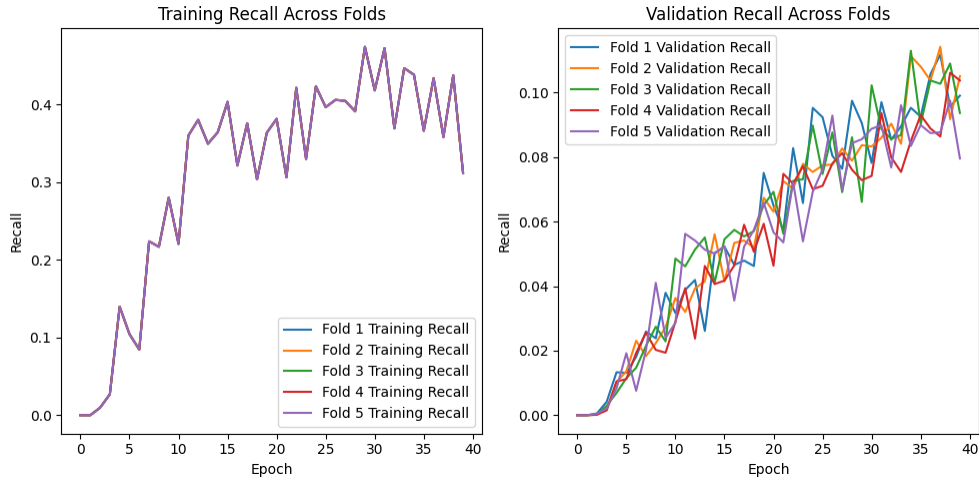


[Figure 22: Feature Recall Curves of Unified Model]

Feature Model:



[Figure 23: Precision Curves of Feature Model]



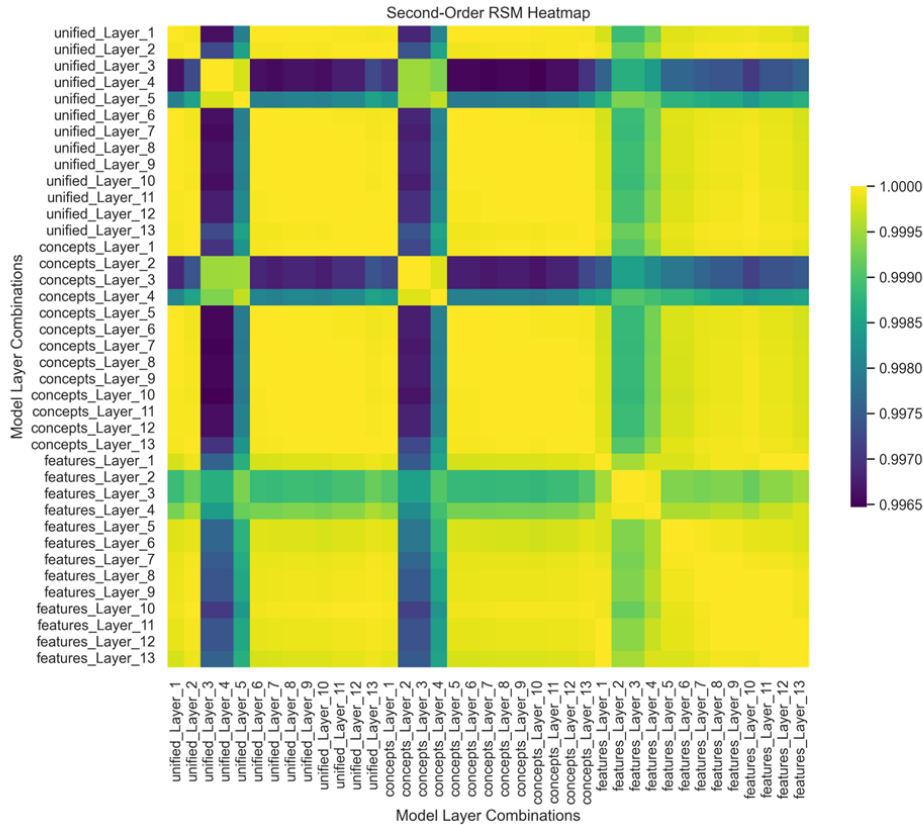
[Figure 24: Recall Curves of Feature Model]

From the experimental results, it can be found that the Unified model is significantly higher than the separate feature model in both precision and recall. For example, in the precision statistics of the test set, the precision of the unified model eventually stabilizes around 0.6, while the separate feature model stabilizes around 0.4. In addition, the accuracy curve of the FEATURE model is more unstable, with a larger up and down range, whereas the accuracy curve of the UNIFIED model is more stable, with a smaller range of fluctuation. In the recall rate statistics graph of the test set, it can be found that the recall rate of the UNIFIED model can reach about 0.175, while the recall rates of the FEATURE model are all below 0.1, which indicates that the UNIFIED model is able to better capture the complex relationships in the data and predict the labels of the samples more accurately. These results indicate that the Unified model is able to improve the performance and prediction accuracy of the model by integrating the advantages of FEATURE prediction and CONCEPT prediction.

In multi-task learning, the Unified model is able to learn not only the relationships between different features, but also the associations between concepts, which enables it to utilize this information more comprehensively when making predictions. By sharing the representation, the Unified model is also able to reduce the risk of model overfitting and improve the generalization ability of the model.

In addition, it can be observed from the experimental results that the performance of the Unified model is relatively stable across datasets, while the separate feature model performs poorly on some datasets. This further illustrates that the Unified model improves the robustness of the model through multi-task learning. In summary, the Unified model effectively combines the advantages of FEATURE prediction and CONCEPT prediction through multi-task learning, which improves the precision and recall of the model, and thus improves the performance and prediction accuracy of the model. These results provide a valuable reference for the application of Unified model in multi-label classification tasks.

6.3 RSA: Difference in Models



[Figure 25: Second-Order RSM]

After a thorough analysis of the data presented by the Second-Order Representational Similarity Matrix (RSM), this study offers the following detailed observations and conclusions:

Diversity at the Primary Level: The heatmap reveals relatively low similarity between the three models at the primary levels (layers 2 to 5), which include two max pooling layers (MAXPOOL1 and MAXPOOL2) and two convolutional layers (CONV2 and CONV3). These layers primarily function to extract low-level features from the input images, such as edges, corners, and textures. The low similarity between the layers of machine learning models. It suggests that each model may have developed unique feature extraction strategies based on specific factors, such as their training datasets, weight initialization, or activation functions.

Consistency at Higher Levels: However, starting from the sixth layer, the similarity significantly increases and continues through to the final layer. This suggests that at the higher levels of the models, feature

representations tend to converge, reflecting commonalities among different models in learning more abstract concepts and classification decisions. These layers likely represent advanced features, such as parts or wholes of objects, which may be represented in a similar manner across different models.

Optimization of Computational Efficiency: This study also suggests that certain layers may contain redundant information with limited contributions to the final decision-making. By removing or consolidating these highly similar layers, the size and complexity of the models can be reduced without significantly sacrificing accuracy. This presents an opportunity to optimize the computational efficiency of the models, particularly in resource-constrained application scenarios.

Dimensionality Reduction and Model Compression: One important aspect of research is to increase efficiency through dimensionality reduction. This involves analyzing features or layers that have little impact on the model's overall performance and subsequently removing them. The idea is to streamline the model's architecture, making it easier to process and potentially better at generalizing across diverse datasets.

By removing redundant layers, which can be characterized by similar or identical activation patterns, the risk of overfitting is reduced. The model becomes less complex and more focused on the most important features of the data. This pruning process reduces the computational load during both training and inference phases and also decreases the model's memory footprint, making it easier to deploy in resource-constrained environments such as mobile devices or embedded systems.

This process of removing unnecessary components can also be seen as a form of model compression. Compressed models retain the essential information while shedding the extraneous weight that could otherwise dilute the model's decision-making processes. It's like refining raw ore into precious metal - removing impurities to leave behind a purer, more valuable substance.

Advanced techniques, such as knowledge distillation, can be used to further enhance this process.

Implementing such dimensionality reduction not only yields a leaner model but may also enhance the interpretability of the neural network. By operating with fewer, more significant layers, the path from input to output becomes less convoluted, allowing researchers and practitioners to more readily trace and understand the model's reasoning. Ultimately, the quest for simplification through dimensionality reduction and model compression dovetails with the overarching goal of creating more efficient, transparent, and accessible deep learning models.

Next Steps: Before translating these observations into practical model optimization measures, it is recommended to conduct a series of experiments to quantitatively analyze the contribution of each layer and test how the model's performance on various tasks changes after the removal of these layers. Furthermore, exploring how techniques such as knowledge distillation can be used to transfer the capabilities of large models to more streamlined architectures may be a valuable direction for future research.

In summary, the RSM heatmap provides compelling visual evidence that computational efficiency and speed of models, particularly in multi-layer neural networks and deep learning models, can be improved through network pruning and dimension reduction. These findings not only provide direction for future model design

but also open possibilities for deploying these models on devices with limited computational resources.

References

Gitlab Repository:

https://gitlab2.eecs.qub.ac.uk/40381868/2023_csc3002_hailin_weng_p25_objectparts.git

- [1] B. J. Devereux, L. K. Tyler, G. Geertzen, et al., "The Centre for Speech, Language and the Brain (CSLB) concept property norms," *Behav Res*, vol. 46, pp. 1119–1127, Dec. 2014. [Online]. Available: <https://doi.org/10.3758/s13428-013-0420-4>. [Accessed: 2023-10-14].
- [2] L. Alzubaidi, J. Zhang, A. Humaidi, A. J. Al-Dujaili, Y. Duan, O. Al-Shamma, L. Santamaría, M. A. Fadel, M. Al-Amidie, and L. Farhan, "Review of deep learning: concepts, CNN architectures, challenges, applications, future directions," *Journal of Big Data*, vol. 8, no. 53, Mar. 2021. [Online]. Available: <https://doi.org/10.1186/s40537-021-00444-8>. [2023-10-25].
- [3] M. N. Hebart, A. H. Dickter, A. Kidder, W. Y. Kwok, A. Corriveau, C. Van Wicklin, and C. I. Baker, "THINGS: A database of 1,854 object concepts and more than 26,000 naturalistic object images," *PLOS ONE*, vol. 14, no. 10, e0223792, Oct. 15, 2019. [Online]. Available: <https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0223792>. [Accessed: 2023-12-13].
- [4] zhulie1025, "cs231n: Notes & Assignments for Stanford CS231n 2020," GitHub repository, 2020. [Online]. Available: <https://github.com/zhulie1025/cs231n>. [Accessed: 2023-10-15].
- [5] WZMIAOMIAO, "deep-learning-for-image-processing: deep learning for image processing including classification and object-detection etc.," GitHub repository, Last modified last month, 2020. [Online]. Available: <https://github.com/WZMIAOMIAO/deep-learning-for-image-processing>. [Accessed: 2023-11-03].
- [6] "Convolution," in *Understanding Virtual Reality (Second Edition)*, 2018. [Online]. Available: <https://www.sciencedirect.com/topics/mathematics/convolution>. [Accessed: 2024-02-08].
- [7] Teco KIDS, "Fully-Connected Layer with dynamic input shape," Medium, Jul. 31, 2019. [Online]. Available: <https://medium.com/@tecokids/fully-connected-layer-with-dynamic-input-shape-70c869ae71af>. [Accessed: 2024-02-09].
- [8] PyTorch Team, "AlexNet," PyTorch Documentation. [Online]. Available: https://pytorch.org/hub/pytorch_vision_alexnet/. [Accessed: 2023-11-01].
- [9] PyTorch, "CrossEntropyLoss," PyTorch Documentation. [Online]. Available: <https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html>. [Accessed: 2023-11-13].
- [10] PyTorch, "BCELoss," PyTorch Documentation. [Online]. Available: <https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.html>. [Accessed: 2023-11-13].
- [11] PyTorch, "BCEWithLogitsLoss," PyTorch Documentation. [Online]. Available: <https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html>. [Accessed: 2023-11-13].
- [12] PyTorch, "MSELoss," PyTorch Documentation. [Online]. Available: <https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html>. [Accessed: 2023-11-13].
- [13] L. Chang, "Representational Similarity Analysis," DartBrains. [Online]. Available: <https://dartbrains.org/content/RSA.html>. [Accessed: 2024-03-06].
- [14] B. J. Devereux, A. Clarke, A. Marouchos, and L. K. Tyler, "Representational Similarity Analysis Reveals Commonalities and Differences in the Semantic Processing of Words and Objects," *Journal of Neuroscience*, vol. 33, no. 48, pp. 18906-18916, Nov. 2013, doi: 10.1523/JNEUROSCI.3809-13.2013. [Accessed: 2024-03-08].