Image classification using the Stanford Cars dataset containing images for 196 models of cars

CNN Models on Stanford Cars dataset

Amit Agarwal, Salim Touati, Selvalakshmi Ramagopalan, Srimallika Potluri

Contents

Participants

- Amit Agarwal (technical lead, data scientist)
- Salim Touati (data engineering)
- Selvalakshmi Ramagopalan (data scientist)
- Srimallika Potluri (data scientist)

The code is hosted on GitHub at **https://github.com/agarwalamit081/deeplearning-dsti**

Dataset

The Stanford Cars dataset, developed by Stanford University AI Lab, contains 16185 images from the rear of each car of 196 different models of cars (classes). The images have been split into 8144 training images and 8041 testing images where each class has been split roughly in a 50-50 split.

The dataset is organized into train and test folders, each containing subfolders for the 196 classes (models of cars) under which we find related images of the cars.

The dataset also contains names.csv containing the names of the 196 classes, anno_train.csv and anno_test.csv which contain the boxing information for each of the images.

The shape of the image is 224 x 244x3 where 224 is the height and width of the image and 3 represents the number of colour channels for red, blue and green.

The dataset is available on Kaggle at https://www.kaggle.com/datasets/jessicali9530/stanford-cars-dataset.

Code

Python 3.9.15 was used along with TensorFlow, OpenCV, skopt, albumentations and other libraries as mentioned below.

The code was run on a GPU enabled pc with CUDA libraries to make use of the NVIDIA GPU with TensorFlow and albumentations libraries compatible with GPUs. Various Convolutional Neural Network (CNN) models were used for the classification problem.

Python version	3.9.15
IPython version	8.15.0
NumPy	1.26.4
Pandas	2.2.3
Matplotlib	3.9.1
logging	0.5.1.2
Scikit-Learn	1.5.2
TensorFlow	2.6.0
CV2	4.5.1
Skopt	0.10.2
Albumentations	1.3.1
Tqdm	4.66.5
CUDNN	8.2.1.32
CudaToolkit	11.3.1

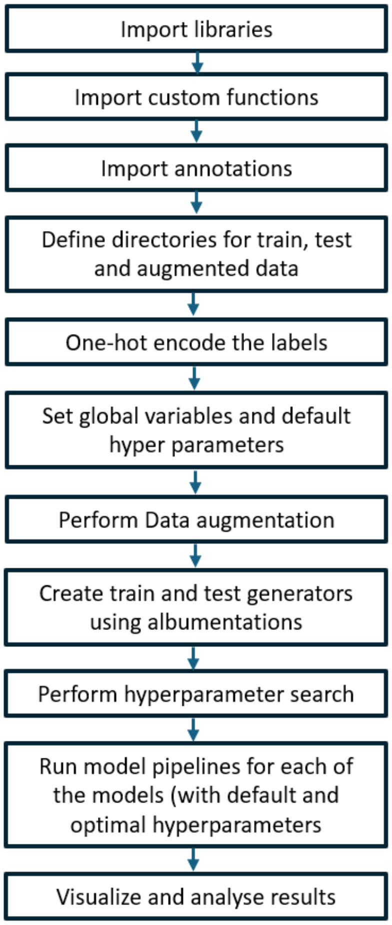
The following objectives were kept in mind:

- A functional coding approach to enhance code reusage
- Usage of GPU supported libraries to make computations faster
- Resiliency and robustness of the code to make it fault tolerant.

- Since there were computations which ran for quite long, care was taken to pickle the intermediate results whenever possible to resume from where it was left off due to something breaking in the middle or an application failure.
- Profilers used all along to gauge the most problematic aspects of the workflow and logging was enabled as well to help capture errors and the causes behind failed code chunks.
- Computation times were also calculated all along.
- Avoided loops that could lead to overflows or make the program run out of memory.

Workflow

The following are the steps involved in the implementation.


Import libraries
Import custom functions
Import annotations
Define directories for train, test and augmented data
One-hot encode the labels
Set global variables and default hyper parameters
Perform Data augmentation
Create train and test generators using albumentations
Perform hyperparameter search
Run model pipelines for each of the models (with default and optimal hyperparameters
Visualize and analyse results

1. Loading the libraries, user defined functions.
2. Load the csv for the names of the classes and the annotations for train and test data.
3. Define path to train and test dataset and create an additional folder for augmented data
4. Enable logging
5. Enable Mixed Precision Training and XLA Compilation
6. Create train and test labels, convert them into a NumPy array for reshaping and one-hot encode them
7. Assign global variables for batch size, epochs, learning rate, dropout etc with some default values.
8. Data Augmentation using albumentations

We augment the data to create newer versions of the original image from the boxing information available. This helps reduce over-fitting by increasing the variations in the training dataset by using slightly different versions of the same images. The different transformations applied to the images (flip, brightness/contrast adjustment, rotation, scaling, blurring, colour variation, and distortion) makes the model more resilient to real-world variations in images.

It helps the model learn the essential features of the objects, rather than on specific details that may vary across samples, leading to better generalization on unseen data.

The pitfalls are that the increased training data takes longer time to train the model and transformations like blurring or rotation can reduce the image quality, making it harder for the model to learn certain details.

It must be noted that the boxing is not quite accurate which impact the results. Recreating the boxing manually for 16185 images manually is not feasible and an automatic boxing technique applied did not yield good results as it clipped off parts of the car in the images.

Nevertheless, the augmentation pipeline is well-balanced, as each transformation has a probability less than 1, meaning that images will be randomly transformed in different ways, preserving enough of the original data to maintain context.

9. Train and test data generators

We create data generators, train_generator, and test_generator for train and test data sets which feeds the images to the model in batches to streamline the training/testing process. It takes input_shape, batch_size, train_dir, and test_dir to configure the generators.

10. Hyperparameter search (only on the training set)

Higher learning rates can lead to faster convergence but risk overshooting optimal points, while lower learning rates can improve accuracy but slow down training.

Weight decay and dropout are regularization techniques help prevent overfitting but can make the model slower to converge if overused.

We calculate the values of the hyper-parameters such as the learning rate, weight decay, dropout, beta1, beta2, decay, momentum and rho values. Initially ran with different kinds of optimizers such as SGD, Adam, RMSProp, AdaGrad, but later restricted to just Adam as it performed better than the others. The results were pickled and saved for later use.

Cross-Validation during hyper-parameter search was avoided as training and validating the model k times (for k folds) is computationally expensive. We instead perform hyperparameter-search on the entire train dataset although it introduces the risk of overfitting and lesser reliable performance metrics.

Other techniques used to optimize the computations includes:

- Early Stopping to enable a more efficient sampling
- Optuna to prune suboptimal trials
- Both Bayesian Optimization and Scikit-Optimize were tried but Scikit-Optimize was preferred in the end.
- Learning rate schedulers (e.g., ReduceLROnPlateau) as fixed learning rates may not work well across all epochs

11. Create the model with the optimal hyper-parameters calculated in the above step.

We analysed the following Convolutional Neural Network models:

I. model1
II. AlexNet
III. VGG16
IV. ResNet50
V. DenseNet
VI. ImageNet
VII. EfficientNet
VIII. InceptionV3

For most of these models, separate functions were created to fine-tune these models further.

We used the same hyper-parameters for all the models due to computational constraints, but it is advised to use different hyperparameters for each of the models due to the following reasons

- VGG16 has a simpler architecture compared to ResNet50, which uses residual connections and may require different learning rates, batch sizes, and optimizers to converge efficiently.

- VGG16 and ResNet50 might respond differently to optimizers like SGD, Adam, or RMSprop due to the difference in layer depth and gradient behaviour (ResNet has skip connections that alleviate vanishing gradients).
- ResNet models often benefit from smaller learning rates and optimizers like SGD with momentum, while VGG16 can handle larger learning rates with Adam in some cases. A learning rate that works well for VGG16 might be too large for ResNet50, leading to instability in training.
- Weight decay (regularization) might need to be higher for models with more parameters like ResNet50 to prevent overfitting. Smaller models like VGG16 might not require as much regularization.
- ResNet50, being more complex, might need a smaller batch size to fit in memory,

12. Compile the model
13. Train the model on the training set and test the model on test data
14. Using k-folds during training (not used)
15. Plot and analyse the metrics for the models: accuracy, loss, precision, recall, and AUC on the train and test data.

CNN layers used

The following CNN layers have been used in the various models presented.

- Conv2D Layer (Convolutional Layer)
- Dropout Layer
- Spatial Dropout Layer
- MaxPooling Layer
- Batch Normalization Layer
- Flatten Layer
- Dense Layer (Fully Connected Layer)
- GlobalAveragePooling2D Layer
- Add Layer
- ZeroPadding2D Layer
- Rescaling Layer
- DepthwiseConv2D Layer
- Reshape Layer
- Custom Layer
- SE Block (Squeeze-and-Excitation Block)
- Attention Block Layer
- Activation (ReLU, SoftMax)

Data Augmentation Step

The Compose function in albumentations combines a series of augmentations into a single transformation pipeline, which will be applied sequentially to each image.

Each transformation has an associated probability p which determines the likelihood of that transformation being applied.

A.HorizontalFlip(p=0.5)

Randomly flips the image horizontally with a 50% probability.

Useful for creating mirrored versions of the images, which helps the model learn that objects are the same whether facing left or right.

A.RandomBrightnessContrast(p=0.2)

Randomly adjusts the brightness and contrast of the image with a 20% probability.

Helps the model generalize across different lighting conditions by making the image brighter or darker, and adjusting contrast between colors.

A.Rotate(limit=12, p=0.5)

Rotates the image by a random angle within a specified range (-12 to +12 degrees) with a 50% probability.

Helps the model be less sensitive to the exact orientation of objects within images, improving robustness to slight rotations.

A.ShiftScaleRotate(shift_limit=0.05, scale_limit=0.05, rotate_limit=4, p=0.2)

Combines shifting, scaling, and rotating transformations:

shift_limit=0.05: Shifts the image horizontally and vertically by up to 5% of its dimensions.

scale_limit=0.05: Scales the image by a random factor within 5% of its original size.

rotate_limit=4: Rotates the image by a random angle within -4 to +4 degrees.

Applied with a 20% probability.

This composite transformation is useful for handling slight translations, rescaling, and rotations, which makes the model more resilient to minor changes in position and size of objects.

A.GaussianBlur(blur_limit=(3, 5), p=0.6)

Applies a Gaussian blur to the image, with the blur kernel size randomly selected between 3 and 5 pixels.

Applied with a 60% probability.

Helps in smoothing out image noise and adding a level of blur, which can simulate out-of-focus images and improve robustness to subtle texture variations.

A.HueSaturationValue(hue_shift_limit=10, sat_shift_limit=20, val_shift_limit=10, p=0.3)

Adjusts the hue, saturation, and value (brightness) of the image:

hue_shift_limit=10: Shifts the hue channel within a range of -10 to +10.

sat_shift_limit=20: Shifts the saturation within -20 to +20.

val_shift_limit=10: Shifts the value (brightness) within -10 to +10.

Applied with a 30% probability.

This augmentation simulates colour variations and helps the model become robust to variations in colour intensity, which may occur in different lighting conditions.

A.GridDistortion(num_steps=5, distort_limit=0.2, p=0.2)

Applies a grid distortion effect by deforming a grid over the image:

num_steps=5: Specifies the number of grid cells in each direction.

distort_limit=0.2: Controls the degree of distortion, allowing a maximum shift of 20% in each cell.

Applied with a 20% probability.

This augmentation introduces random local distortions, which can help the model handle slight geometric variations, like in images of textures or organic shapes.

Hyper-parameter Search

The following hyper parameters have been used: learning_rate, weight_decay, dropout, beta_1 and beta_2, decay, momentum, rho, and optimizer_type.

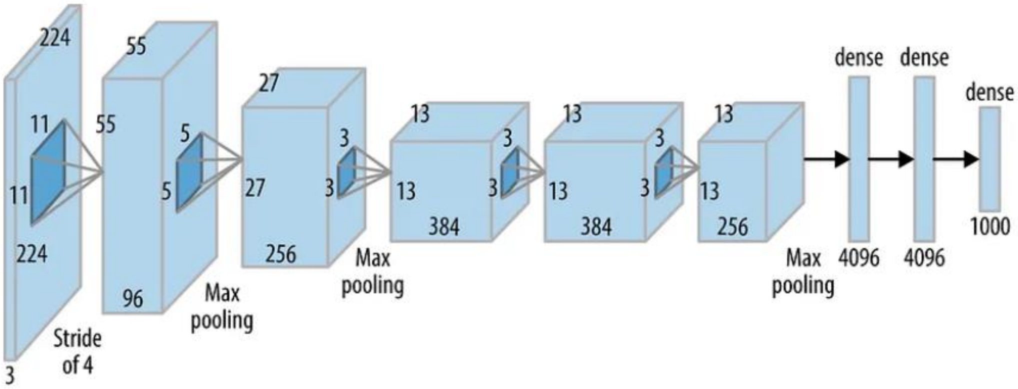
Overview of the Models

The models used in the study are briefly described below.

AlexNet

A convolutional neural network (CNN) architecture with 29 layers and 47,605,796 parameters.

It contains layers for input (1), Conv2D, Spatial Dropout, MaxPooling, Batch Normalization, Attention Block, SE Block, Custom Layer, flatten, and dense layers.



The function, finetune_alexnet_model, sets up the structure of AlexNet with additional attention and SE blocks and takes the same parameters as input.

The use of SpatialDropout2D improves regularization by randomly dropping feature maps to prevent overfitting.

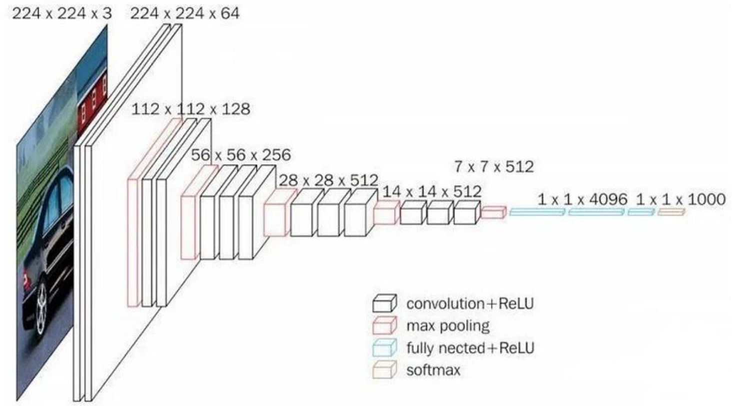
The model includes two dense layers (Fully Connected Layers) with ReLU activation and a final output layer with SoftMax for classification.

The dropout rate is dynamically controlled by the dropout hyperparameter.

VGG16

A type of Convolutional Neural Network (CNN) with 31 layers and 27,859,652 parameters. The 16 in VGG16 refers to 16 layers that have weights.

It contains layers for input (1), Conv2D (13), Spatial Dropout (1), MaxPooling2D (5), Attention Block (1), SE Block (1), Custom (2), flatten (1), dense (2) and dropout(1) layers.



We use a pre-trained VGG16 model as a feature extractor and adds custom layers, including attention and SE (Squeeze-and-Excitation) blocks.

The base VGG16 model is loaded with weights pre-trained on ImageNet (weights='imagenet').

- include_top=False means the top layers (fully connected layers) are excluded, allowing for custom layers to be added.
- transfer_layer: The layer block5_pool (the final pooling layer) is chosen as the transition point where the custom layers will be attached.

The final output layer is a dense layer with SoftMax activation to classify inputs into num_classes.

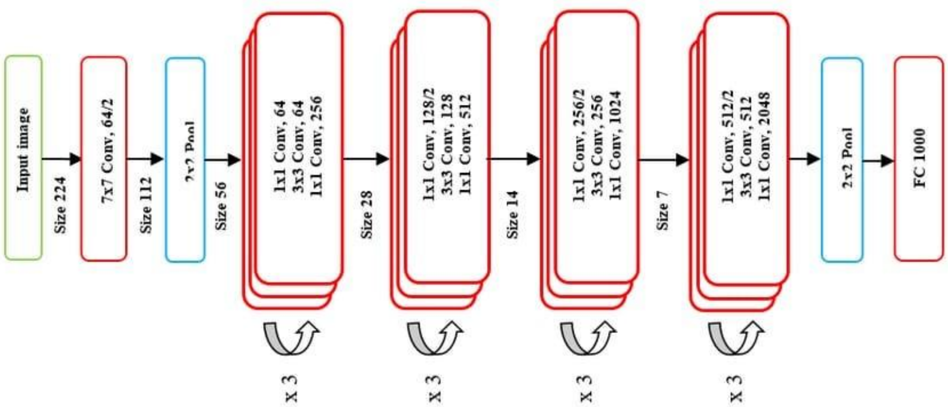
This setup allows for effective transfer learning by leveraging pre-trained weights and customizing the model with attention and SE mechanisms for improved performance in image classification tasks.

ResNet50 (Residual Network)

A type of deep Convolutional Neural Network (CNN) with 177 layers and 27,892,036 parameters.

It contains layers for input (1), Conv2D (53), Batch normalization (53), MaxPooling2D (1), Add (16), Activation (16), ZeroPadding2D (2), Global Average Pooling (1), Attention Block (1), SE Block (1), Custom (2), dense (2) and dropout(1) layers.

It uses residual connections, which allow the network to learn a set of residual functions that map the input to the desired output by eliminating vanishing gradients, a problem where the gradients of the parameters in the deeper layers become very small. Skip connections allow the information to flow directly from the input to the output of the network, bypassing one or more layers.



We create and fine-tunes a ResNet50 model using transfer learning with added layers, attention, and Squeeze-and-Excitation (SE) blocks for enhanced image classification.

Pre-trained weights on ImageNet (weights='imagenet') help the model retain feature extraction ability.

Transfer Layer: conv5_block3_out is chosen as the output layer of the feature extraction portion of ResNet50.

Custom layers have been added to the base model and they include Spatial Dropout, Attention Block, SE Block and Global Average Pooling layers.

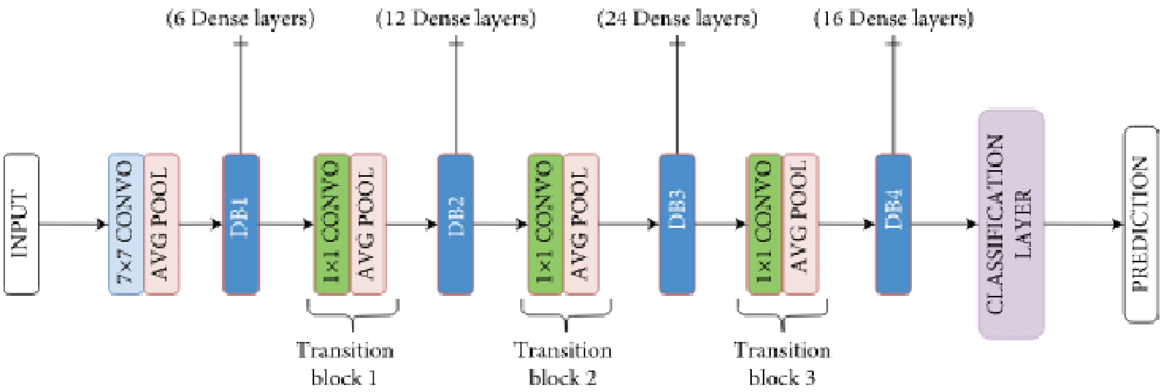
Fully Connected Layers: Adds dynamically determined dense layers based on fc_layers from hyperparams, applying Dropout after each to prevent overfitting.

The output layer adds a dense layer with SoftMax activation, mapping the final output to num_classes.

DenseNet121

A type of Convolutional Neural Network (CNN) with typically 121 layers but 354 layers have been used (includes additional layers for attention, SE and custom layers) and 33,224,132 parameters of which 33,140,484 are trainable.

It contains layers for Input (1), Conv2D (121), BatchNormalization (124), Activation (124), ZeroPadding2D (2), MaxPooling2D (1), AveragePooling2D (2), Concatenate (48), Flatten (1), Dense (2), Dropout (1), Spatial Dropout2D (1), Attention Block (1), and SE Block (1). It addresses challenges such as feature reuse, vanishing gradients, and parameter efficiency.



It loads the DenseNet121 model without the top (classification) layers using include_top=False.

Weights are pre-trained on ImageNet to give the model a strong feature extraction base.

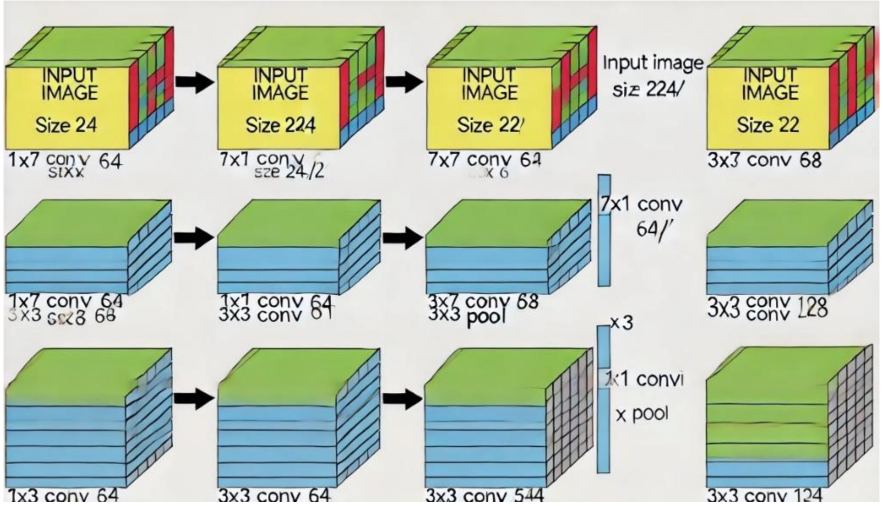
Custom layers have been added to the base model and they include Spatial Dropout, Attention Block, SE Block and Flattening layers.

The output layer adds a final Dense layer with SoftMax activation for multi-class classification, matching the number of classes defined by num_classes.

ImageNet

A type of Convolutional Neural Network (CNN) with 230 layers and 78,223,684 parameters of which 78,170,564 are trainable.

It contains layers for Input (1), ZeroPadding2D (2), Conv2D (Convolutional Layers)(53), BatchNormalization (53), Activation (ReLU)(53), MaxPooling2D (1), Add (Skip Connections)(16), Flatten (1), Dense (Fully Connected)(2), Dropout (1), Spatial Dropout2D (1), Attention Block (1), SE Block (Squeeze-and-Excitation)(1), and Custom (2)



This ImageNet model is derived from a a custom ResNet50 model using transfer learning and fine-tunes it for a specific classification task.

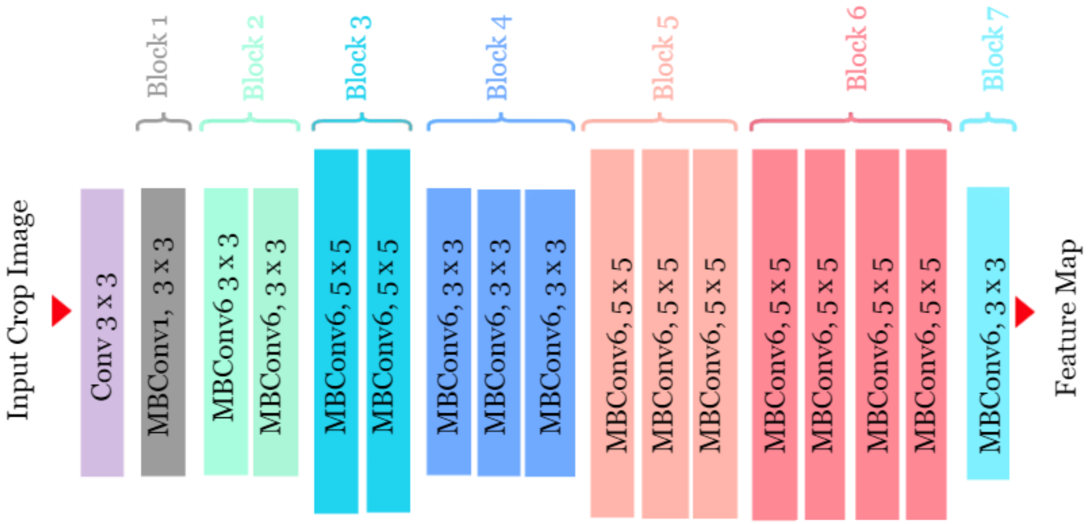
Each fully connected layer is followed by a Dropout layer, using the specified dropout rate to mitigate overfitting.

The output layer adds a final dense layer with SoftMax activation to produce class probabilities, with the number of units equal to num_classes.

EfficientNetB0

A type of Convolutional Neural Network (CNN) with 237 layers and 37,497,671 parameters of which 37,455,648 are trainable.

It contains layers for Input (1), Rescaling (1), Normalization (1), ZeroPadding2D (5), Conv2D (Standard Convolutional) (24), DepthwiseConv2D (Depthwise Convolutional) (12), BatchNormalization (44), Activation (25), Reshape (6), GlobalAveragePooling (Squeeze in SE blocks) (6), Conv2D (SE block reduce and expand) (12), Multiply (SE block excite) (6), Dropout (6), Add (Skip connections) (6), Flatten (1), Dense (Fully connected) (2), SpatialDropout2D (1), AttentionBlock (1), SEBlock (1), and Custom (2).



The EfficientNetB0 model, pre-trained on ImageNet, is loaded without its top (classification) layers using include_top=False, making it ready for transfer learning.

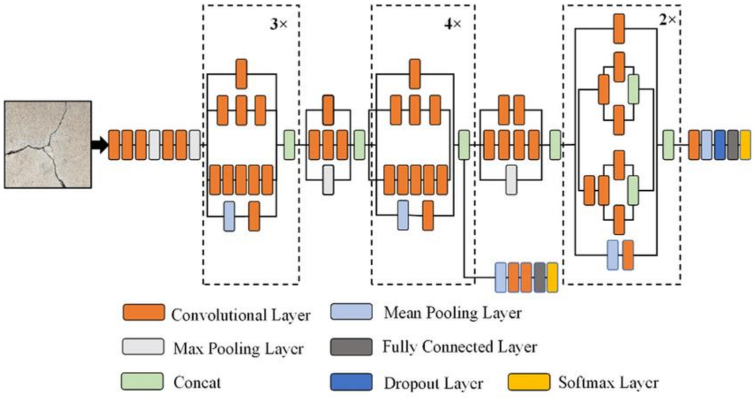
Additional custom layers include Spatial Dropout, and Attention Block.

The output layer is a dense layer with num_classes units and a SoftMax activation, generating class probabilities for each output class. An Adam optimizer is configured using the provided learning rate, beta values, and decay.

InceptionV3

A type of Convolutional Neural Network (CNN) with 207 layers and 24,529,764 parameters of which 24,495,332 are trainable.

It contains layers for Conv2D (94), BatchNormalization (47), Activation (47), MaxPooling2D (4), AveragePooling2D (8), GlobalAveragePooling2D (1), Dropout (2), Dense (2), AttentionBlock (1), and SEBlock (1).



The fine tuning allows selective layer freezing, custom layers, and additional dropout for regularization.

Custom layers include Spatial Dropout, Attention and SE Blocks, and Global Average Pooling.

Fully Connected Layers: fc_layers is used to create one or more fully connected layers after feature extraction.

Each layer applies ReLU activation, followed by dropout to prevent overfitting.

The output layer is a dense layer with num_classes neurons and SoftMax activation for multi-class classification.

Model Pipeline

The function, run_model_pipeline manages the setup, training, and evaluation of a neural network model. It centralizes the workflow of model initialization, compilation, and training.

Function Parameters

model_func, model_name: Model function and name for architecture setup and logging.

hyperparams, input_shape, num_classes: Hyperparameters, input dimensions, and output classes for model configuration.

train_generator, test_generator: Data generators for training and testing.

epochs, accum_steps: Training duration and steps for gradient accumulation.

Filter Hyperparameters

The code retrieves the signature of model_func using inspect.signature(model_func).

Filters hyperparams so only relevant parameters are passed, improving flexibility across different model architectures.

For example, in DenseNet, the dropout parameter is removed as it is not needed.

Model Creation and Summary

model_func initializes the model with input_shape, num_classes, and hyperparams.

A summary of the model is printed and saved to a text file, enabling verification and documentation.

Optimizer Selection

It uses the optimizer specified in hyperparams['optimizer_type'] with a default to 'Adam' if unspecified.

It sets learning_rate, beta_1, beta_2, and decay for Adam, momentum for SGD, and rho for RMSprop based on hyperparams. If an unsupported optimizer type is used, an error is raised.

Model Compilation

Compiles the model with optimizer, categorical_crossentropy loss, and custom metrics imported from metrics.py.

Callbacks

The function, create_callbacks, generates a list of callbacks to be applied during model training to manage training events like early stopping and learning rate adjustments.

Callback Parameters

It sets monit_acc and monit_loss to monitor the top-3 accuracy and validation loss, respectively.

It uses mode ('max' for accuracy, 'min' for loss) to track the best performing epoch for each metric.

Callback Functions

The following callback functions have been used:

EarlyStopping

Monitors validation loss, stopping training if it does not improve after a specified patience (5 epochs). Restores the best weights upon stopping.

ReduceLROnPlateau

Reduces the learning rate by a factor of 0.2 when the validation loss plateaus for two epochs. Ensures learning does not stagnate.

ModelCheckpoint

Saves the model whenever top-3 validation accuracy improves, storing the model file for each fold.

Gradient Accumulation

The callbacks add a gradient accumulation callback, GradientAccumulation(accum_steps=accum_steps), to handle large batches when memory is limited and is called at the end of each training batch.

It accumulates gradients across accum_steps batches before applying the weight updates allowing for larger effective batch sizes.

Training and Saving Results

Model is trained with the specified epochs and callbacks, and its history (training performance metrics) is saved for further analysis.

The final model is saved as an .h5 file, and training results are plotted using plot_train_results.

Results

The results have been presented below.

Metrics Used

The following metrics have been used to measure the performance of the models:

1. categorical_accuracy
2. accuracy
3. TopKCategoricalAccuracy
4. Precision
5. Recall
6. Area under the Curve, AUC

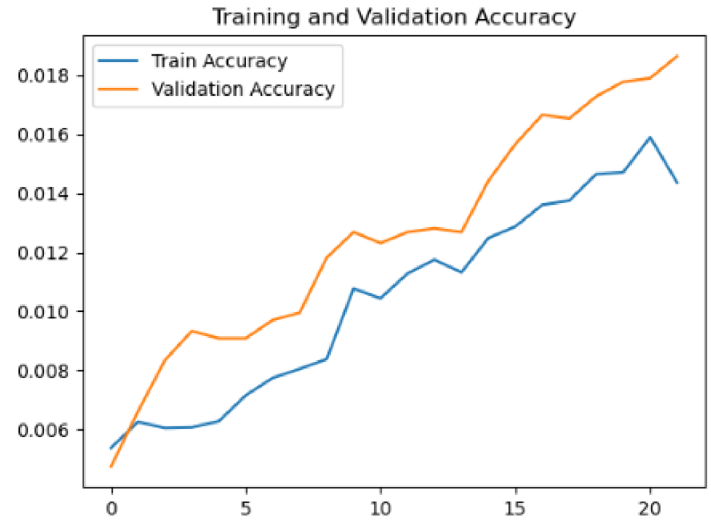
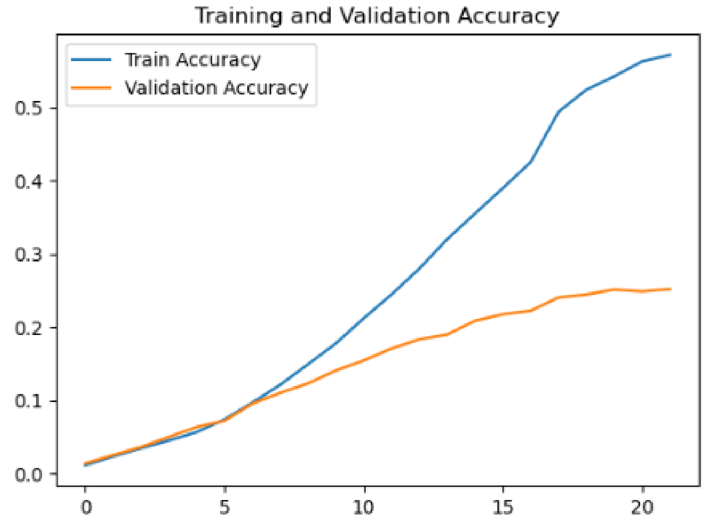
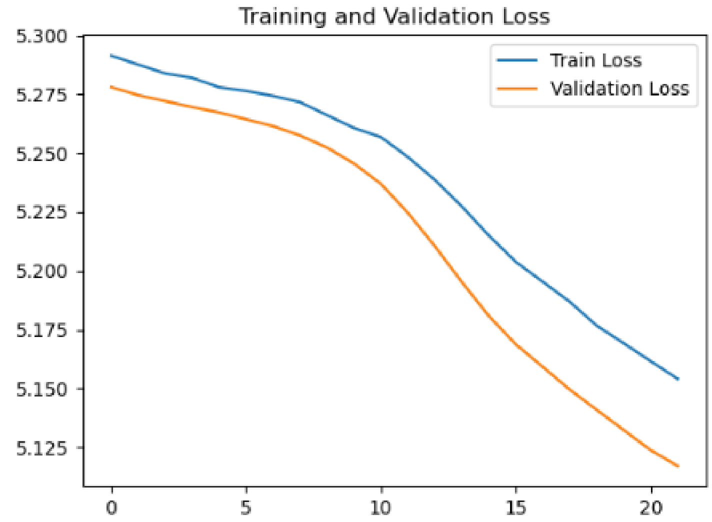
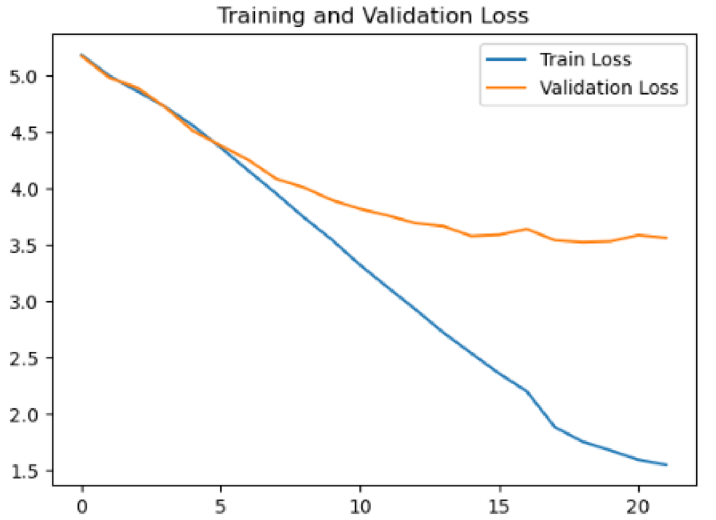
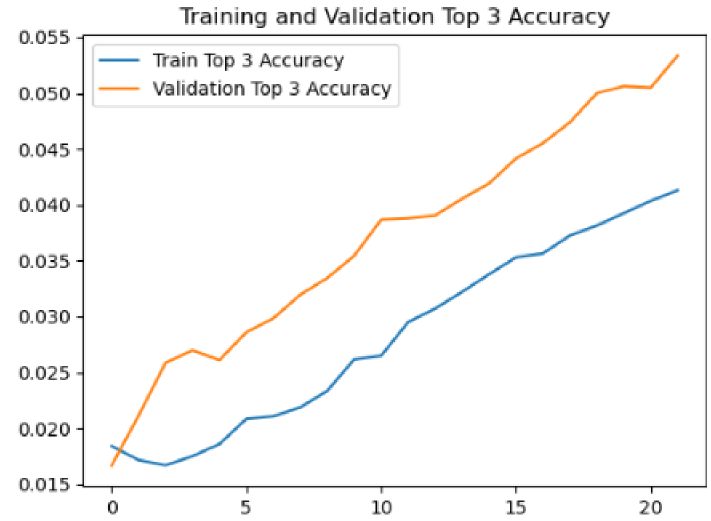
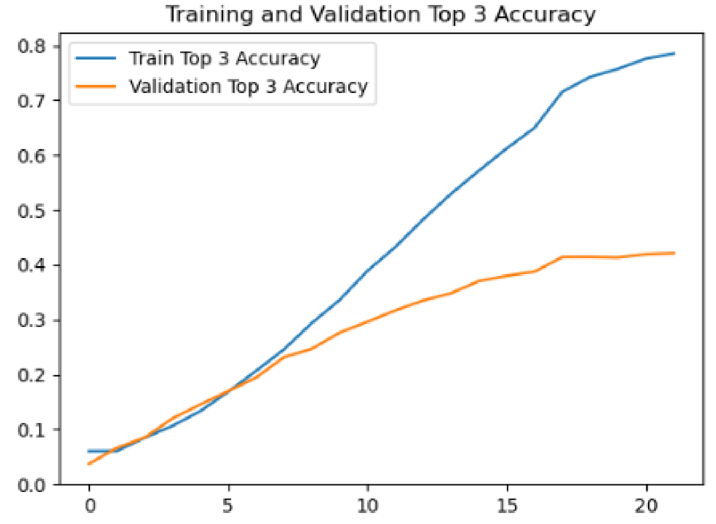
Execution time

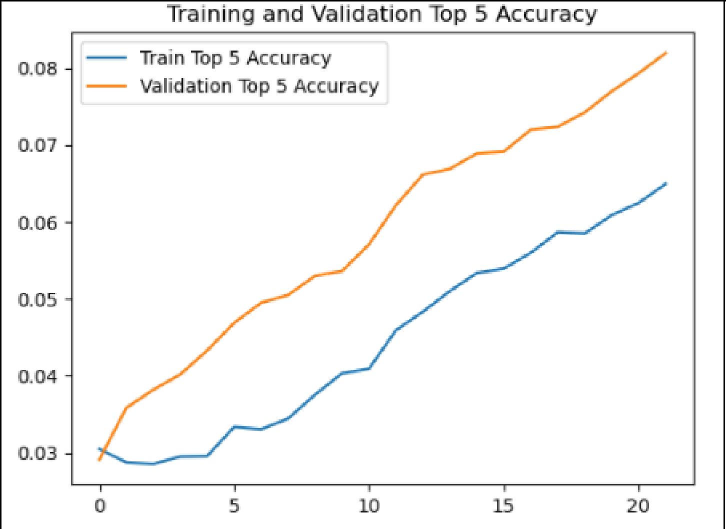
	Default hyper parameters		**Optimal hyper parameters**	
Data augmentation	**Execution time (minutes)**	**CPU Execution Time (minutes)**	**Execution time (minutes)**	**CPU Execution Time (minutes)**
Data augmentation	30.645	4.52	30.645	4.52
Hyper parameter Search	N/A	N/A	N/A	N/A
model1	187.98	90.26	185.31	89.47
AlexNet	184.83	101.20	187.66	90.80
VGG16	193.18	136.59		
ResNet50	187.2	94.63	186.12	110.22
DenseNet121	195.61	104.76	190.37	117.01
ImageNet	192.86	104.40	191.60	117.94
EfficientNetB0	200.31	105.54	190.43	113.55
InceptionV3	200.09	112.44	188.71	113.33
Total (hours)	27+	15+	23+	13+

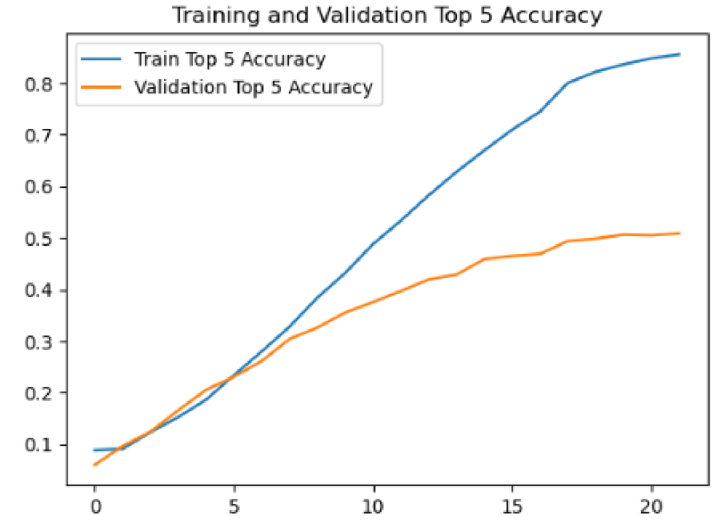
On an average, 30 hours were needed to run the notebook once and over 15 iterations had been made during the evolution of the code.

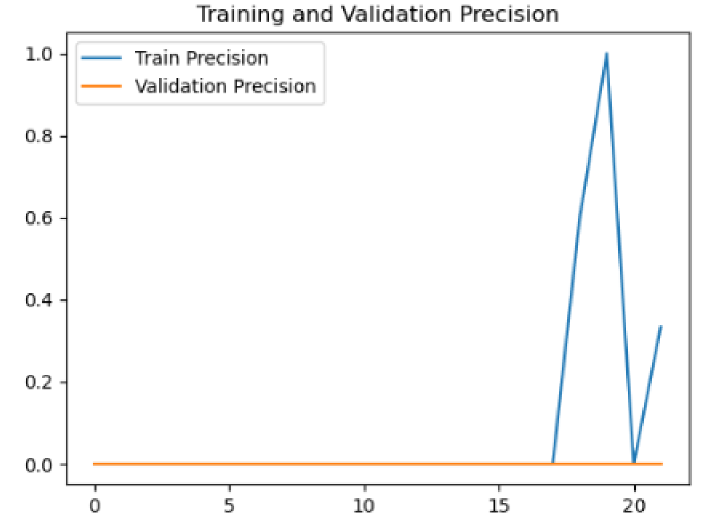
Results of the various models

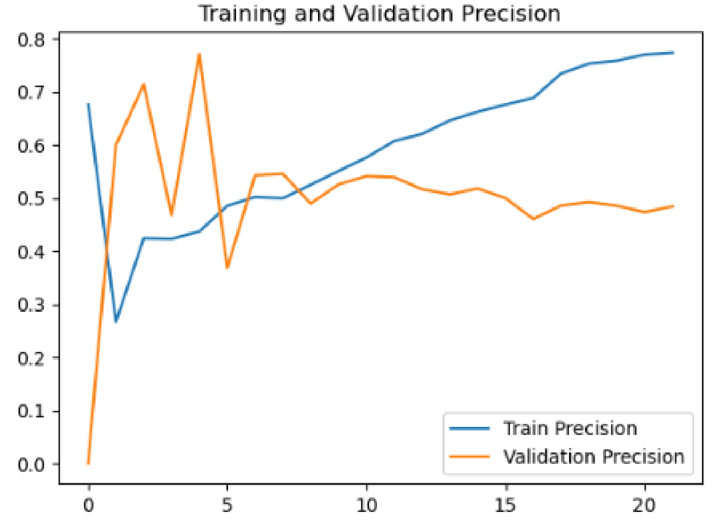
AlexNet Results

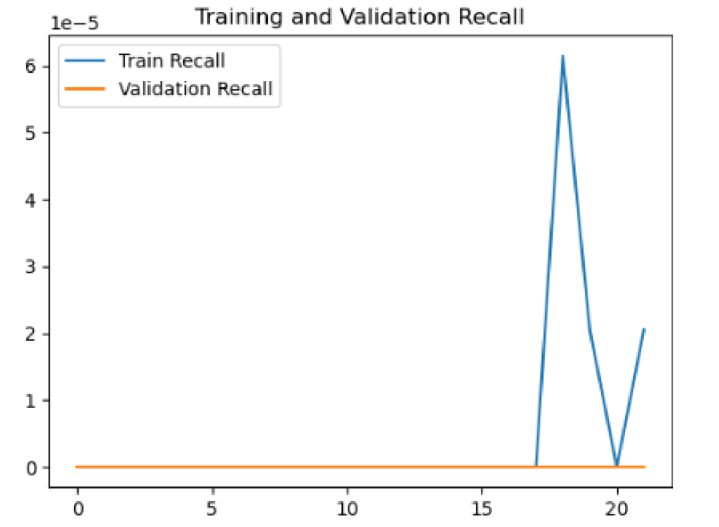
With Default hyper-parameters	With optimal hyper-parameters
	
	
	

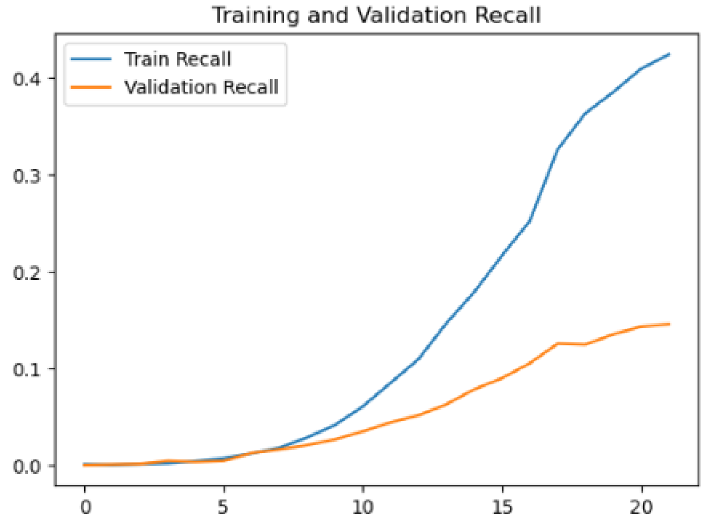

Training and Validation Top 5 Accuracy
Train Top 5 Accuracy
Validation Top 5 Accuracy
0.08
0.07
0.06
0.05
0.04
0.03
0
5
10
15
20

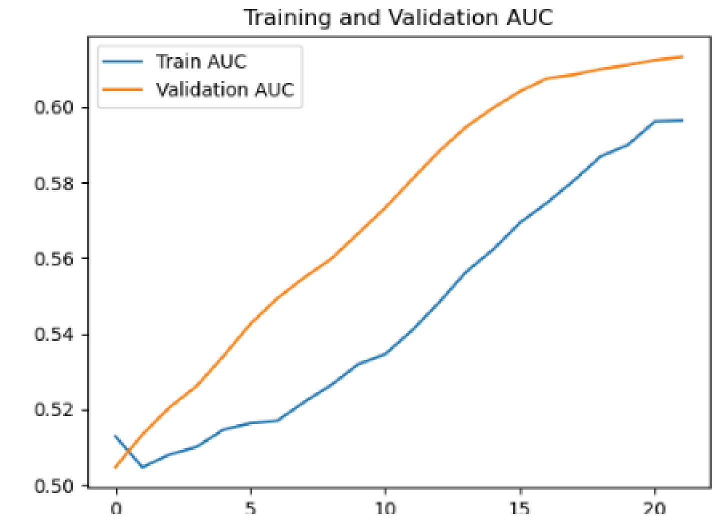
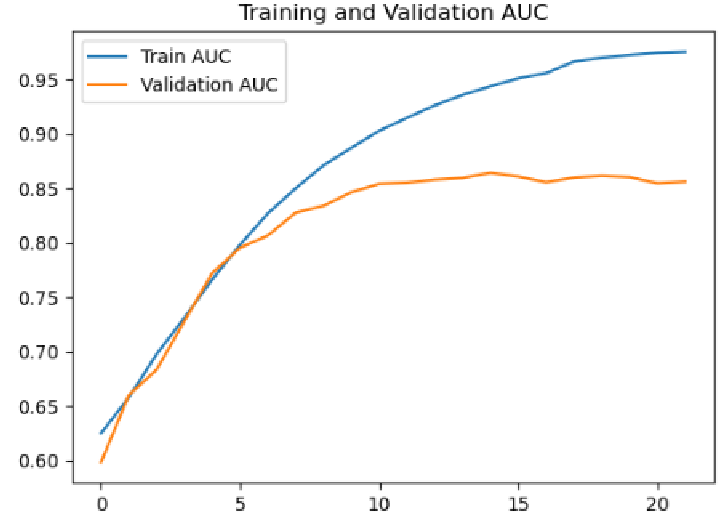

Training and Validation Top 5 Accuracy
Train Top 5 Accuracy
Validation Top 5 Accuracy
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
5
10
15
20


Training and Validation Precision
Train Precision
Validation Precision
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20


Training and Validation Precision
Train Precision
Validation Precision
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0
5
10
15
20

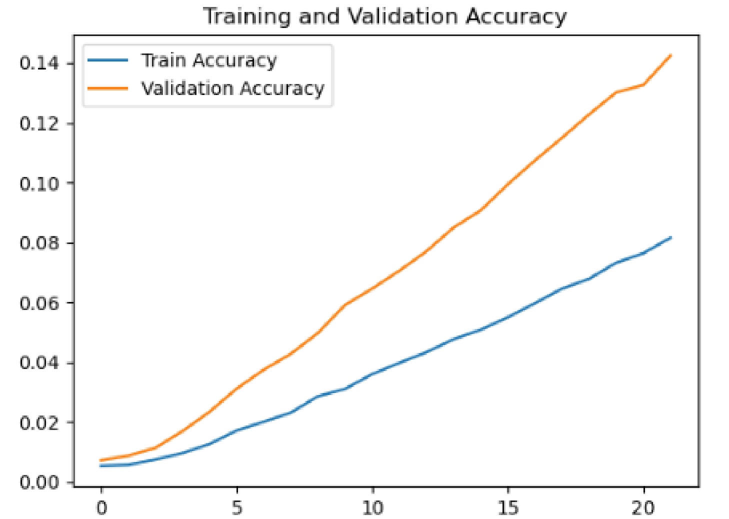
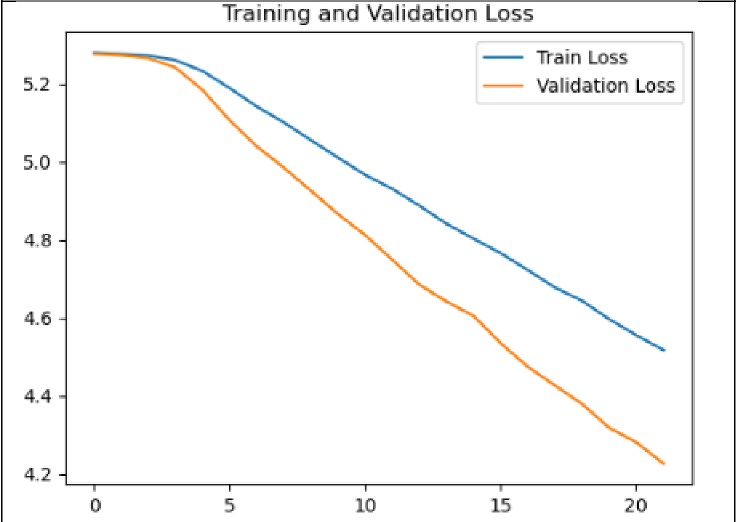

Training and Validation Recall
1e−5
Train Recall
Validation Recall
6
5
4
3
2
1
0
0
5
10
15
20

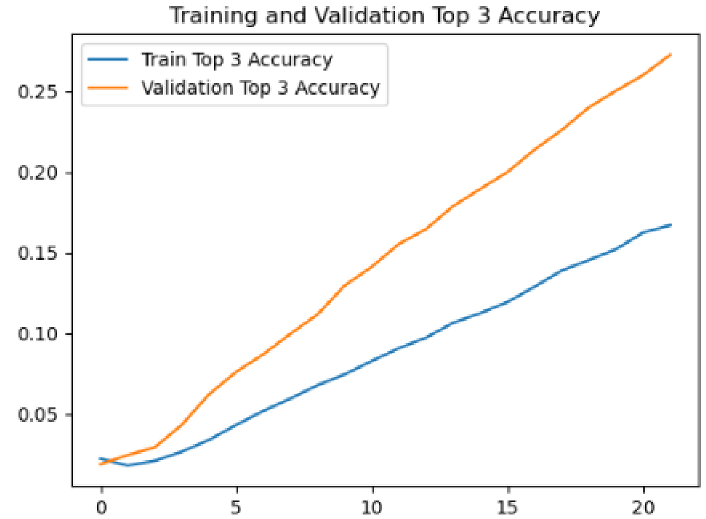

Training and Validation Recall
Train Recall
Validation Recall
0.4
0.3
0.2
0.1
0.0
0
5
10
15
20

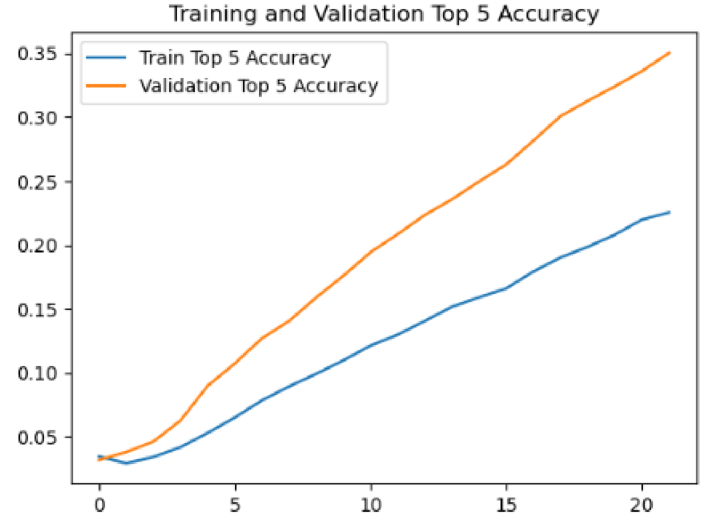
	

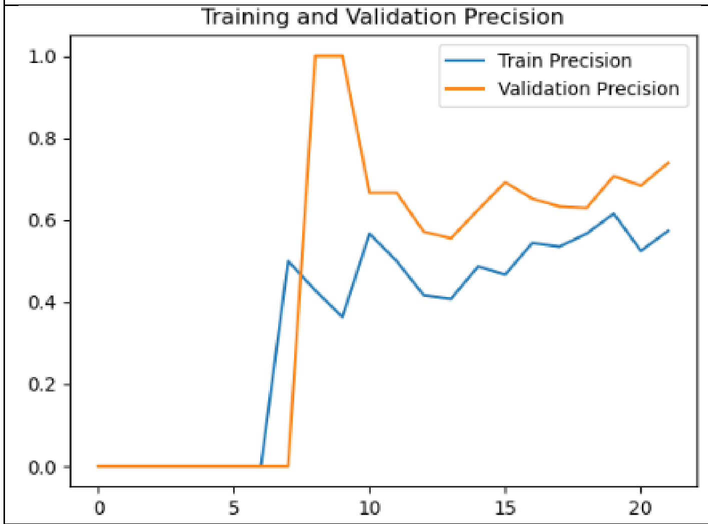
VGG16 Results

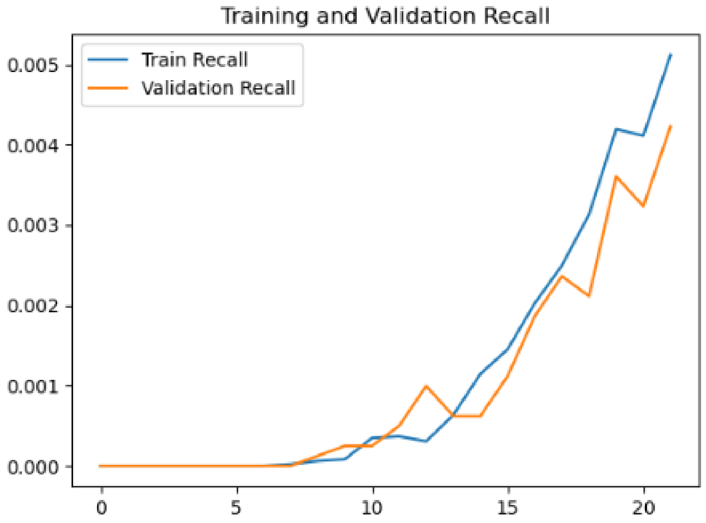
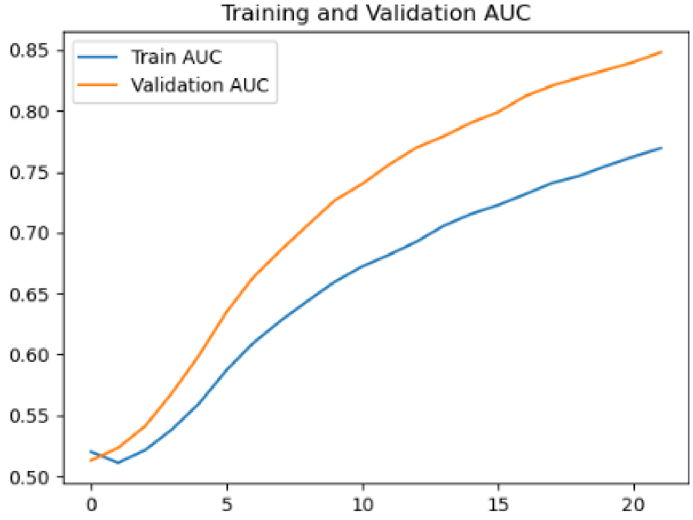
There are not results for optimal hyper parameters as the code breaks possibly due to a high learning rate which produces NaNs or negative values in the predictions which is unexpected for a classification problem or there could have been a gradient overflow due to the gradient accumulation or mixed precision.

With Default hyper-parameters	**With optimal hyper-parameters**
	
	

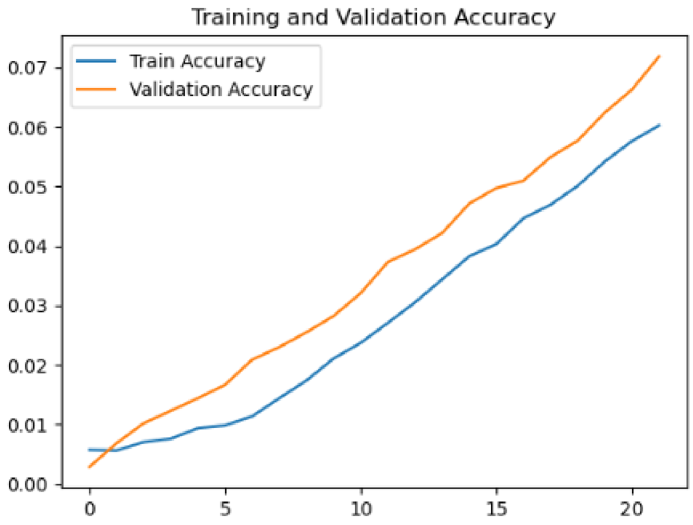
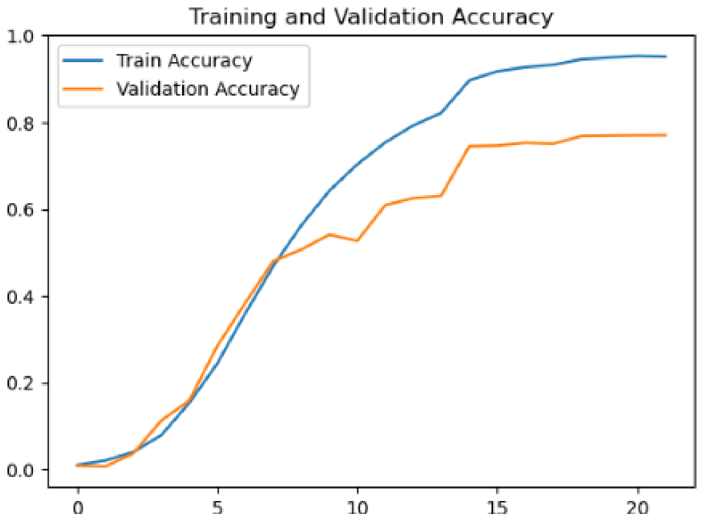

Training and Validation Top 3 Accuracy
Train Top 3 Accuracy
Validation Top 3 Accuracy
0.25
0.20
0.15
0.10
0.05
0
5
10
15
20

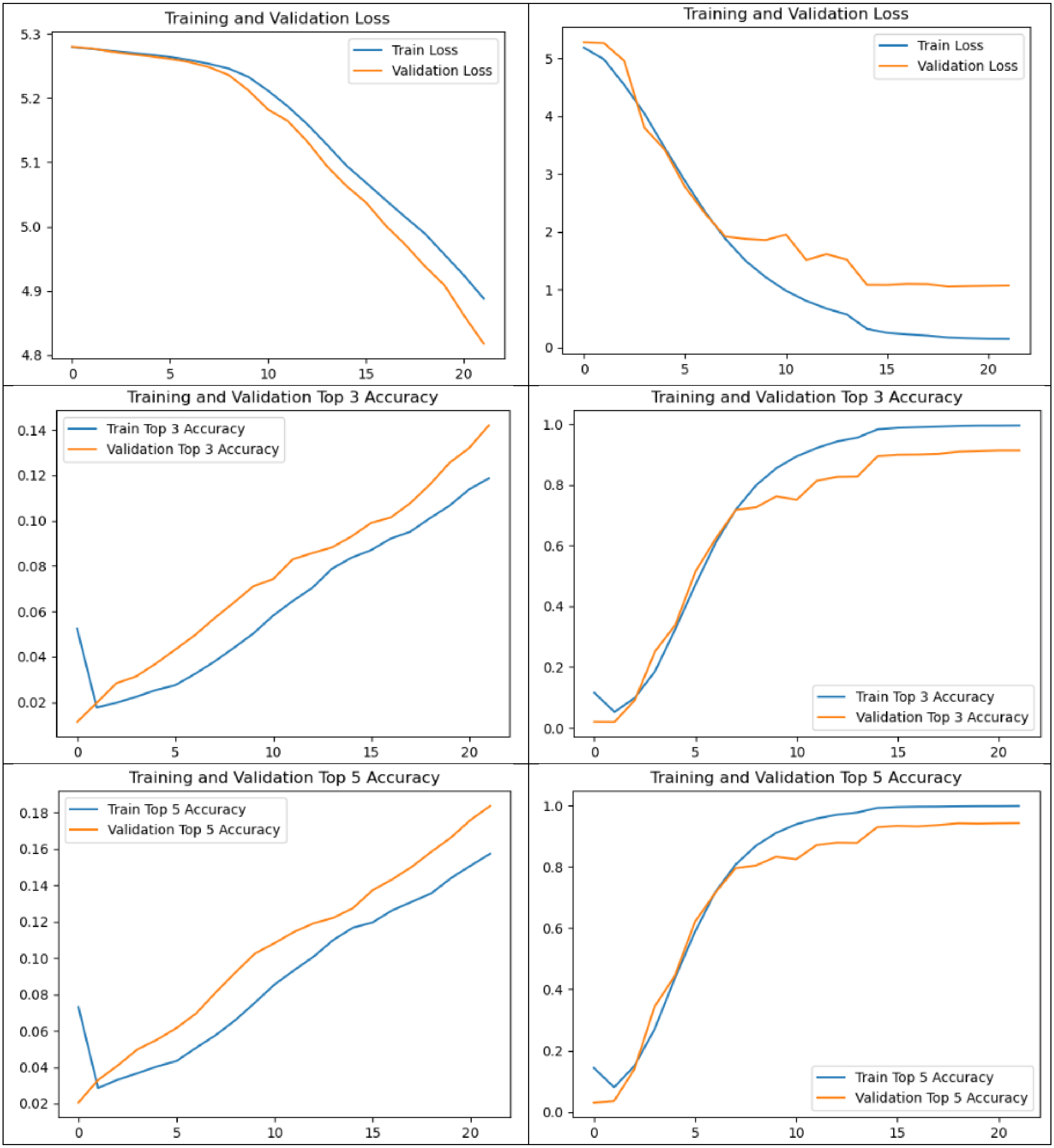

Training and Validation Top 5 Accuracy
Train Top 5 Accuracy
Validation Top 5 Accuracy
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0
5
10
15
20

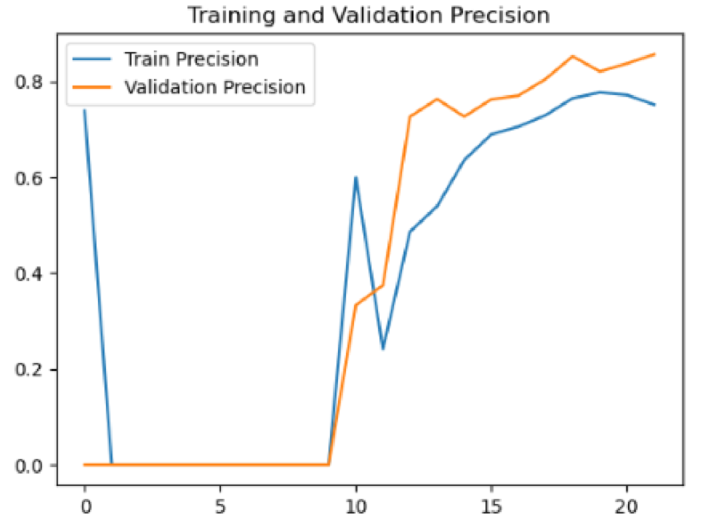

Training and Validation Precision
Train Precision
Validation Precision
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20

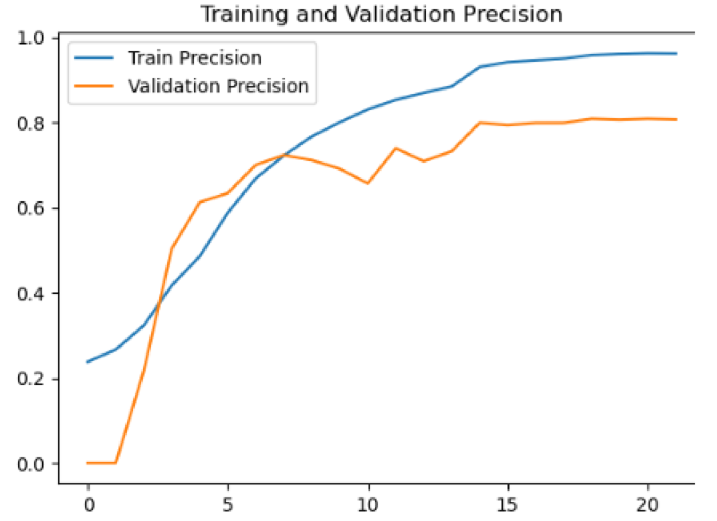
 Training and Validation Recall	
 Training and Validation AUC	

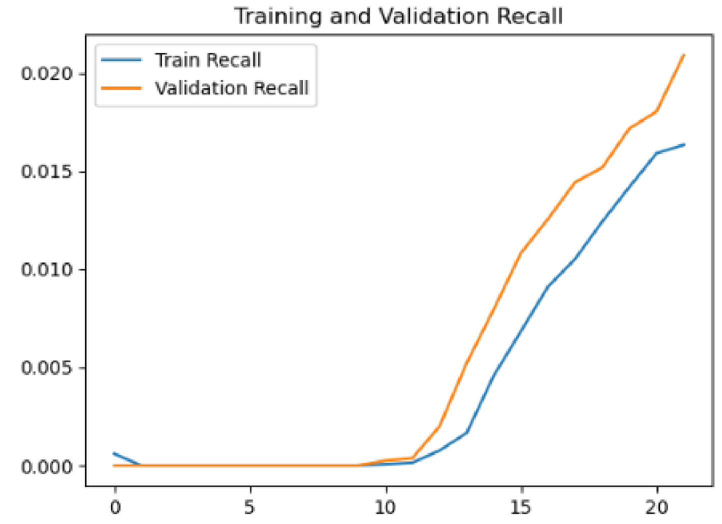
ResNet50 Results

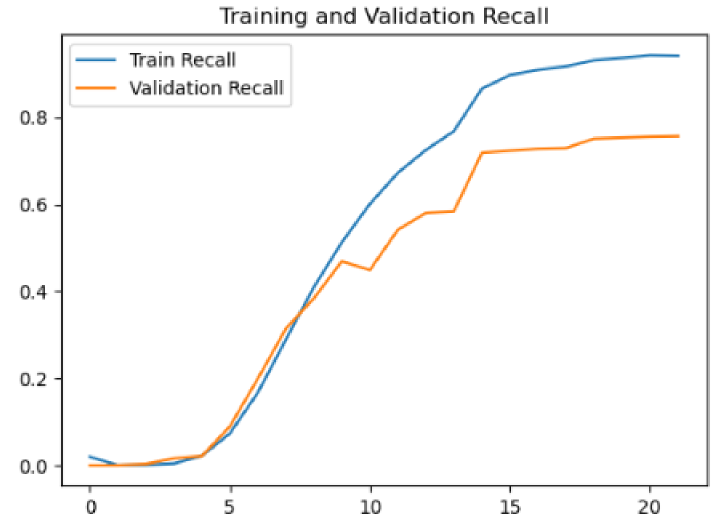
With Default hyper-parameters	With optimal hyper-parameters
 Training and Validation Accuracy	 Training and Validation Accuracy

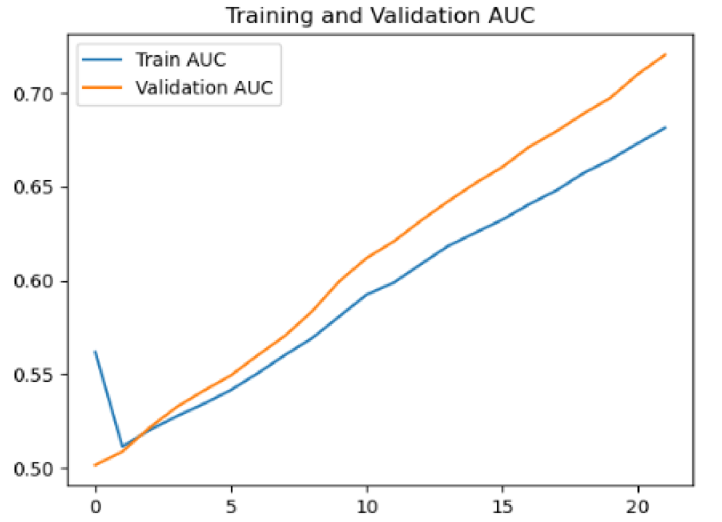

Training and Validation Loss
Train Loss
Validation Loss
5.3
5.2
5.1
5.0
4.9
4.8
0
5
10
15
20
Training and Validation Loss
Train Loss
Validation Loss
5
4
3
2
1
0
0
5
10
15
20
Training and Validation Top 3 Accuracy
Train Top 3 Accuracy
Validation Top 3 Accuracy
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0
5
10
15
20
Training and Validation Top 3 Accuracy
Train Top 3 Accuracy
Validation Top 3 Accuracy
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
Training and Validation Top 5 Accuracy
Train Top 5 Accuracy
Validation Top 5 Accuracy
0.18
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0
5
10
15
20
Training and Validation Top 5 Accuracy
Train Top 5 Accuracy
Validation Top 5 Accuracy
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20

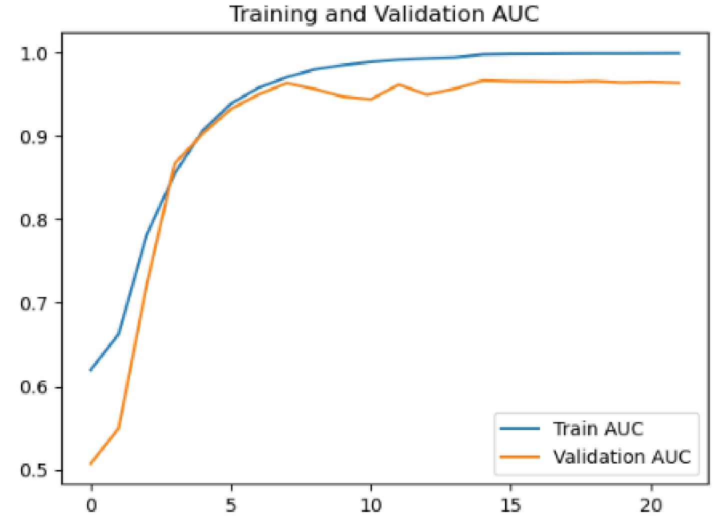

Training and Validation Precision
Train Precision
Validation Precision
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20


Training and Validation Precision
Train Precision
Validation Precision
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20

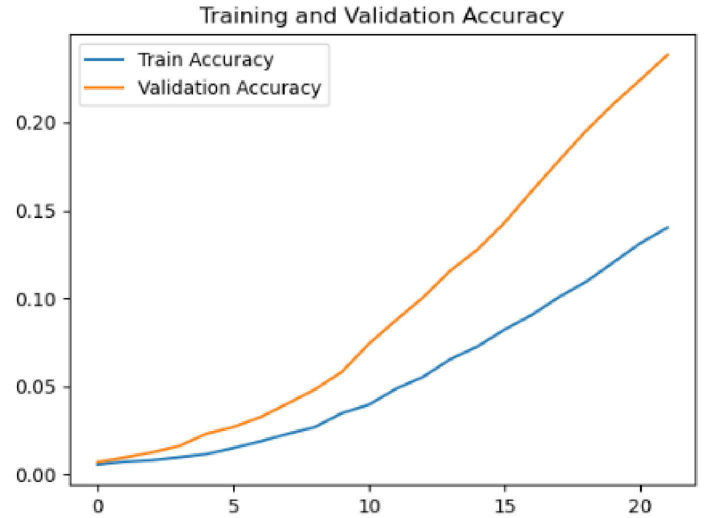
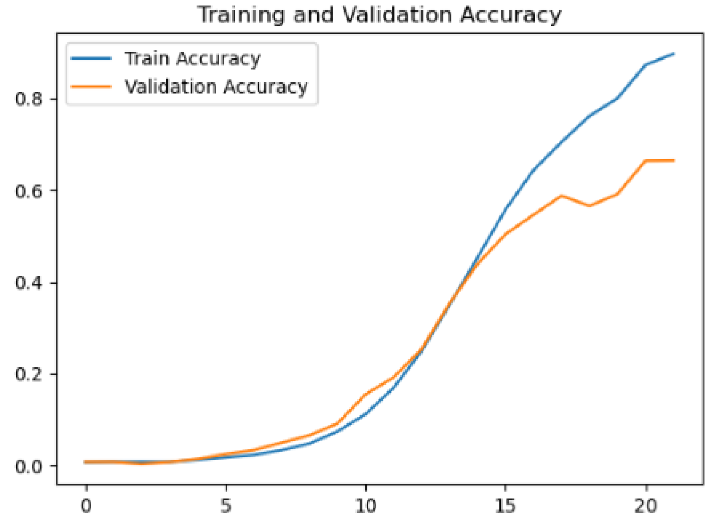
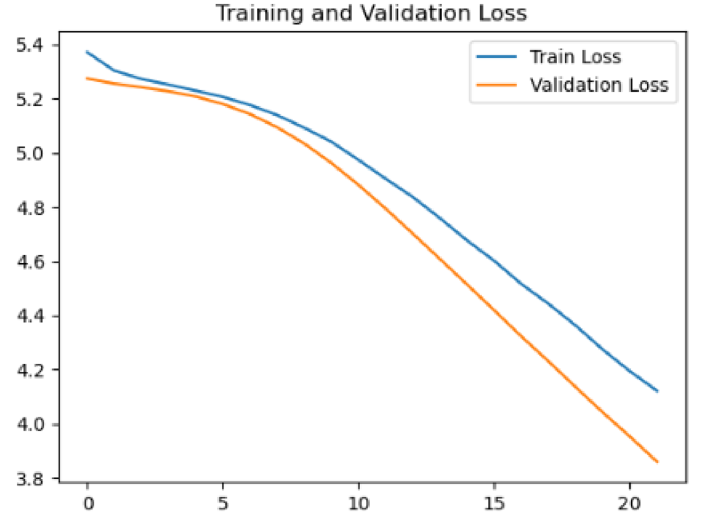
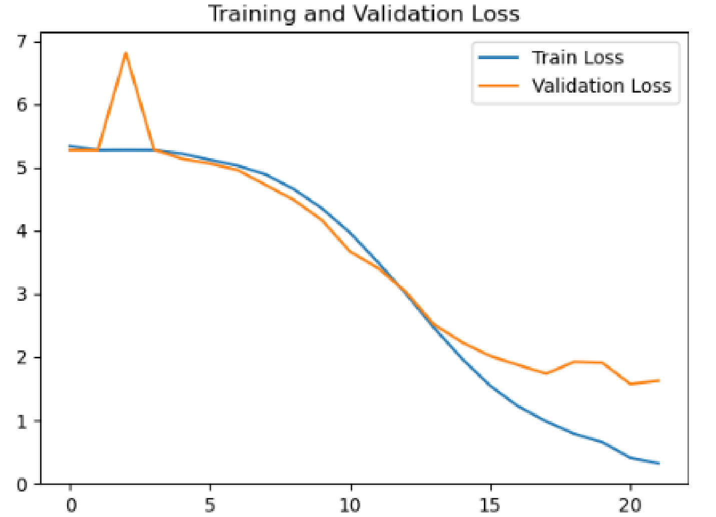
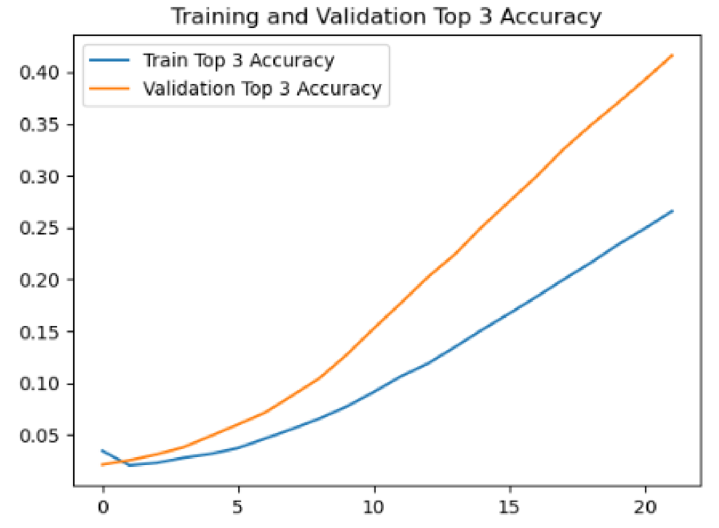
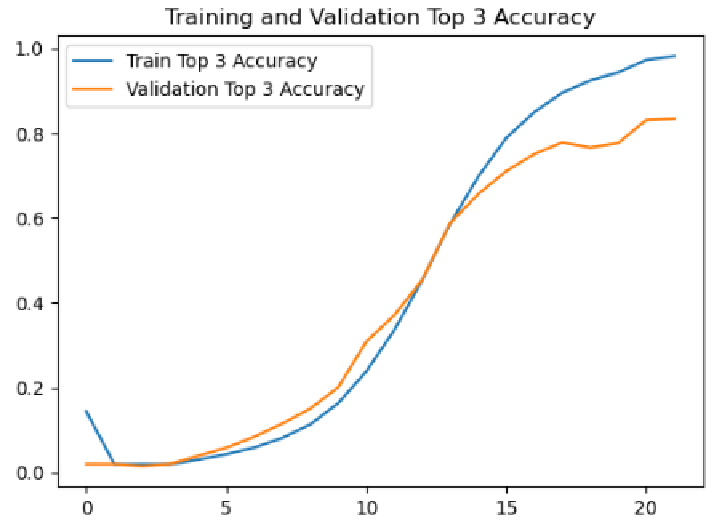

Training and Validation Recall
Train Recall
Validation Recall
0.020
0.015
0.010
0.005
0.000
0
5
10
15
20

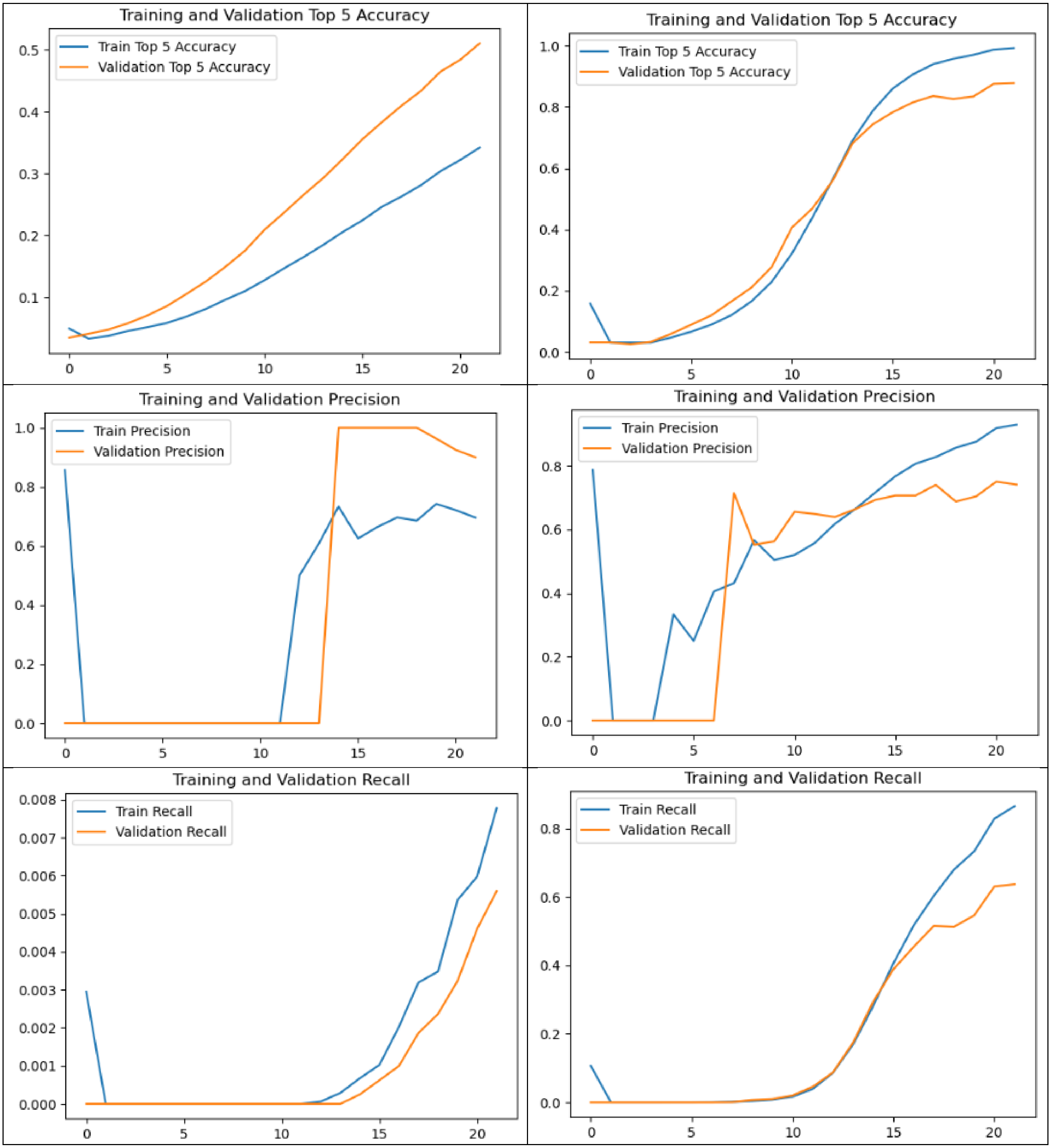

Training and Validation Recall
Train Recall
Validation Recall
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20

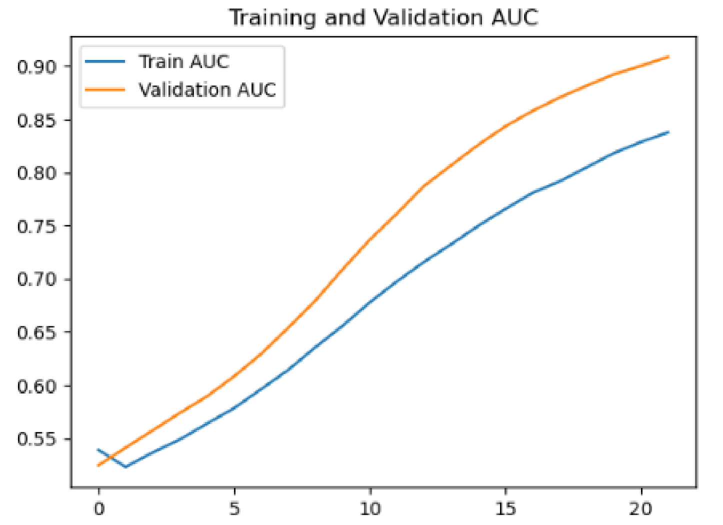
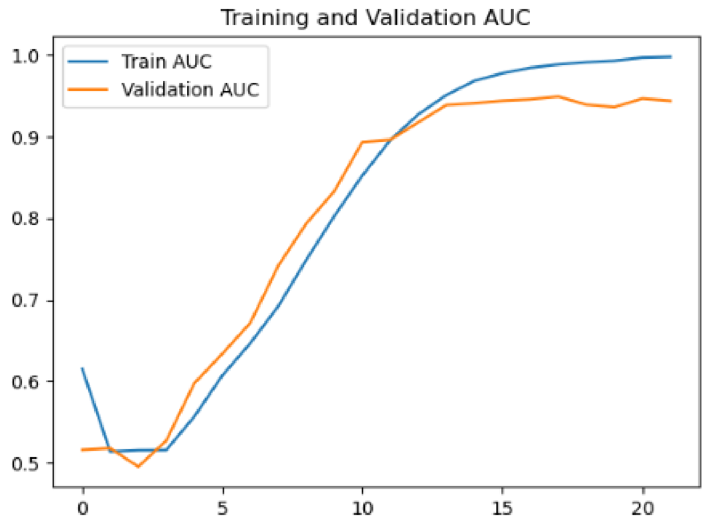

Training and Validation AUC
Train AUC
Validation AUC
0.70
0.65
0.60
0.55
0.50
0
5
10
15
20


Training and Validation AUC
Train AUC
Validation AUC
1.0
0.9
0.8
0.7
0.6
0.5
0
5
10
15
20

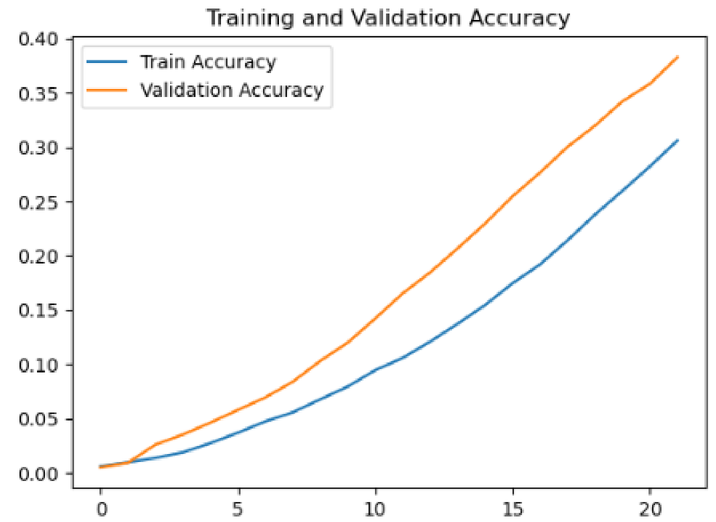
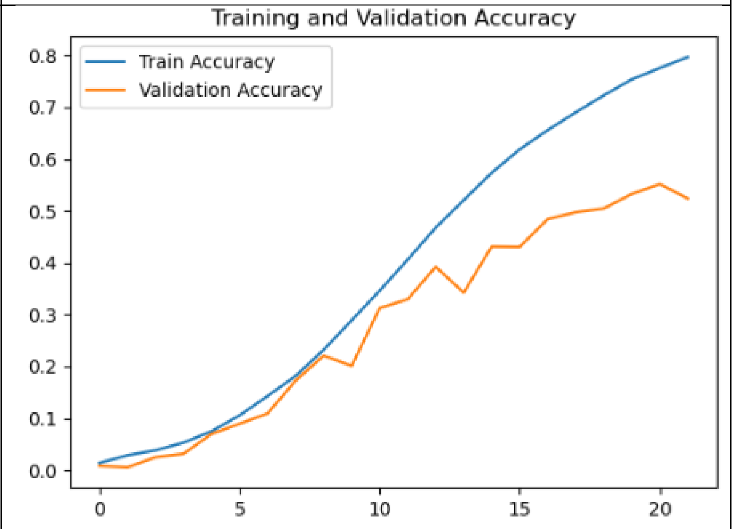
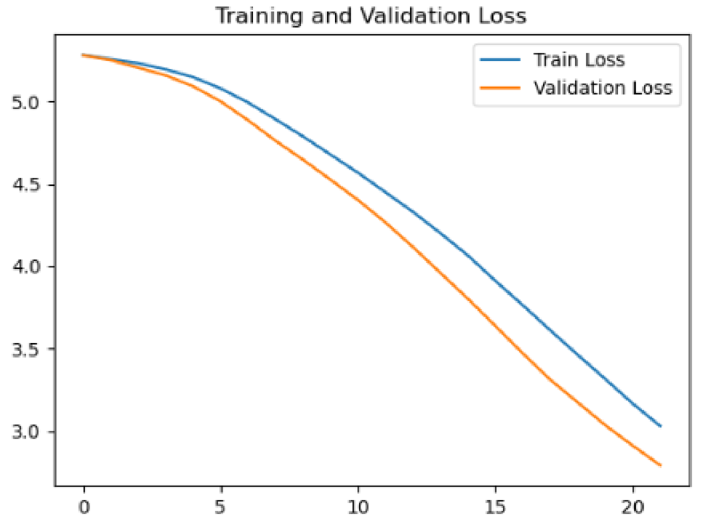
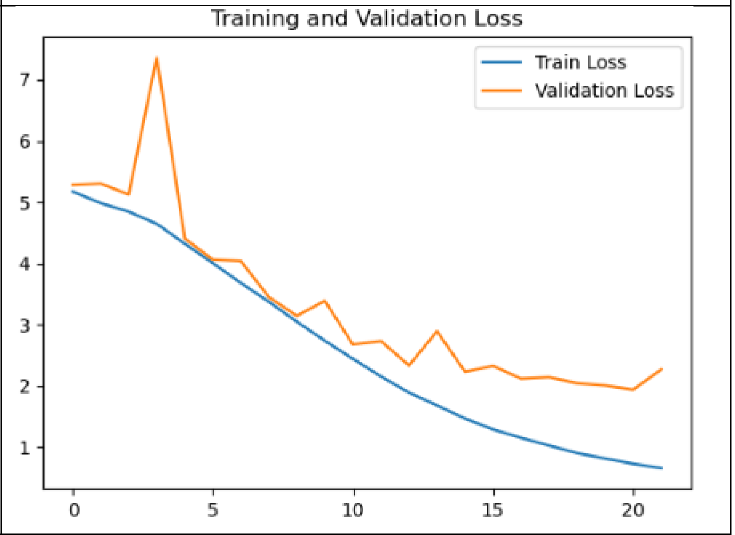
DenseNet121 Results

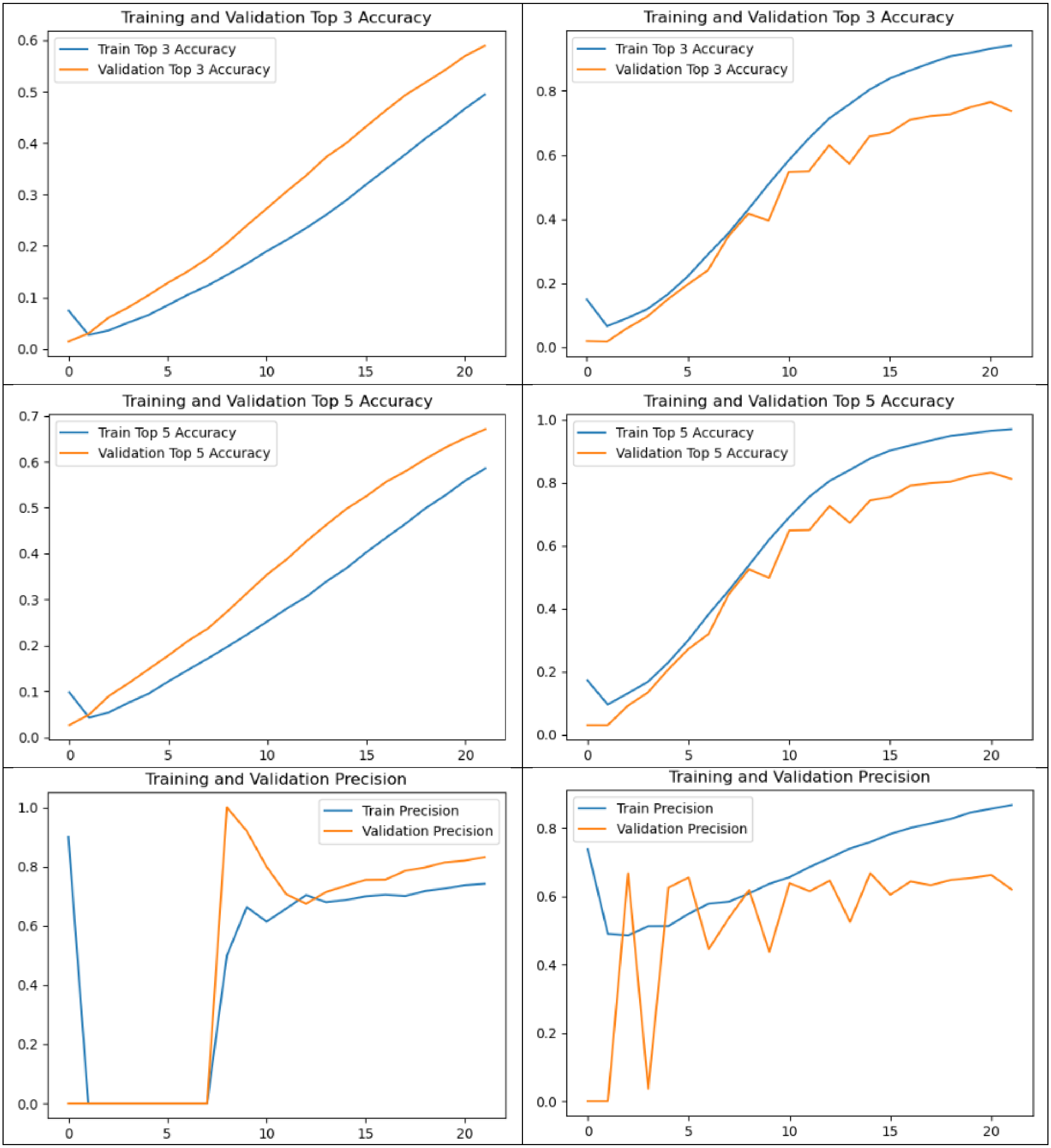
With Default hyper-parameters	With optimal hyper-parameters
	
	
	

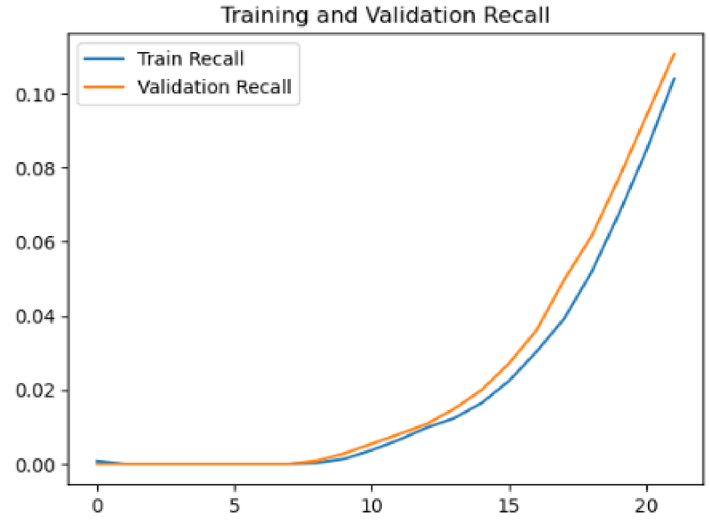
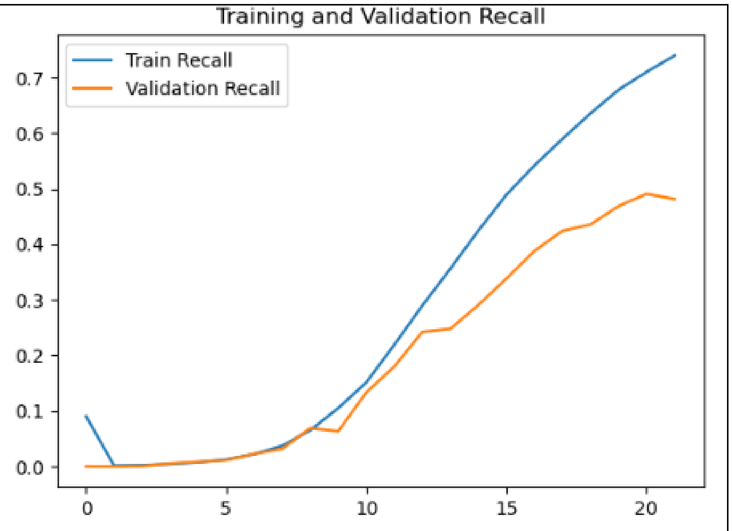
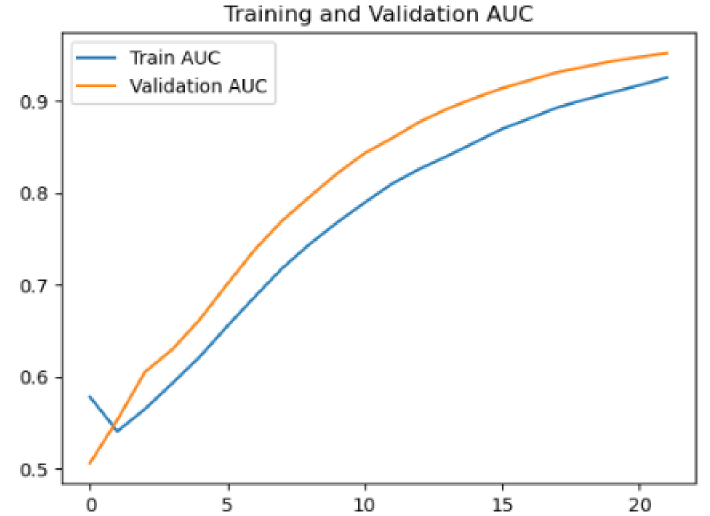
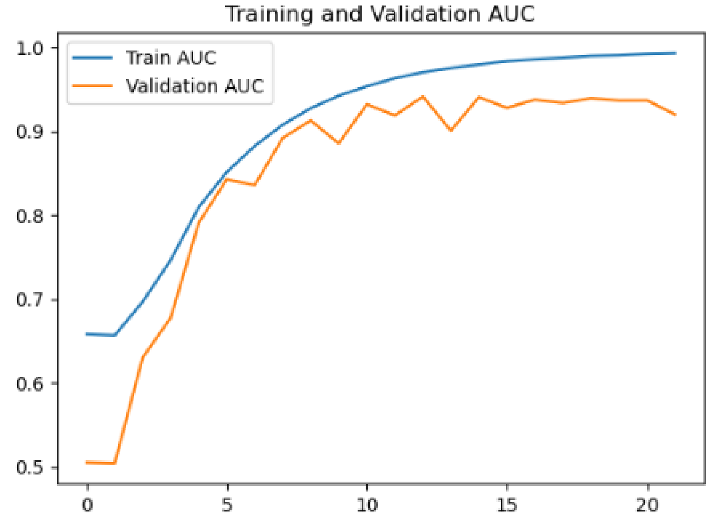

Training and Validation Top 5 Accuracy
Train Top 5 Accuracy
Validation Top 5 Accuracy
0.5
0.4
0.3
0.2
0.1
0
5
10
15
20
Training and Validation Top 5 Accuracy
Train Top 5 Accuracy
Validation Top 5 Accuracy
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
Training and Validation Precision
Train Precision
Validation Precision
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
Training and Validation Precision
Train Precision
Validation Precision
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
Training and Validation Recall
Train Recall
Validation Recall
0.008
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0.000
0
5
10
15
20
Training and Validation Recall
Train Recall
Validation Recall
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20

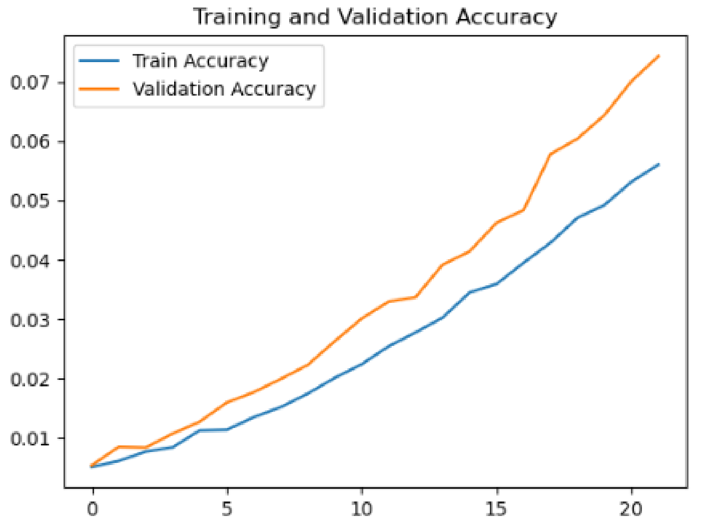
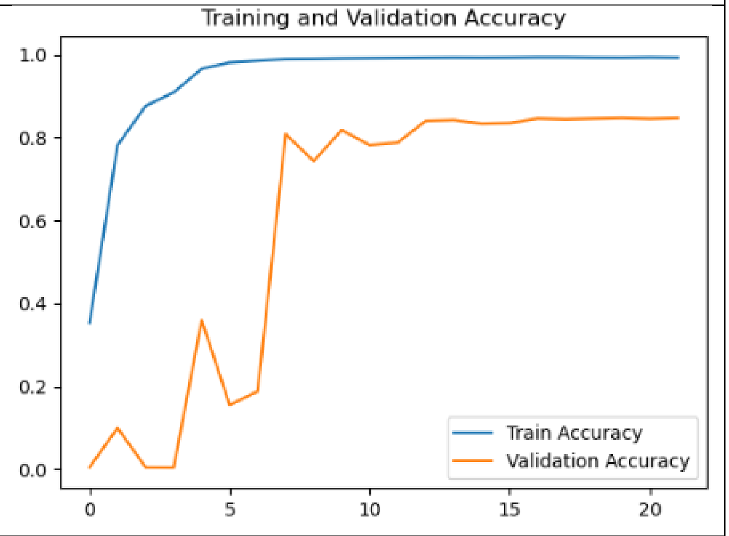
ImageNet Results

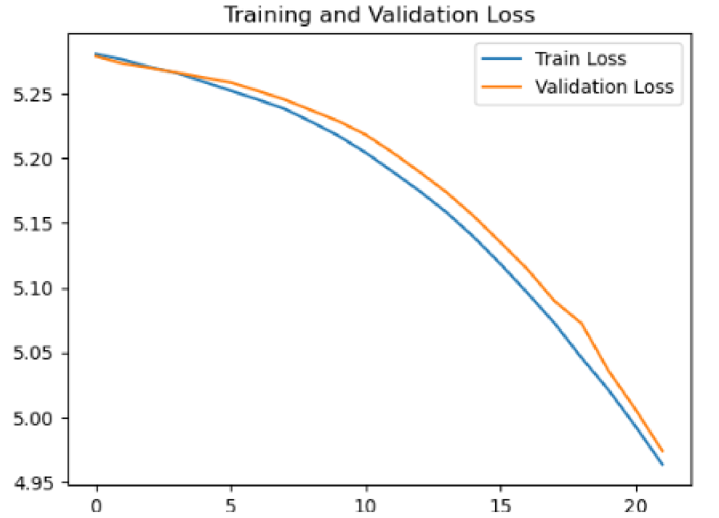
With Default hyper-parameters	With optimal hyper-parameters
	
	

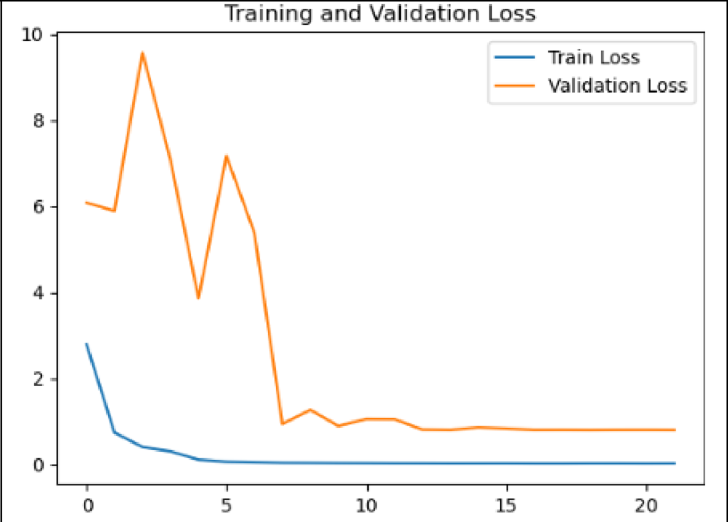

Training and Validation Top 3 Accuracy
Train Top 3 Accuracy
Validation Top 3 Accuracy
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0
5
10
15
20
Training and Validation Top 3 Accuracy
Train Top 3 Accuracy
Validation Top 3 Accuracy
0.0
0.2
0.4
0.6
0.8
0
5
10
15
20
Training and Validation Top 5 Accuracy
Train Top 5 Accuracy
Validation Top 5 Accuracy
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0
5
10
15
20
Training and Validation Top 5 Accuracy
Train Top 5 Accuracy
Validation Top 5 Accuracy
0.0
0.2
0.4
0.6
0.8
1.0
0
5
10
15
20
Training and Validation Precision
Train Precision
Validation Precision
0.0
0.2
0.4
0.6
0.8
1.0
0
5
10
15
20
Training and Validation Precision
Train Precision
Validation Precision
0.0
0.2
0.4
0.6
0.8
0
5
10
15
20

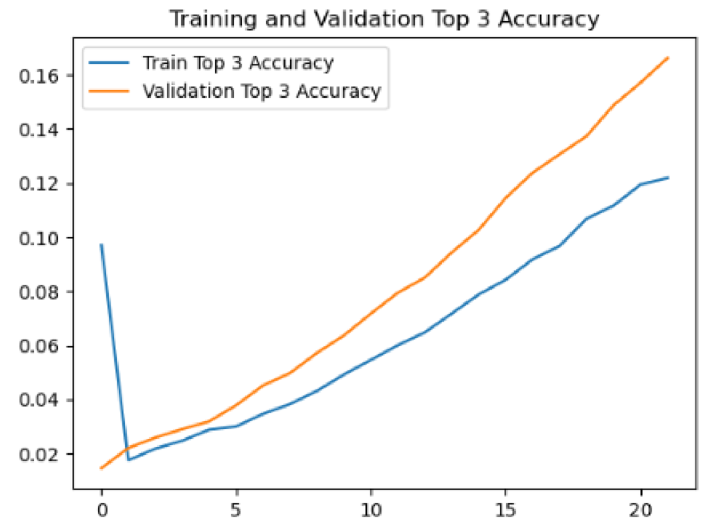
	
	

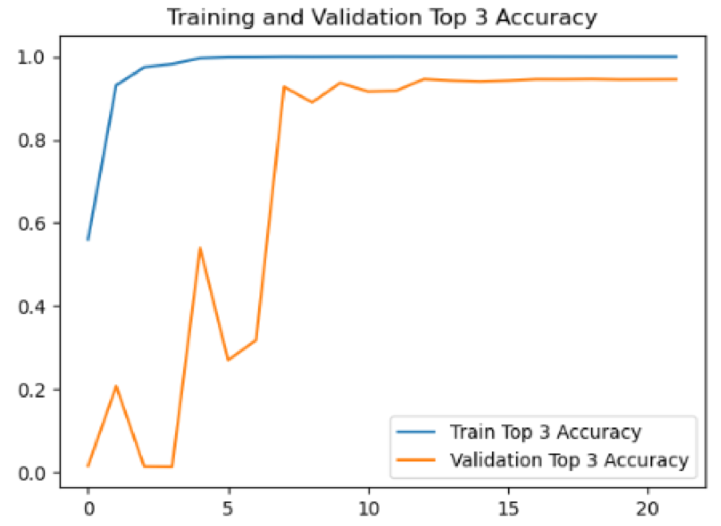
EfficientNetB0 Results

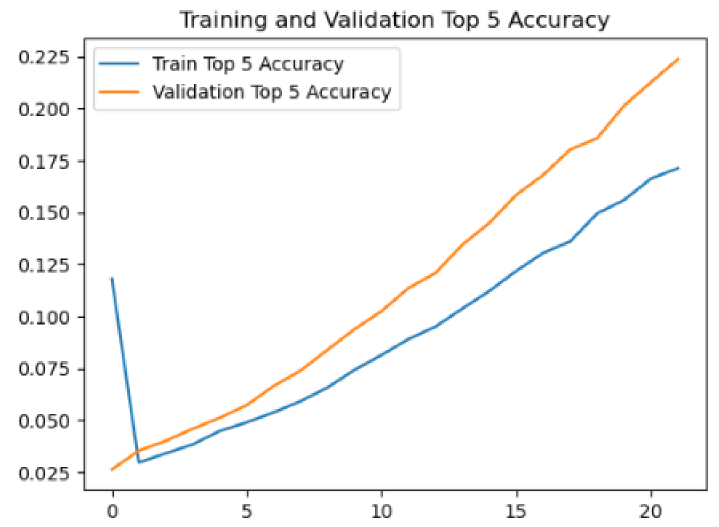
With Default hyper-parameters	With optimal hyper-parameters
	

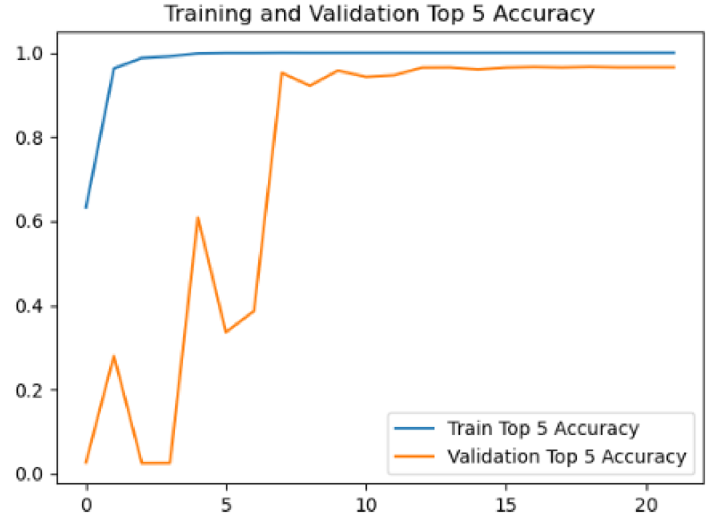

Training and Validation Loss
Train Loss
Validation Loss
5.25
5.20
5.15
5.10
5.05
5.00
4.95
0
5
10
15
20

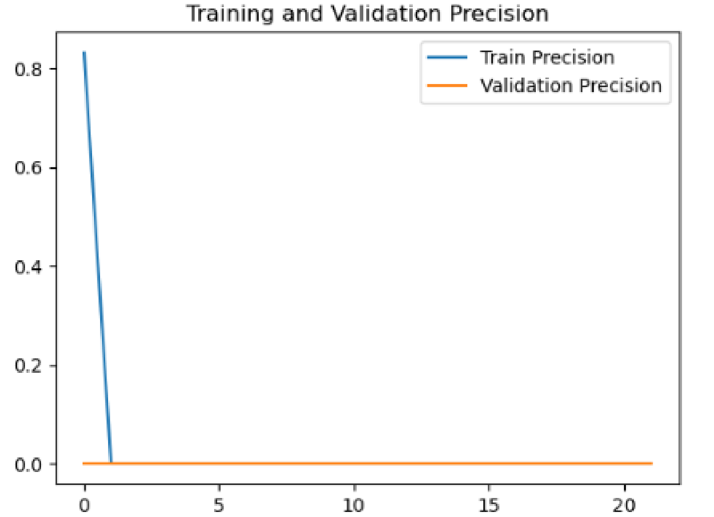
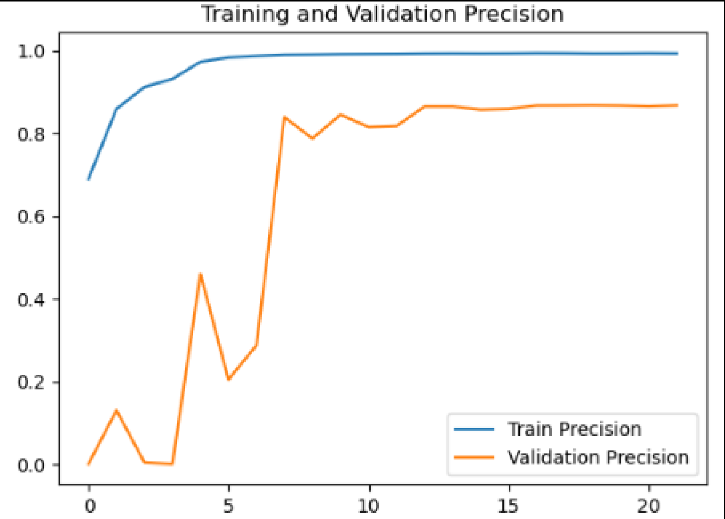
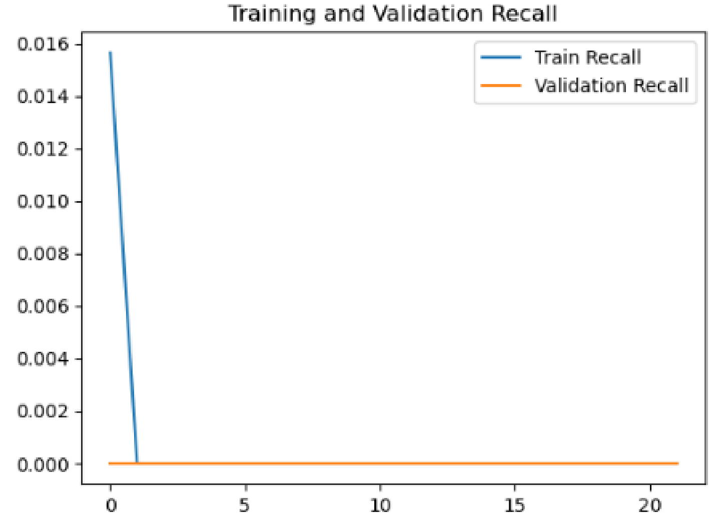
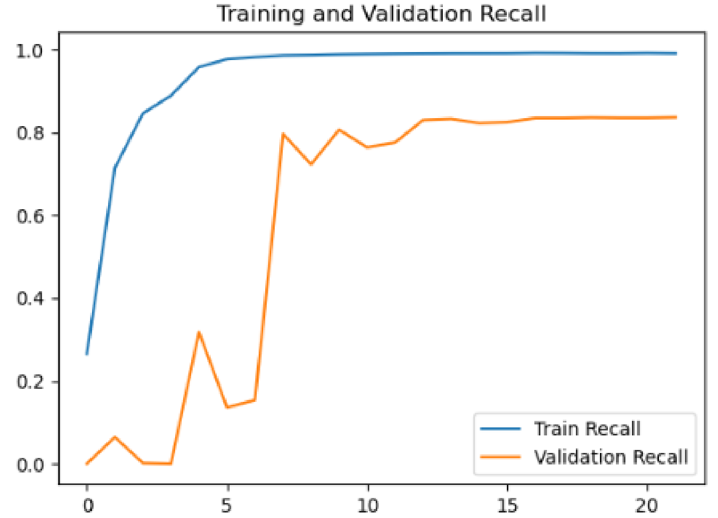
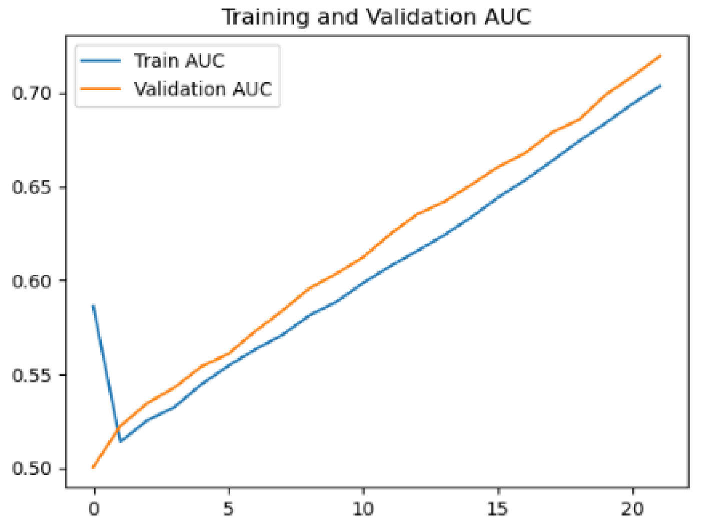
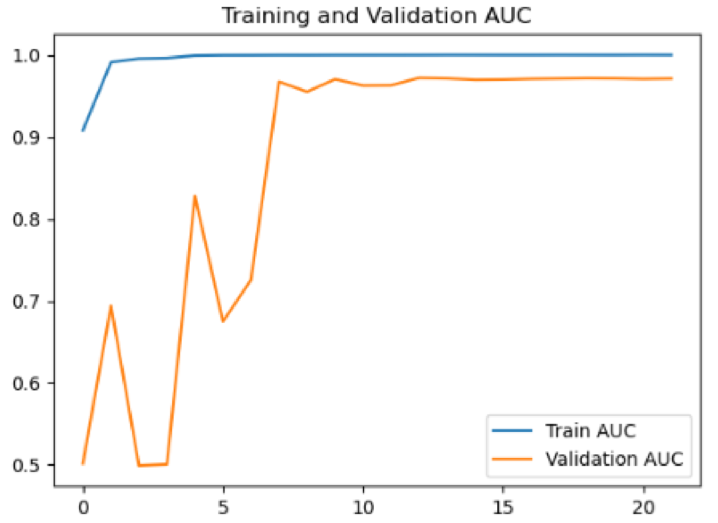

Training and Validation Loss
Train Loss
Validation Loss
10
8
6
4
2
0
0
5
10
15
20


Training and Validation Top 3 Accuracy
Train Top 3 Accuracy
Validation Top 3 Accuracy
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0
5
10
15
20


Training and Validation Top 3 Accuracy
Train Top 3 Accuracy
Validation Top 3 Accuracy
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20

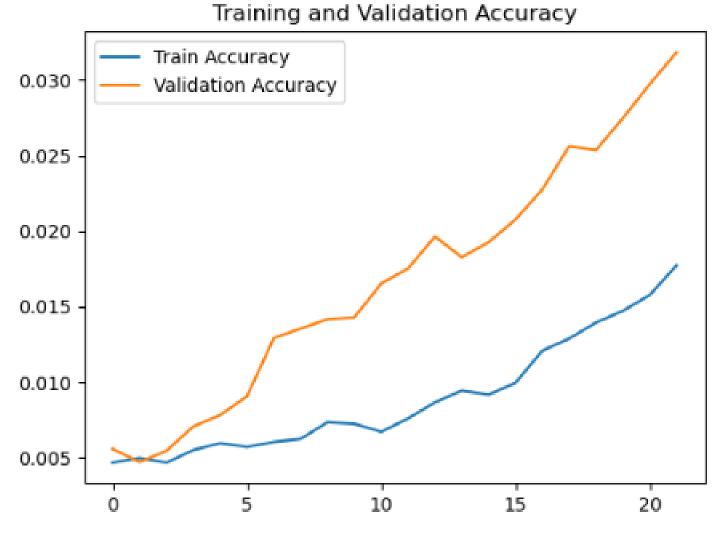

Training and Validation Top 5 Accuracy
Train Top 5 Accuracy
Validation Top 5 Accuracy
0.225
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0
5
10
15
20


Training and Validation Top 5 Accuracy
Train Top 5 Accuracy
Validation Top 5 Accuracy
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20

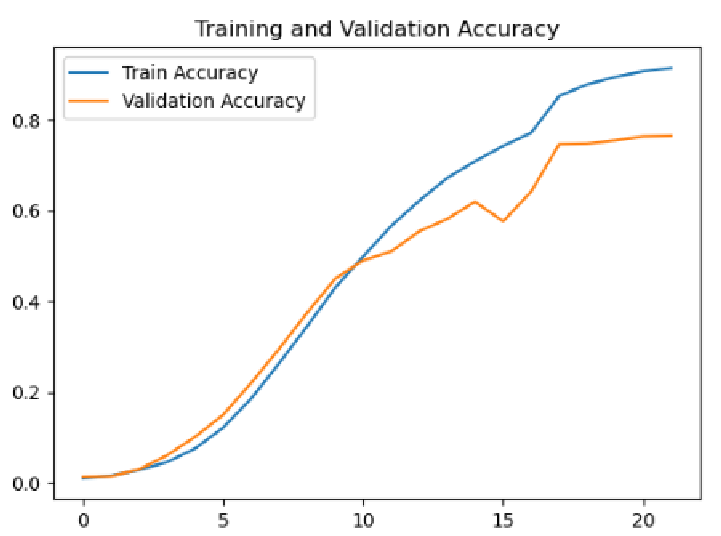
	
	
	

InceptionV3 Results

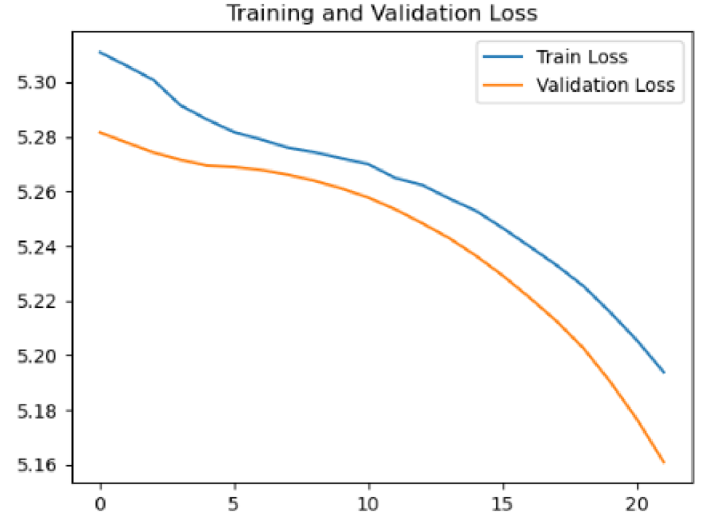
With Default hyper-parameters	With optimal hyper-parameters



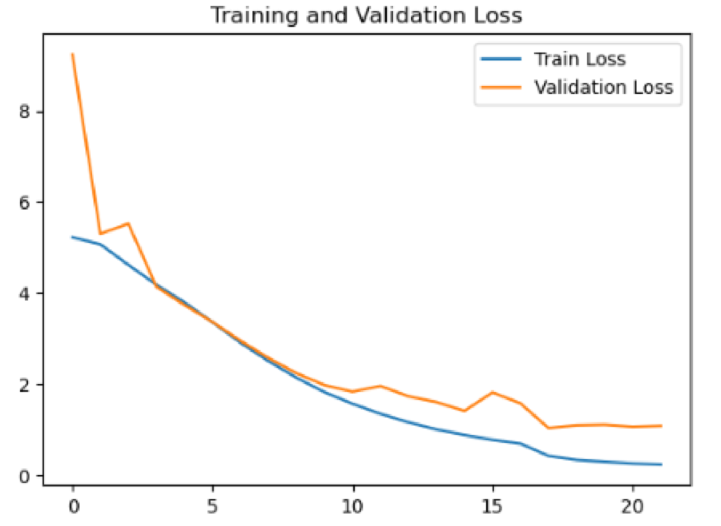
Training and Validation Accuracy



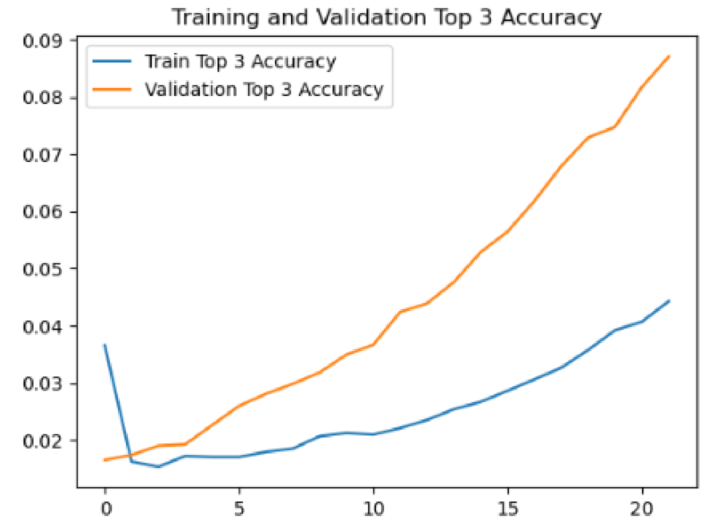
Training and Validation Accuracy



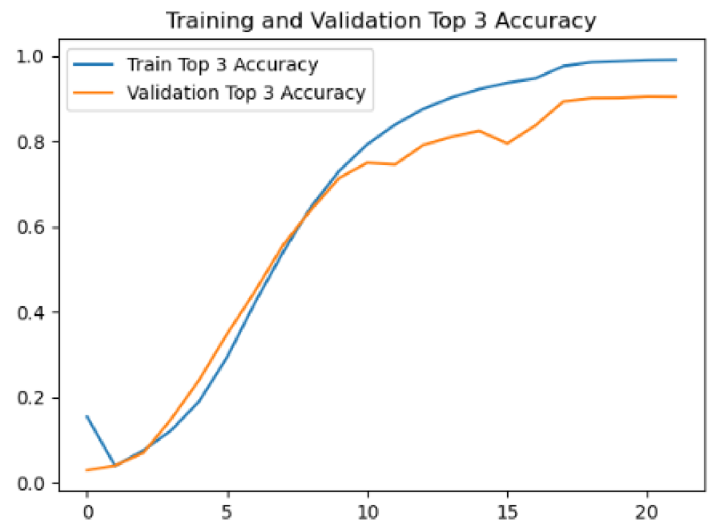
Training and Validation Loss



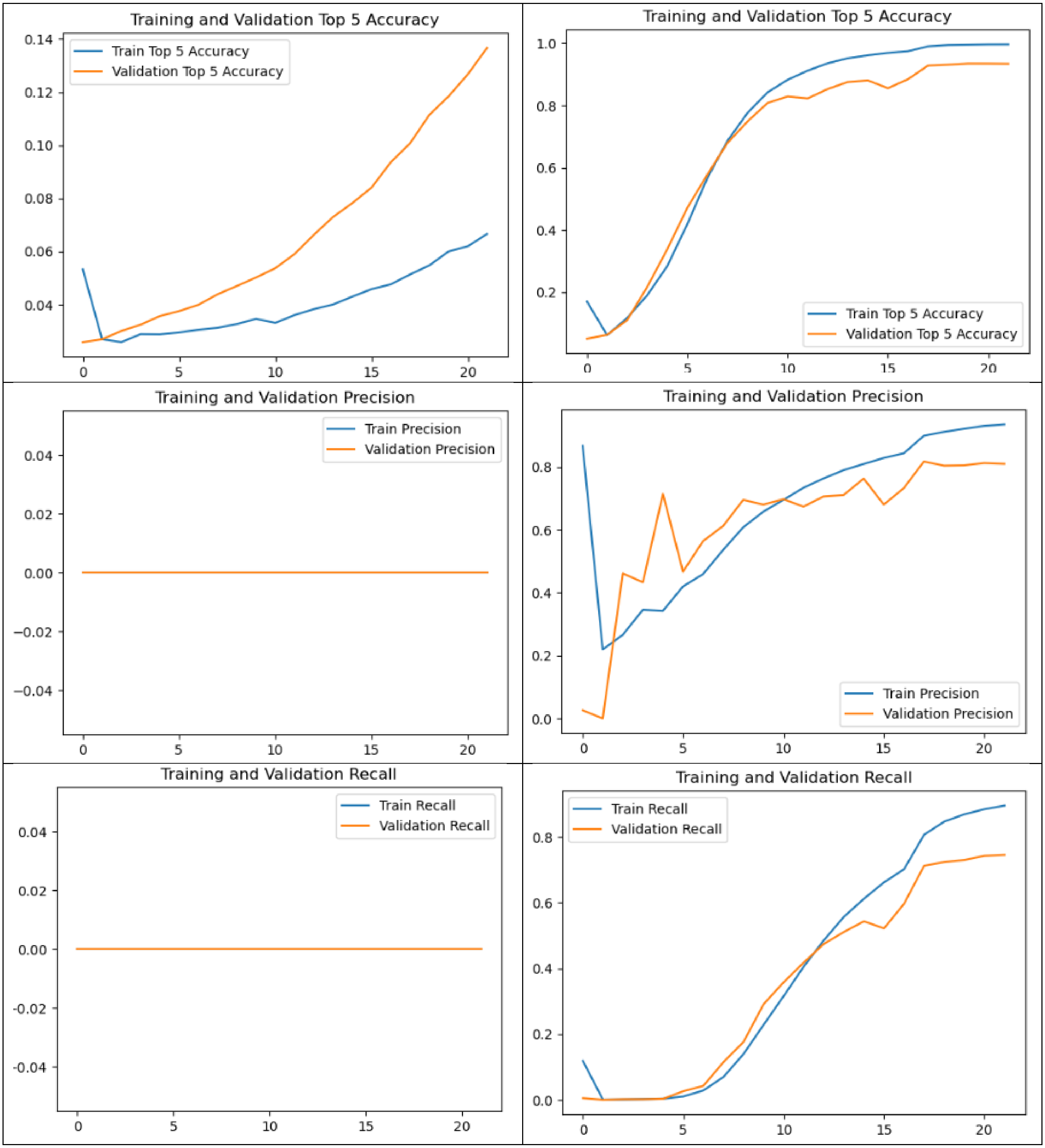
Training and Validation Loss

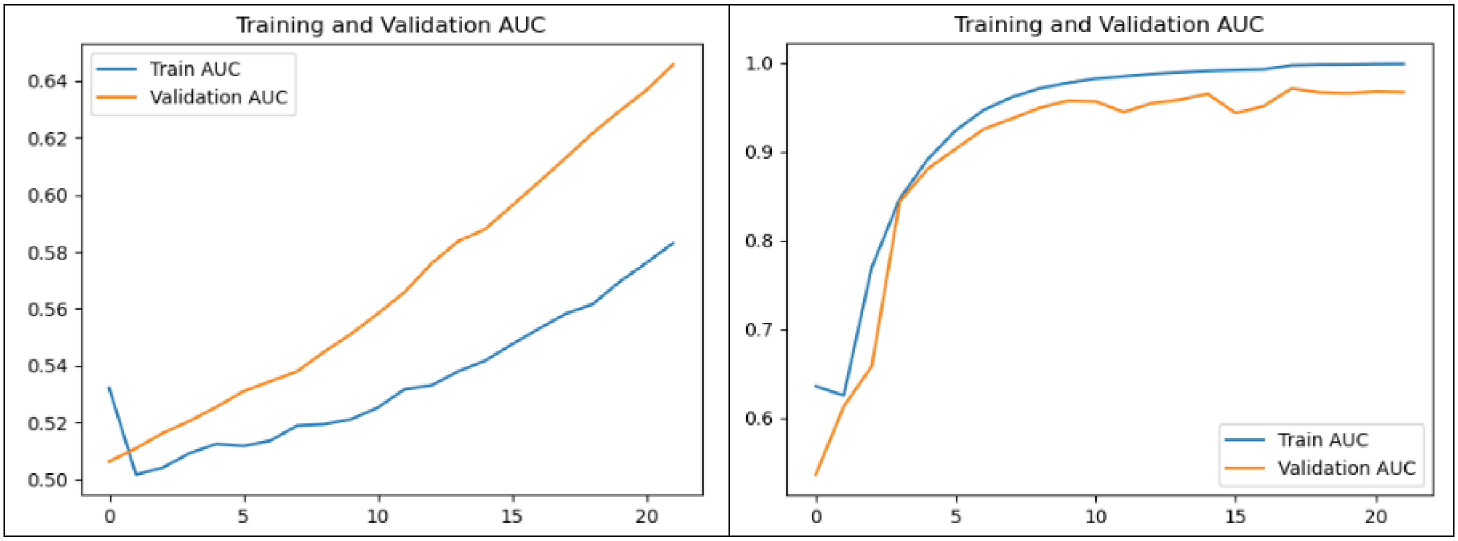


Training and Validation Top 3 Accuracy

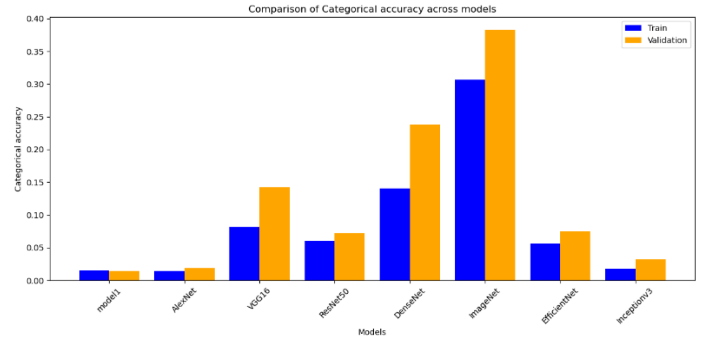
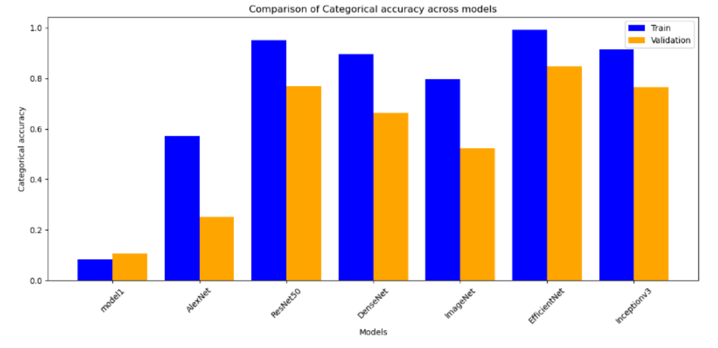
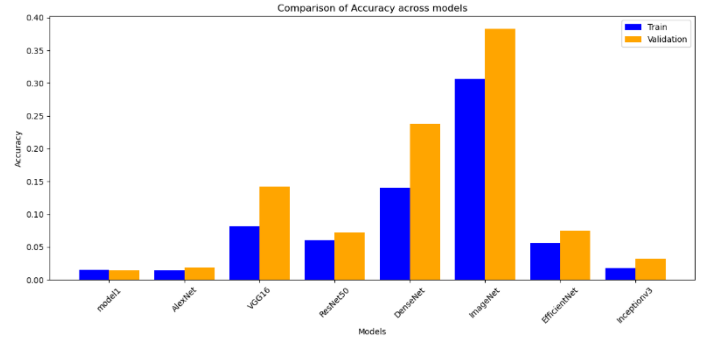
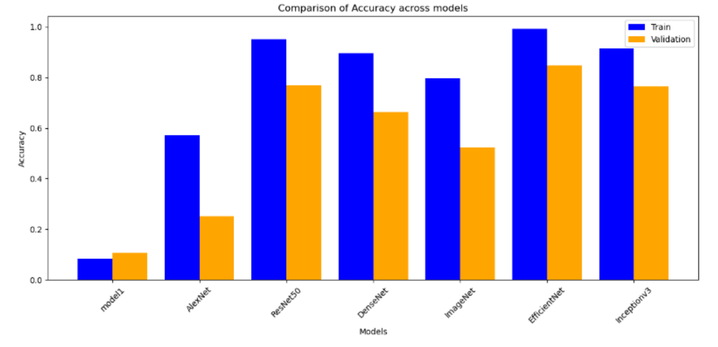
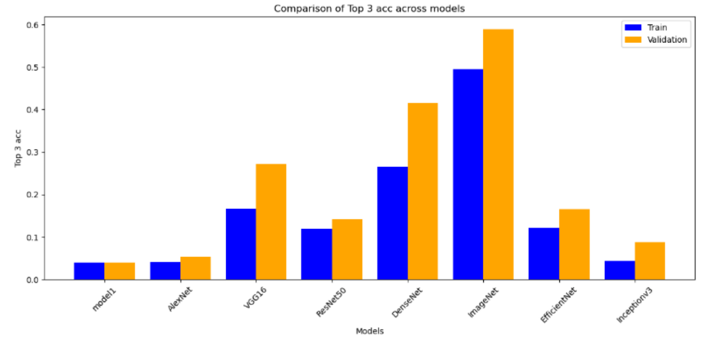
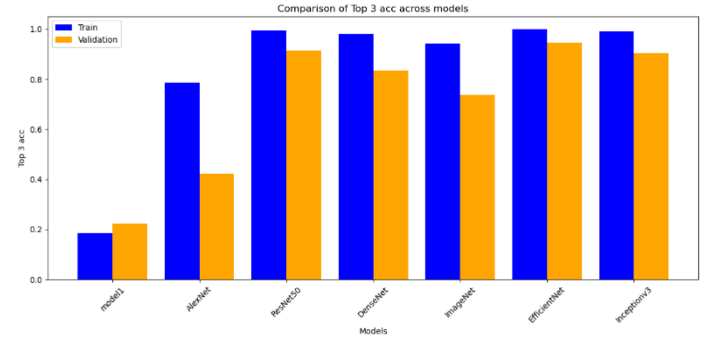


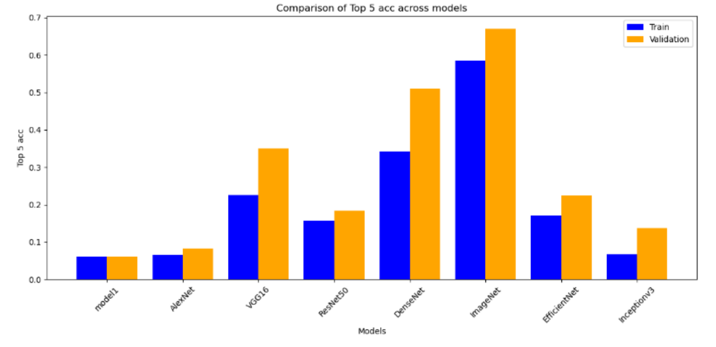
Training and Validation Top 3 Accuracy

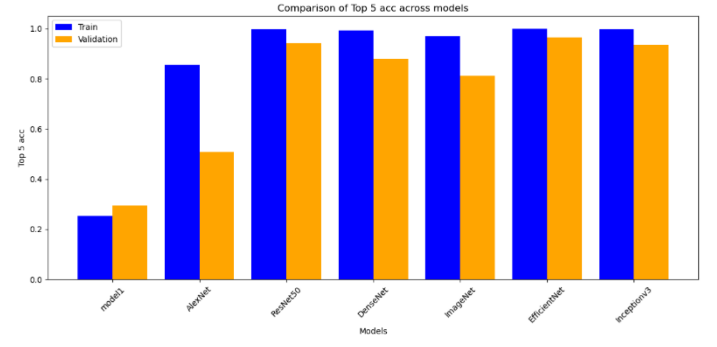

Training and Validation Top 5 Accuracy
Train Top 5 Accuracy
Validation Top 5 Accuracy
0.14
0.12
0.10
0.08
0.06
0.04
0
5
10
15
20
Training and Validation Top 5 Accuracy
1.0
0.8
0.6
0.4
0.2
Train Top 5 Accuracy
Validation Top 5 Accuracy
0
5
10
15
20
Training and Validation Precision
Train Precision
Validation Precision
0.04
0.02
0.00
−0.02
−0.04
0
5
10
15
20
Training and Validation Precision
0.8
0.6
0.4
0.2
0.0
Train Precision
Validation Precision
0
5
10
15
20
Training and Validation Recall
Train Recall
Validation Recall
0.04
0.02
0.00
-0.02
-0.04
0
5
10
15
20
Training and Validation Recall
Train Recall
Validation Recall
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20

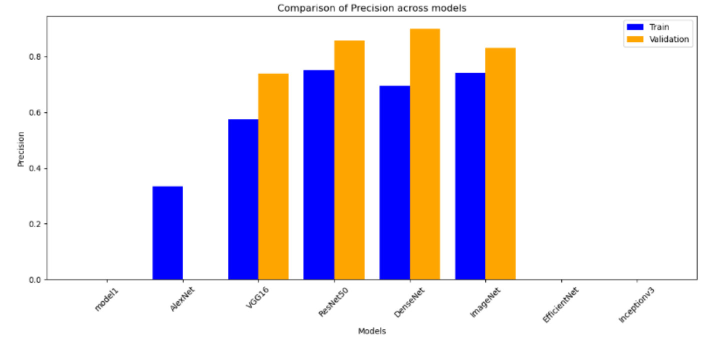


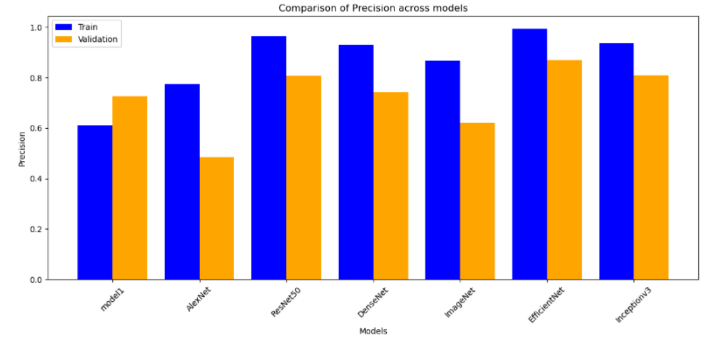
Comparison of models using default vs. optimal hyper-parameters

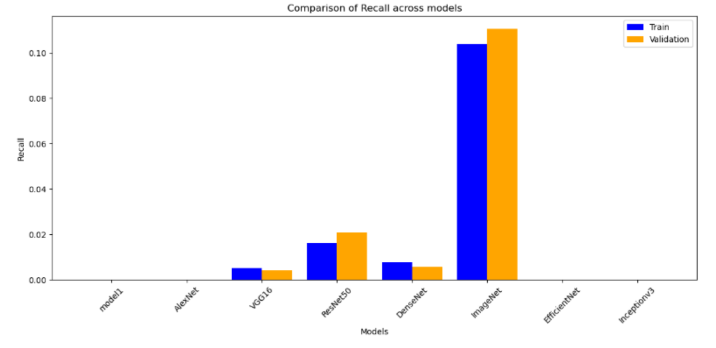
With default hyperparameters	With optimal hyper parameters
	
	
	

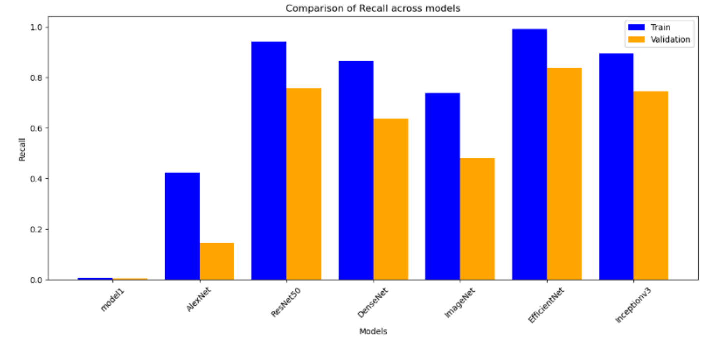


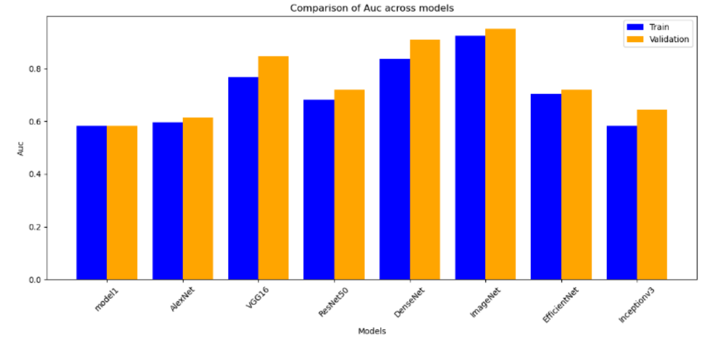


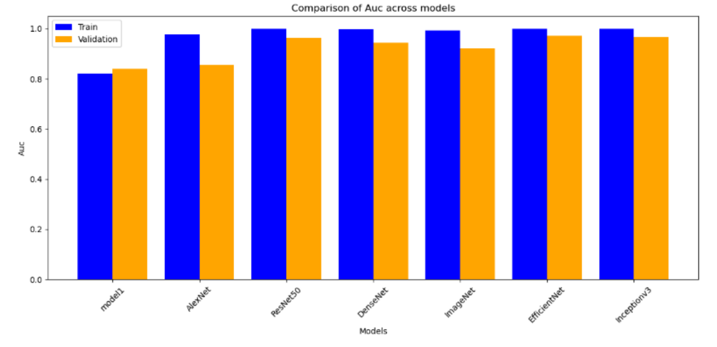












Conclusions

The results of the various models are highly influenced by:

- Data augmentation techniques used
- Boxing coordinates of the images
- Hyper parameters
- Order of transformations performed during the data augmentation

From the results obtained we conclude that

- Hyper parameter search to find optimal values yielded better results

- The best performing model was EfficientNetB0 followed by ResNET and InceptionV3 both of which had similar results.
- AlexNet faired quite poorly.

Next Steps

- 22 epochs do not seem to be sufficient to reach the plateau. Other similar projects arrived at optimal results with 30-35 epochs.
- The data augmentation techniques need to be retried with cutmix and mixup techniques as features of several different models of cars may overlap with each other.
- The order of transformations performed during the data augmentation can significantly change the results. Hence different orders of transformations with different probabilities need to be tried out.
- The functions can be further streamlined and broken down and more intermediate results can be captured and pickled to make the code more robust to failures.