DEEP LEARNING PROJECT

Project 1

Abstract

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

Amit Agarwal, Salim Touati, Selvalakshmi Ramagopalan, Srimallika Potluri

agarwalamit081@gmail.com, touati-salim1@hotmail.fr, selvalakshmi.r@gmail.com, potluri.mallika@gmail.com

Contents

Project Title

Classification of car model using the Stanford Cars dataset

Participants

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

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 version of NumPy library used had some compatibility issues with tensorflow-gpu, so the following code was added to make it work with TensorFlow

np.object = object

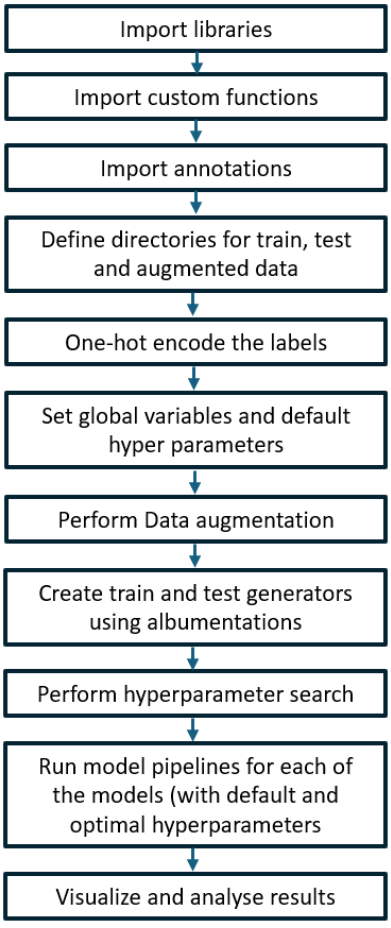
np.int = int
np.float = float
np.bool = bool

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.



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.

There was a specific issue with the given name for “Ram C-V Cargo Van Minivan 2012” and some work arounds were implemented to deal with this case.

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.

We had defined additional functions for data augmentation techniques named cutmix and mixup.

Cutmix

CutMix augments data by blending two images within the batch and combining their labels based on the proportion of each image's area.

generates a new sample by blending two random images from a batch. It randomly replaces a region of the target image with a region from a different image in the batch. The labels are also combined proportionally based on the area of each image in the blended result.

The steps are structured as follows:

Shuffle and Permute Images and Labels: The function randomly shuffles images and labels within the batch.

Random Patch Selection: A patch is randomly selected from the source image and replaced with a region from a shuffled image.

Label Mixing: Label proportions are adjusted based on the replaced patch’s area.

Return Blended Image and Adjusted Labels: The function outputs the mixed image and blended labels.

The function takes the following inputs:

- batch_size holds the number of images in the batch.
- indices generates a random permutation of batch indices for shuffling.
- shuffled_images and shuffled_labels are the randomized images and labels, used later for mixing patches.

Mixup

Mixup creates new data by linearly combining pairs of images and their corresponding labels from a batch, based on a random mixing coefficient.

This augmentation generates images that are a blend of two different classes, with the labels proportionally mixed, which helps the model learn smoother decision boundaries and prevents overfitting.

The function takes the following inputs:

- image: A batch of images.
- label: Corresponding labels for each image in the batch.
- alpha: A hyperparameter that controls the strength of the mixing. A larger alpha increases the likelihood of the mixed images resembling a more even blend of two images, whereas a smaller alpha will keep images closer to their original form.

Since the improvements were not so significant and it increased the time to train the dataset, we decided to disable it.

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.

Optimizers like Adam, SGD, and RMSprop each have different strengths:

- Adam is efficient but may converge too quickly in some cases.
- SGD with momentum can lead to better generalization but might require more tuning.
- RMSprop is useful for certain types of neural networks but can be sensitive to the decay parameter.

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 (later dropped)

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

We analysed the following Convolutional Neural Network models:

I. model1 (provided by Salim)
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 output metrics for the various models: accuracy, loss, precision, recall, and AUC on the train and test data.

Convolutional Neural Networks (CNN)

CNNs are a class of deep learning artificial neural networks that are most used for image recognition and classification tasks.

The models used are based on the Convolutional Neural Network (CNN) architecture containing one or more of the following layers which work together in a deep learning model to extract features, focus on important parts of the data, reduce dimensions, and make predictions.

They can automatically learn features from images in a hierarchical fashion (i.e. each layer builds upon what was learned by the previous layer).

CNNs can take advantage of the 2D structure of an image. Each convolutional layer in a CNN contains several neurons with weights which are assigned specific values during the training phase. These layers learn from small portions or patches of an input image, so they understand the local features better than Fully Connected Neural Networks, FNNs, which analyse entire images at once.

CNN architectures are less prone to overfitting due to their shared parameters across different locations in a single layer. This forces them to generalize well beyond any given training set data and ensures better performance on unseen samples compared with other models.

Since all layers share some sort of weight sharing scheme – meaning each filter output is convoluted with every pixel in its receptive field – these networks also require significantly fewer parameters.

Feature maps are a key concept in CNNs, which are generated by convolving filters over the input image, and each filter specializes in detecting specific features.

The feature maps serve as the input for subsequent layers, enabling the network to learn higher-level features and make accurate predictions.

Parameter sharing is another critical aspect of CNNs. It allows the network to detect similar patterns regardless of their location in the image, promoting spatial invariance. This enhances the network's robustness and generalization ability.

Convolution applies filters to the input image, sliding across and generating feature maps, while filters are small tensors with learnable weights, capturing specific patterns or features. Multiple filters detect various visual features simultaneously, enabling rich representation.

Padding preserves spatial dimensions by adding pixels around the input image. Stride determines the shift of the filter's position during convolution. Proper choices control output size, spatial information, and receptive fields. Zero padding is a technique used in CNNs to maintain the spatial dimensions of feature maps when applying convolutional operations. It

involves adding extra rows and columns of zeros around the input, which helps preserve the size and resolution of the feature maps during the convolution process and prevents information loss at the borders of the input data.

Each filter specializes in detecting specific patterns or features. Multiple filters capture diverse aspects of the input image simultaneously, while filter weights are learned through training, allowing adaptation to relevant patterns. The filter size in CNNs plays a crucial role in feature extraction, influencing the network's ability to detect and capture relevant patterns and structures in the input data.

Pooling layers are an integral part of Convolutional Neural Networks (CNNs) and play a crucial role in down sampling feature maps generated by convolutional layers while retaining important features and helps reduce spatial dimensions.

The down sampling process reduces the spatial dimensions of the feature maps, resulting in a compressed representation of the input. By reducing the spatial dimensions, pooling layers enhance computational efficiency and address the problem of overfitting by reducing the number of parameters in subsequent layers. Additionally, pooling layers help in capturing and retaining the most salient features while discarding less relevant or noisy information.

Activation functions are essential in Convolutional Neural Networks (CNNs) as they introduce non-linearity, enabling the network to learn complex feature relationships. Non-linearity allows CNNs to approximate complex functions and tackle tasks like image recognition and object detection.

They include ReLU, sigmoid, and tanh.

Description of the CNN layers

We briefly describe the layers present in the different CNN models used.

Conv2D Layer (Convolutional Layer)

A Conv2D layer applies convolution operations to input data (images). It uses a set of filters (kernels) to scan the input data and extract important features such as edges, textures, and patterns. This is the core of how convolutional neural networks (CNNs) learn to recognize different elements in images. It helps with feature extraction (e.g., detecting edges, colours, shapes).

Parameters:

filters: Number of filters (feature maps).

kernel_size: Size of the convolutional filters (e.g., 3x3).

strides: Determines the movement of the filters over the input.

activation: Activation function (e.g., ReLU).

Dropout Layer

Dropout randomly sets a fraction of input units to 0 at each update during training to prevent overfitting.

Dropout regularizes the network and improves generalization by reducing over-reliance on specific neurons.

Spatial Dropout Layer

Spatial Dropout is a regularization technique used to prevent overfitting. It randomly drops entire feature maps (instead of individual neurons) during training. This helps the model generalize better by ensuring that not all features are overly dependent on each other. It prevents overfitting and improve generalization by randomly setting feature maps to zero during training.

Parameters:

rate: Fraction of the feature maps to drop.

MaxPooling Layer

MaxPooling reduces the spatial dimensions (height and width) of the input data by selecting the maximum value from a defined window (e.g., 2x2) within the feature map. This reduces the computational load and helps the model focus on the most important features. It helps in dimensionality reduction while retaining key information.

Parameters:

pool_size: Size of the window (e.g., 2x2) over which to take the maximum.

strides: Steps the window moves during pooling.

Batch Normalization Layer

Batch Normalization normalizes the input to each layer so that the mean output is close to 0 and the standard deviation is close to 1 by normalizing the activations (or inputs to the next layer) within a mini batch during training.

After normalization, the layer applies two learned parameters: a scale (gamma) and a shift (beta) to allow the network to represent the identity transformation, which is important in retaining expressive power.

Normalization is performed separately for each feature (channel) over a mini-batch, hence the name "Batch" Normalization.

This improves training speed, stability, and performance by reducing internal covariate shift (changes in the input distribution during training) and reduces sensitivity to initialization and helps mitigate the vanishing or exploding gradient problems, thereby allowing the model to converge faster.

It also adds a bit of regularization by introducing noise in the form of mini-batch statistics, which can improve generalization.

Parameters:

axis: Axis along which normalization is applied (usually the channel axis).

momentum: Momentum for running mean and variance.

Flatten Layer

A Flatten layer reshapes the multi-dimensional output of the previous layers into a one-dimensional vector. This is often used when transitioning from convolutional layers to fully connected (dense) layers.

It converts 2D feature maps into a 1D vector before feeding into a dense layer.

Parameters: No trainable parameters, it just reshapes the input.

Dense Layer (Fully Connected Layer)

A Dense layer (fully connected layer) connects every neuron in the current layer to every neuron in the previous layer. It combines all features and passes them through an activation function, helping the network make final classifications or predictions. It performs high-level reasoning and make predictions.

Parameters:

units: Number of neurons.

activation: Activation function (e.g., ReLU for hidden layers, softmax for output layers).

GlobalAveragePooling2D Layer

It reduces each channel in the feature map to a single value by taking the average of all values in the spatial dimensions (height and width).

This layer is often used in classification tasks as it reduces the spatial dimensions of the feature maps while retaining information, providing a more compact representation for the fully connected layers.

Add Layer

It performs element-wise addition between two tensors (feature maps).

This layer is used for the identity shortcut connection in ResNet. The "Add" operation is a critical part of the residual connection, where the input is directly added to the output of the block. This helps in preserving information and allows the model to learn identity functions.

ZeroPadding2D Layer

It adds padding (extra border pixels) to the input data to control the spatial dimensions of the output after convolution operations.

Zero-padding ensures that the convolution preserves spatial dimensions, so no important edge information is lost, and helps the network handle edge pixels more effectively.

Rescaling Layer

The Rescaling layer is used to adjust the scale of input data, typically to ensure that pixel values are in a certain range.

The rescaling layer performs a simple operation where the input is multiplied by a scalar value (often a fraction, like (1/255) to rescale pixel values from their original range (typically 0–255 for image data) to a smaller range (like 0–1).

This is beneficial for models because smaller values help avoid issues related to large weight updates, which could lead to instability in training (like exploding gradients).

DepthwiseConv2D Layer

In a traditional Conv2D layer, filters convolve across all input channels, performing a full transformation at every spatial location.

The DepthwiseConv2D layer is a more efficient variant of the standard convolutional layer. It significantly reduces the computational cost, and the number of parameters compared to regular Conv2D layers, which is especially useful in lightweight models such as EfficientNet and MobileNet.

In DepthwiseConv2D, instead of applying a filter to all input channels simultaneously, each filter is applied to a single channel at a time. This means it performs spatial convolution independently for each input channel (depthwise convolution).

The depth wise convolutions are usually followed by pointwise convolutions (Conv2D with 1x1 kernel size), which help mix the information from different channels.

Benefit: This approach dramatically reduces computation and memory usage, making the DepthwiseConv2D layer ideal for mobile and edge devices without sacrificing much accuracy.

Reshape Layer

The Reshape layer is used to change the shape of the input tensor without altering its data. This layer is primarily used to prepare data for a specific operation or to transition between layers that expect different input formats.

It takes an input tensor with a particular shape and rearranges it into a new specified shape. This transformation can be used to flatten multi-dimensional tensors (like feature maps from convolutional layers) before passing them into a fully connected (Dense) layer.

Importantly, the number of elements in the input tensor and the reshaped tensor remains the same, meaning it only modifies the layout of the data.

Custom Layer

Function: A custom layer allows developers to implement specific functionality not available in standard layers. This layer can apply custom operations on the input based on the problem being solved.

It performs specialized operations defined by the user (e.g., attention, SE mechanisms).

Parameters: Defined by the custom layer implementation (e.g., block types, number of channels).

SE Block (Squeeze-and-Excitation Block)

Function: The SE Block enhances important feature channels and suppresses less useful ones. It works by "squeezing" the spatial dimensions to generate global descriptors and "exciting" (scaling) the feature maps based on these descriptors.

It helps improve the representational power of the model by emphasizing important feature channels.

Steps:

Squeeze: Global average pooling to generate a summary of each channel.

Excitation: Scaling each channel by a weight learned from the "squeeze" step.

se_block.py defines a class SEBlock that takes reduction as a parameter which controls the amount of feature map reduction in the first dense layer and contains three methods: build, call, and get_config.

1. The build method initializes layers for performing the squeeze-and-excitation operation.

Global Average Pooling reduces each channel to a single average value, creating a global context vector.

Two dense layers are used for recalibration:

The first dense layer reduces the number of channels by reduction, applying ReLU activation.

The second dense layer restores the original channel count with sigmoid activation, creating channel-wise importance weights.

Reshape and multiply reshapes the channel importance weights to match the original input shape and multiplies them by the input tensor to apply channel recalibration.

2. The call method executes the SE operation on the input.
3. The get_config method returns the configuration for the block, which is useful for saving or loading the model.

Attention Block Layer

Function: An Attention Block helps the model focus on the most relevant parts of the input data. It assigns weights to different parts of the input so that more attention is given to the important features while processing.

Purpose: Improve model accuracy by focusing on the most relevant parts of the input.

Parameters: Typically internal to the block, attention mechanisms compute scores (weights) for each feature.

attention_block.py defines the AttentionBlock class and contains the build, call, and get_config methods.

1. The build method initializes components needed for attention computation:

The Global Average Pooling layer creates a summary for each channel by averaging across spatial dimensions.

The Dense Layers applies a bottleneck (channel reduction) and expansion strategy:

The first dense layer reduces the channel count, with ReLU activation while the second dense layer restores the original number of channels, with sigmoid activation.

The computed attention weights are reshaped and multiplied by the original input to enhance key areas of the feature map.

2. The call method executes the attention mechanism.
3. The get_config method returns the configuration settings for potential model reusability or transfer learning.

Activation (ReLU)

Function: The ReLU (Rectified Linear Unit) activation function sets all negative values in the output to zero while keeping positive values unchanged. Mathematically: f(x) = max(0, x).

Purpose: ReLU introduces non-linearity to the network, which is essential for learning complex patterns and features.

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

Below is a description of some of the hyper parameters used and optimized.

learning_rate

It controls how much the model weights are adjusted in response to the estimated error at each step. A lower learning rate makes the model learn more slowly, but it can improve the stability of training.

The learning rate directly influences convergence:

- High values can speed up training but risk overshooting the optimal weights.
- Low values ensure stability and accuracy but slow down the training process.

Typical Values: Often ranges from 1e-3 to 1e-6 depending on the model and dataset.

Example Usage: For fine-tuning or sensitive models, a smaller learning rate like 1e-6 helps prevent the model from overshooting minima.

weight_decay

Weight decay, also known as L2 regularization, penalizes large weight values by adding a small penalty term proportional to the square of the weights in the loss function.

By penalizing large weights, weight decay encourages the model to avoid complexity and thus helps in reducing overfitting.

The strength of this regularization effect is determined by the weight_decay value, with larger values adding stronger regularization.

Typical Value: Values are usually quite small, like 1e-4 to 1e-6, balancing between regularization and model flexibility.

dropout

Dropout is a regularization technique to prevent overfitting by randomly setting a fraction of the input units to zero at each training step.

Dropout randomly "drops out" a percentage of neurons, meaning they are ignored during the forward and backward passes.

This discourages the model from relying on any specific subset of neurons and instead forces it to learn more robust features.

Typical Values: Between 0.1 to 0.5. A value of 0.5 means that half of the neurons are dropped randomly.

Usage: With dropout=0.5, the model will ignore 50% of the neurons during each training step, which encourages the network to learn more robust features that aren’t reliant on any specific subset of neurons.

beta_1 and beta_2

These are hyperparameters for the Adam optimizer, controlling the decay rates of running averages for the first and second moment estimates (the mean and uncentered variance of gradients, respectively).

beta_1 controls the exponential decay rate for the first moment estimate (the moving average of gradients). It affects how much past gradients impact the current gradient.

beta_2 controls the exponential decay rate for the second moment estimate (the moving average of squared gradients), affecting how much past gradients' magnitudes influence the current step size.

These parameters help Adam optimize the learning rate adjustments by keeping track of past gradients and squared gradients, leading to a more stable and efficient optimization process.

beta_1: Commonly set to 0.9, this parameter allows the optimizer to maintain some "memory" of past gradients, reducing oscillations.

beta_2: Often set to 0.999, beta_2 smooths out the variance in gradient magnitude, making updates more stable and leading to efficient convergence.

Typical Value: The values 0.9 and 0.999 work well for most cases but may be adjusted slightly depending on the model and dataset characteristics.

decay

It controls the learning rate decay over time, which helps reduce the learning rate by reducing the step size as training progresses. This gradual decrease in learning rate helps the model converge better by making smaller updates as it approaches an optimal solution.

By gradually reducing the learning rate, the model makes smaller adjustments as it approaches the minimum, preventing overshooting.

This helps ensure the model does not oscillate near the optimal point and improves stability in training.

Typical Values: A small value like 1e-6 is often used.

Usage: Helps fine-tune learning rate so that the model doesn’t overshoot or diverge as it nears convergence.

momentum

A parameter specific to the SGD optimizer that helps accelerate gradient descent by adding a fraction of the previous gradient update to the current one. It is designed to accelerate convergence in relevant directions and reduce oscillations.

Momentum accumulates a fraction of the past gradients, which adds inertia to the updates, allowing the model to speed up along the direction of the gradient.

By considering past gradients, it helps the model overcome shallow local minima and make smoother transitions.

Typical Values: Usually set around 0.9. It is not used with optimizers like Adam, which inherently manage adaptive learning rates.

Usage: When using momentum (e.g., with SGD), the optimizer remembers the previous gradients and "speeds up" along the relevant direction, allowing the model to converge faster and avoid local minima.

rho

This is a parameter specific to the RMSprop optimizer, which controls the decay rate of moving average of squared gradients , helping RMSprop adjust the learning rate for each parameter.

Like beta_2 in Adam, rho impacts how strongly past squared gradients affect current parameter updates.

By adapting the learning rate individually per parameter, RMSprop with rho prevents large updates for parameters with frequent large gradients.

Typical Values: Usually around 0.9.

Usage: By controlling how fast past gradients decay, rho helps RMSprop adaptively adjust the learning rate for each parameter, especially useful in training deep networks.

optimizer_type

It specifies the type of optimizer to use during training.

Typical Options:

Adam: Combines the benefits of momentum and adaptive learning rates. It is efficient and it adjusts learning rates per parameter using beta_1 and beta_2.

SGD (Stochastic Gradient Descent): A straightforward optimizer with optional momentum. Works well for simpler models or when more control over the optimization process is needed.

RMSprop: Especially useful for recurrent neural networks or models with non-stationary objectives, as it adjusts the learning rate for each parameter based on the recent gradients. It Adapts the learning rate for each parameter, preventing the learning rate from being too large for frequently updated weights.

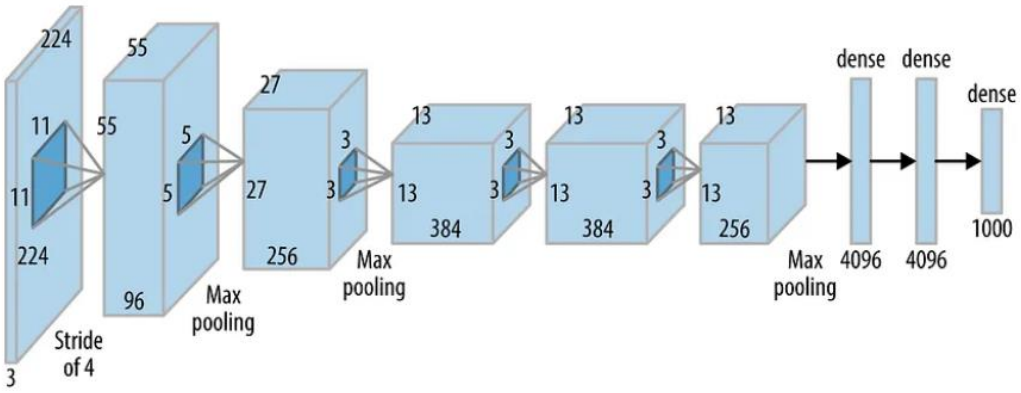
Overview of the Models

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, takes input_shape, num_classes, hyperparams, and x_trainable as parameters where:

- input_shape: Specifies the input data shape.
- num_classes: Defines the number of output classes for classification.
- hyperparams: Contains hyperparameters for configuring the model, including learning_rate, dropout, and others.
- x_trainable: Indicates whether all layers should be trainable, or some should be frozen.

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.

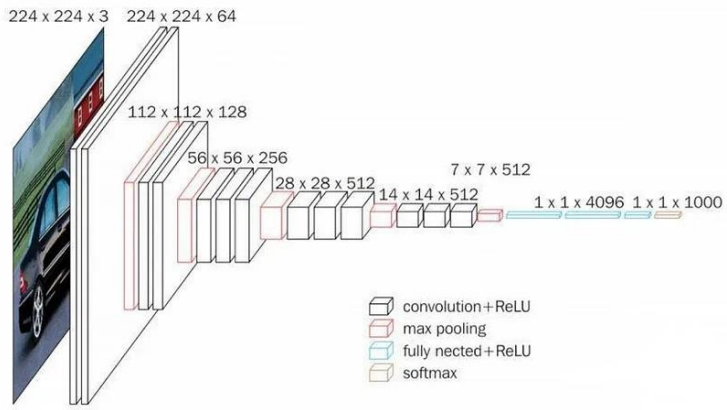
It contains layers for input (1), Conv2D (13), Spatial Dropout (1), MaxPooling2D (5), Attention Block (1), SE Block (1), Custom (2), flatten (1), dense (2) and dropout(1) layers.

In VGG16 there are thirteen convolutional layers, five Max Pooling layers, and three Dense layers which sum up to 21 layers, but it has only sixteen weight layers i.e., learnable parameters layer.

Instead of having many hyper-parameters they focused on having convolution layers of 3x3 filter with stride 1 and always used the same padding and MaxPool layer of 2x2 filter of stride 2.

Conv-1 Layer has 64 number of filters, Conv-2 has 128 filters, Conv-3 has 256 filters, Conv 4 and Conv 5 has 512 filters.

Three Fully Connected (FC) layers follow a stack of convolutional layers: the first two have 4096 channels each, the third performs 1000-way ImageNet Large Scale Visual Recognition Challenge (ILSVRC) classification and thus contains 1000 channels (one for each class).



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

The create_vgg16_model.py includes the main function for creating a VGG16 model with dynamic hyperparameters for image classification.

The function, create_vgg16_model, takes input_shape, num_classes, and hyperparams as inputs where:

- input_shape: The shape of input images (e.g., (224, 224, 3)).
- num_classes: The number of output classes for classification.
- hyperparams: A dictionary of hyperparameters used for model configuration, including the learning rate, dropout, optimizer type, etc.

Values like learning_rate, dropout, and optimizer_type are extracted from hyperparams with defaults provided if these keys are missing.

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 function, finetune_vgg16_model, is called to further modify and configure the VGG16 model, using the extracted transfer layer, specified dropout, fully connected layers, and other parameters.

It adds dropout, attention, and SE blocks, as well as fully connected layers and returns the total number of layers and the number of frozen layers for verification.

The optimizer is dynamically selected based on optimizer_type.

The model is compiled with the chosen optimizer, categorical_crossentropy loss (standard for multi-class classification), and a set of metrics for evaluation.

The function takes the following inputs:

- base_model: The pre-trained VGG16 model loaded in create_vgg16_model.
- transfer_layer: The selected layer (block5_pool) where custom layers will be added.
- x_trainable: Specifies the layers to freeze or train in the base model (all, none, or partial).
- dropout: The dropout rate for regularization.
- fc_layers: A list of units for the fully connected (dense) layers, such as [512].
- num_classes: Number of output classes for the final classification layer.
- new_weights: Optional path for loading pre-trained weights into the fine-tuned model.

The layers are controlled based on x_trainable value:

- all: All layers are set as trainable.
- none: All layers are frozen.
- partial: Freezes all layers except the last four (or a specific number if set by x_trainable).

CustomLayer provides additional flexibility by allowing an attention or SE block to be added.

Custom Blocks: Attention and SE blocks are added to enhance the model's capability to focus on important features.

SpatialDropout2D: Adds spatial dropout to prevent overfitting by randomly dropping feature maps.

Attention Block and SE Block: Custom blocks to focus on relevant features and channels.

The feature maps are flattened and dense layers (Fully Connected Layers) with dropout are added to regularize the fully connected layers.

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.

These residual (skip) connections enable the network to learn much deeper architectures without suffering from the problem of vanishing gradients, a problem that occurs when training deep neural networks, where the gradients of the parameters in the deeper layers become very small, making it difficult for those layers to learn and improve.

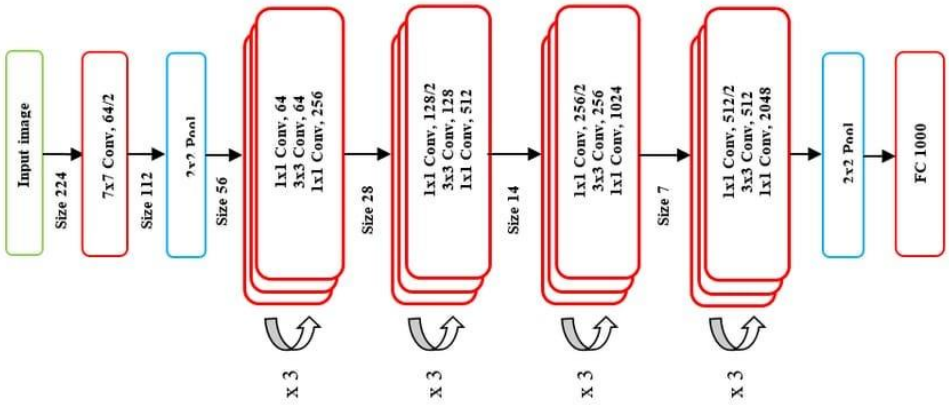
Skip connections address this problem by allowing the information to flow directly from the input to the output of the network, bypassing one or more layers. This allows the network to learn residual functions that map the input to the desired output, rather than having to learn the entire mapping from scratch.

The architecture of ResNet50 is divided into four main parts: the convolutional layers, the identity block, the convolutional block, and the fully connected layers.

The convolutional layers are responsible for extracting features from the input image, while the identity block and convolutional block are responsible for processing and transforming these features.

Finally, the fully connected layers are used to make the final classification.

The identity block and convolutional block are the key building blocks of ResNet50. The identity block is a simple block that passes the input through a series of convolutional layers and adds the input back to the output. This allows the network to learn residual functions that map the input to the desired output. The convolutional block is like the identity block, but with the addition of a 1x1 convolutional layer that is used to reduce the number of filters before the 3x3 convolutional layer.



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.

In create_resnet50_model.py, the function, create_resnet50_model function, loads a ResNet50 model, customizes it, and compiles it with selected hyperparameters.
It takes the following inputs:

- input_shape: Shape of the input images (e.g., (224, 224, 3)).
- num_classes: Number of output classes.
- hyperparams: Dictionary of hyperparameters used to customize training, such as learning_rate, dropout, and optimizer_type.

We load ResNet50 without the top (classification) layers using include_top=False.

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. The custom layers will be attached here.

The model is compiled with the chosen optimizer, categorical cross-entropy loss (standard for multi-class classification), and evaluation metrics.

The function, finetune_resnet50_model, adds custom layers, sets layer training states, and adds additional regularization with dropout.
It adds attention blocks, SE blocks, and fully connected layers for classification.
The function takes the following parameters as input:

- base_model: The pre-trained ResNet50 model without the top layers.
- transfer_layer: The chosen transfer layer (conv5_block3_out) where custom layers will be appended.
- num_classes: The number of output classes.
- hyperparams: Dictionary of hyperparameters, such as dropout, fc_layers, and x_trainable.

The layer freezing controls which layers to train on.

- all: All layers are trainable.
- none: All layers are frozen, i.e., they are not updated during training.

- partial: Only the last four layers are unfrozen, allowing fine-tuning of high-level features.

The following layers have been added:

- Spatial Dropout: Adds SpatialDropout2D, a regularization layer that randomly drops feature maps to prevent overfitting.
- Attention Block and SE Block: Introduces an attention mechanism (AttentionBlock) and channel recalibration (SEBlock), to improve feature focusing.
- Custom Layer: Allows flexibility in adding additional SE and attention blocks via CustomLayer.
- Global Average Pooling: Averages each feature map to a single value, reducing spatial dimensions before dense 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).

Dense Convolutional Network, a Convolutional Neural Networks (CNN) architecture, improves information flow and gradient propagation. It addresses challenges such as feature reuse, vanishing gradients, and parameter efficiency.

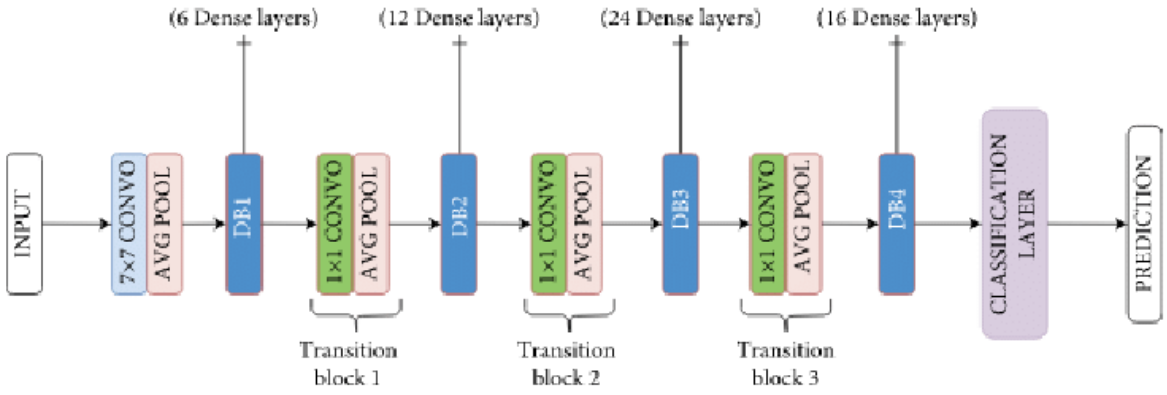
It helps gradients flow more smoothly through the network by connecting each layer to every other layer within a dense block and It allows each layer to use features from all preceding layers, enhancing feature reuse.

Unlike traditional CNN architectures where each layer is connected only to subsequent layers, DenseNet establishes direct connections between all layers within a block. This results in a network with L(L+1)/2 direct connections for L layers.

This dense connectivity enables each layer to receive feature maps from all preceding layers as inputs, fostering extensive information flow throughout the network.

By concatenating features from all preceding layers, DenseNet encourages feature reuse, reducing redundancy and improving efficiency.

DenseNet is parameter efficient, and it eliminates the need to relearn redundant features, resulting in fewer parameters compared to traditional networks.



In create_densenet_model.py, the create_densenet_model function loads a DenseNet121 model, modifies it, and compiles it based on the input hyperparameters.

It takes the following inputs:

- input_shape: The shape of the input data (e.g., (224, 224, 3) for color images).
- num_classes: The number of output classes.
- hyperparams: Dictionary of hyperparameters such as learning_rate, dropout, and optimizer_type.
- x_trainable: Specifies whether all, none, or a portion of the model layers are trainable.

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.

The x_trainable parameter defines which layers are trainable and it helps the model retain general features learned on ImageNet while allowing customization of higher layers for the specific task.

- all: All layers are trainable.
- partial: All layers except the last four are frozen.

Any other value freezes all layers.

The function, finetune_densenet_model, takes in the customized base model and hyperparameters where additional layers like attention, SE blocks, and fully connected layers are added.
It takes the following inputs:

- base_model: The base DenseNet121 model without classification layers.
- num_classes: Number of output classes for the final classification layer.
- fc_layers: List defining the number of units in each fully connected layer. By default, it has one layer with 512 units.
- dropout: Dropout rate for fully connected layers to prevent overfitting.

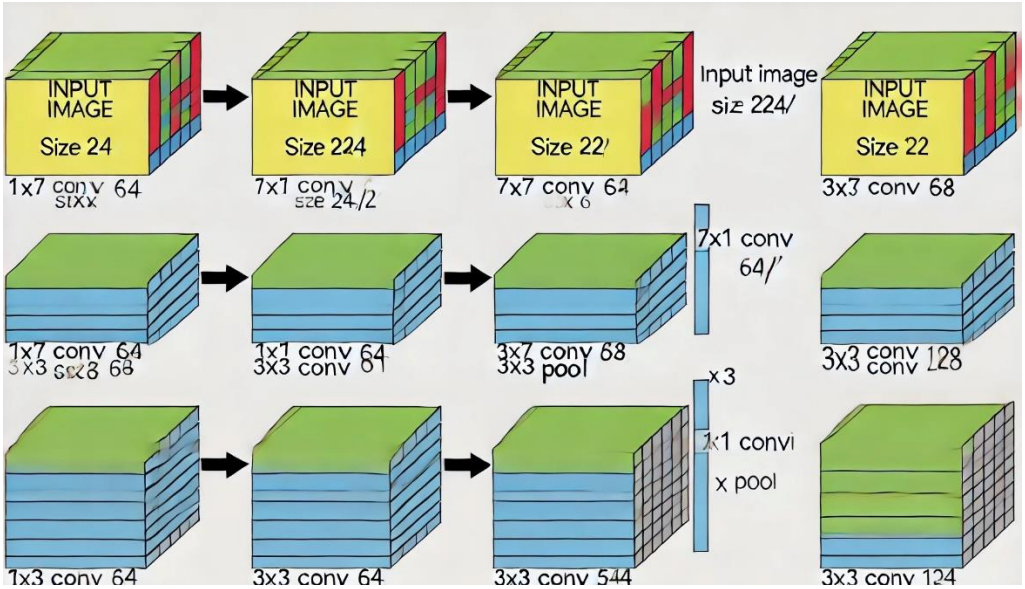
Custom layers are added to the base model:
Spatial Dropout: Adds SpatialDropout2D(0.2), which drops entire feature maps, preventing over-reliance on certain features and reducing overfitting.
Attention and SE Blocks: Introduces attention (AttentionBlock) and SE blocks (SEBlock), which allow the model to focus on relevant features.
Flattening: Converts the 2D feature maps into a 1D vector for fully connected layers.

The dynamic Fully Connected layers loops through fc_layers to add fully connected layers with Dropout after each layer.
Dropout is set dynamically using the dropout hyperparameter, reduces overfitting by randomly dropping neurons during training.

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.

In create_imagenet_model.py, the function, create_imagenet_model, loads the ResNet50 model (pre-trained on ImageNet) as the base and modifies it according to the provided hyperparameters.

The ResNet50 model is loaded without its top classification layer by setting include_top=False.

The model is initialized with ImageNet weights, making it suitable for transfer learning tasks.

It takes the following inputs:

- input_shape: Specifies the shape of the input images.
- num_classes: Defines the number of output classes for the model's final classification layer.
- hyperparams: Dictionary containing model and optimizer parameters.

The ResNet50 base model is loaded without the classification head, and its layers' trainability is configured according to the x_trainable flag

Based on the x_trainable flag within hyperparams the trainability of the layers can be configured:

- all: All layers are set to trainable, allowing fine-tuning across the entire model.
- partial: All layers except the last 4 layers are frozen.

Any other value implies freezing all layers, relying purely on the learned ImageNet features.

The function, finetune_imagenet_model, adds custom layers to the ResNet50 base model, selects an optimizer, and compiles the fine-tuned model.

It takes the following parameters as inputs:

- input_shape: Specifies the shape of the input data.
- num_classes: Defines the number of output classes.
- hyperparams: A dictionary containing model configuration options, including optimizer type and dropout rate.

Custom layers have been added and they include:

Spatial Dropout: A SpatialDropout2D layer with a dropout rate of 0.2 is added immediately after the base model's output, reducing overfitting by dropping entire feature maps.

Attention Block: Focuses on important spatial features within each feature map, enhancing the model's ability to recognize relevant patterns.

SE Block: Squeeze-and-Excitation Block recalibrates the importance of each feature map channel.

Custom Layer: Includes an additional CustomLayer with blocks for both attention and SE.

Flattening and Fully Connected Layers:
A Flatten layer flattens the output of the convolutional blocks, converting 2D spatial features into a 1D feature vector for classification.

Fully connected (dense) layers are added dynamically based on the fc_layers hyperparameter.
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.

The model is compiled with categorical cross-entropy loss (for multi-class classification) and a list of metrics defined in metrics.

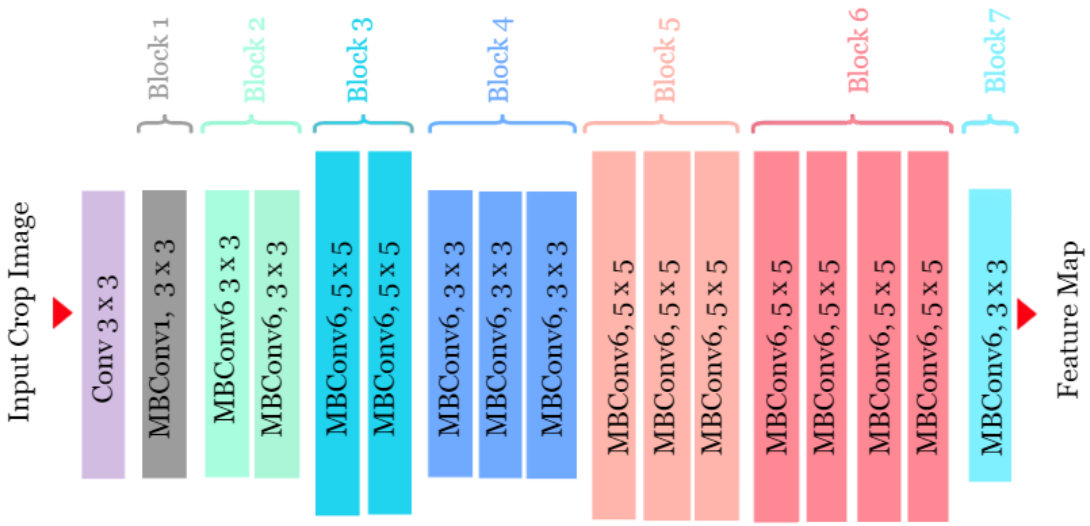
EfficientNetB0

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).

EfficientNet's layers are based on a compound scaling method that uniformly scales the depth, width, and resolution of the network, with the stem layer being the initial part of the network that performs initial convolutions to process the input image before deeper layers all whilst using fewer parameters.
Stem refers to the initial set of layers that process the input data before it passes through the main body of the network.
In the case of EfficientNetB0, the stem consists of the following components:

- Conv2D: 3x3 convolution with stride 2
- Batch Normalization
- Swish activation (a type of smooth, non-linear activation function)

Resolutions not divisible by 8, 16, etc. cause zero-padding near boundaries of some layers which wastes computational resources. This applies to smaller variants of the model, hence the input resolution for B0 and B1 are chosen as 224 and 240.
The building blocks of EfficientNet demands channel size to be multiples of 8.
The Top Layer in the EfficientNetB0 architecture is the final stage of the network, where the processed features are transformed into outputs suitable for specific tasks like classification.
This layer plays a crucial role in determining the network's output based on the learned features.



In create_efficientnet_model.py, the function, create_efficientnet_model, initializes the EfficientNetB0 model, configures it based on the specified hyperparameters, and compiles it with a suitable optimizer.

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

It takes the following parameters as inputs:

- input_shape: Defines the dimensions of the input images (e.g., (224, 224, 3))
- num_classes: Specifies the number of output classes.
- hyperparams: A dictionary of hyperparameters to configure the model's behaviour and training process

The x_trainable parameter defines which layers are trainable and it helps the model retain general features learned on ImageNet while allowing customization of higher layers for the specific task.

- all: All layers are trainable
- partial: All layers except the last four are frozen

Any other value freezes all layers.

In finetune_efficientnet_model.py, the function, finetune_efficientnet_model, modifies the EfficientNetB0 model further by adding custom layers, implementing attention mechanisms, and defining the output layers.

It takes the following parameters as inputs:

- base_model: The pre-trained EfficientNetB0 model.
- num_classes: Number of classes for classification.
- learning_rate, weight_decay: Configures optimizer parameters.
- x_trainable: Specifies layer trainability

Additional custom layers include:

Spatial Dropout: A SpatialDropout2D layer is applied to the convolutional output to reduce overfitting by dropping entire feature maps randomly.

Attention Block: Dynamically weighs feature importance across the spatial dimensions.

SE Block: The Squeeze-and-Excitation block recalibrates feature channels, helping the model focus on the most critical feature maps.

Custom Layers: CustomLayer is used to implement additional attention and SE mechanisms, enhancing the model's ability to prioritize essential information.

Flatten and Fully Connected Layers:

The feature maps are flattened using a Flatten layer, converting them into a 1D vector suitable for dense layers.

A fully connected (Dense) layer with ReLU activation and 512 units is added, followed by a dropout layer to further regularize the model.

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.

The model is compiled with categorical cross-entropy loss and a list of metrics from the custom metrics function, preparing it for training.

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).

It improves on Inception v2 by using factorized convolutions and improves on Inception v1 by adding batch normalization and removing dropout and local response normalization.

It makes use of label smoothing, factorized 7 x 7 convolutions, and the use of an auxiliary classifier to propagate label information lower down the network (along with the use of batch normalization for layers in the side head).

It also uses a form of dimension-reduction by concatenating the output from a convolutional layer and a pooling layer and the lowest auxiliary classifier is removed during training.

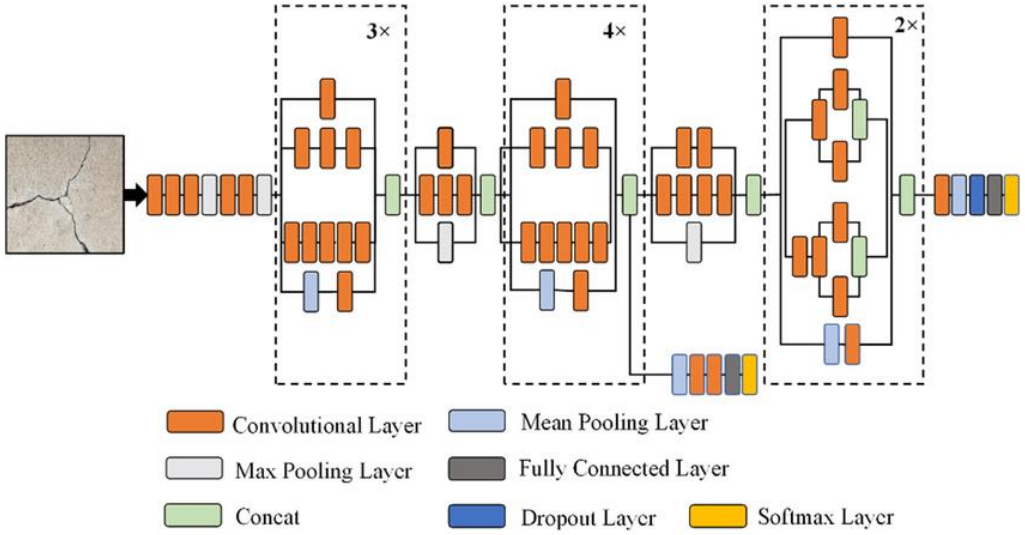
Factorized Convolutions help reduce the computational efficiency as it reduces the number of parameters involved in a network. It also keeps a check on the network efficiency.

Replacing bigger convolutions with smaller convolutions leads to faster training. Say a 5 × 5 filter has 25 parameters; two 3 × 3 filters replacing a 5 × 5 convolution has only 18 (3*3 + 3*3) parameters instead.

Asymmetric convolutions: A 3 × 3 convolution could be replaced by a 1 × 3 convolution followed by a 3 × 1 convolution. If a 3 × 3 convolution is replaced by a 2 × 2 convolution, the number of parameters would be slightly higher than the asymmetric convolution proposed.

Auxiliary classifier: an auxiliary classifier is a small CNN inserted between layers during training, and the loss incurred is added to the main network loss. In Inception v3 an auxiliary classifier acts as a regularizer.

Grid size reduction: Grid size reduction is usually done by pooling operations.



In create_inceptionv3_model.py, the function, create_inceptionv3_model, takes the following inputs:
input_shape: Specifies the input dimensions for the model (e.g., (224, 224, 3) for 224x224 RGB images).
num_classes: The number of output classes for the classification task.
hyperparams: A dictionary that contains various hyperparameters to control the model's configuration.

The base InceptionV3 is loaded without its top classification layers, using include_top=False. This allows custom layers to be added on top.
transfer_layer points to the 'mixed10' layer, the last feature extraction layer in InceptionV3, where custom modifications begin.

The custom function, finetune_inceptionv3, applies custom layers and configurations to the InceptionV3 model.
It also allows selective layer freezing, custom layers, and additional dropout for regularization.
It takes the base model, transfer layer, and hyperparameters as inputs. Arguments to the function include:

- base_model: Pre-trained InceptionV3 model up to the transfer layer.
- transfer_layer: Specifies the layer from which fine-tuning starts.
- x_trainable: Defines which layers should be trainable.
- dropout: Dropout rate for fully connected layers.
- fc_layers: List of integers representing the number of neurons in fully connected layers.
- num_classes: Number of classes in the classification task.
- new_weights: Optional path for loading specific weights.

x_trainable determines how many layers to freeze:

- all: All layers are trainable.
- none: All layers are frozen.
- partial: All but the last 4 layers are frozen.
- Integer: Custom freezing, freezing all but the specified number of layers.

Custom layers include:
Spatial Dropout: Applied to convolutional layers to improve regularization.
Attention and SE Blocks: Custom blocks (AttentionBlock and SEBlock) to enhance model focus on relevant features.
Global Average Pooling: Reduces the spatial dimensions and generates a feature map that averages values over each channel.

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.

It does the following:

- Checks the accumulation condition, i.e..: when self.batch is divisible by self.accum_steps, the accumulated gradients are used to update the model's weights.
- Applies the gradients by calling apply_gradients on the optimizer to adjust the weights based on the accumulated gradients.
- Resets the gradients by resetting self.accum_grads to None, clearing the accumulated gradients for the next cycle.
- Increments the batch counter, self.batch, to track the total number of batches processed.

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. AUC

Categorical_accuracy

This metric is useful for multi-class classification problems. It calculates how often the predicted class matches the true class.

For each sample, if the model's predicted class label matches the true class label, the accuracy for that sample is 1; otherwise, it's 0. The categorical accuracy for a batch is the average of these values.

Use Case: Used primarily in multi-class classification problems where labels are one-hot encoded.

Accuracy

It calculates the proportion of correctly classified instances.

It checks the predicted class label against the true label and outputs the percentage of correct predictions.

Use Case: Can be used interchangeably with categorical_accuracy for multi-class problems, though accuracy is more general and sometimes implemented differently depending on the task.

TopKCategoricalAccuracy (Top-3 and Top-5 Accuracy)

It measures the percentage of times the true label is within the top-k predicted probabilities. Here, TopKCategoricalAccuracy is instantiated twice with k=3 and k=5 to measure "top-3 accuracy" and "top-5 accuracy."

- Top-3 Accuracy: For each sample, the metric checks if the true label is one of the top 3 predicted labels (based on prediction probability).
- Top-5 Accuracy: Similarly, it checks if the true label is within the top 5 predictions.

Use Case: Commonly used in multi-class classification tasks with a large number of classes, such as object recognition in images. It's useful for cases where the model can be considered correct even if the exact top prediction isn't correct, if the true label is among the top k predictions.

Precision

Precision (or Positive Predictive Value) measures how many of the samples predicted as a particular class (e.g., "positive") are of that class. It evaluates the quality of positive predictions.

Precision = True Positives / (True Positives + False Positives)

Use Case: Precision is particularly useful when the cost of false positives is high, such as in medical diagnosis, where incorrectly predicting a disease when it isn't present could lead to unnecessary treatment.

Recall

Recall (or Sensitivity) measures how many of the actual positive samples are correctly identified by the model. It evaluates the model's ability to detect positive samples.

Recall = True Positives / (True Positives + False Negatives)

Use Case: Recall is particularly important when the cost of false negatives is high, such as in medical diagnosis or fraud detection, where failing to detect a true positive could have significant consequences.

AUC (Area Under the Curve)

AUC measures the area under the Receiver Operating Characteristic (ROC) curve. The ROC curve plots the true positive rate (recall) against the false positive rate. AUC is often considered a summary of model performance across all classification thresholds.

AUC is computed as the area under the ROC curve, which plots the True Positive Rate (sensitivity) against the False Positive Rate (1-specificity) across various thresholds.

A perfect classifier would have an AUC of 1.0, indicating perfect separation of positive and negative classes. A model with an AUC of 0.5 performs no better than random chance.

Use Case: AUC is especially useful in binary classification and for comparing the relative performance of models, especially when dealing with imbalanced datasets where accuracy might be misleading.

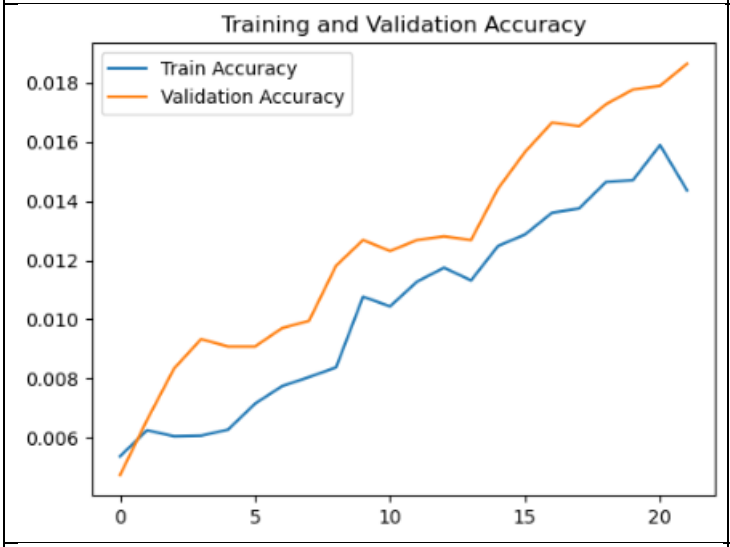
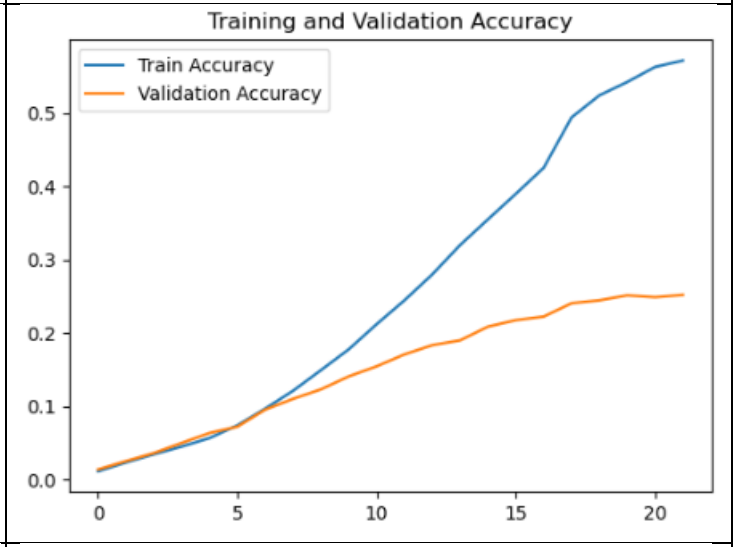
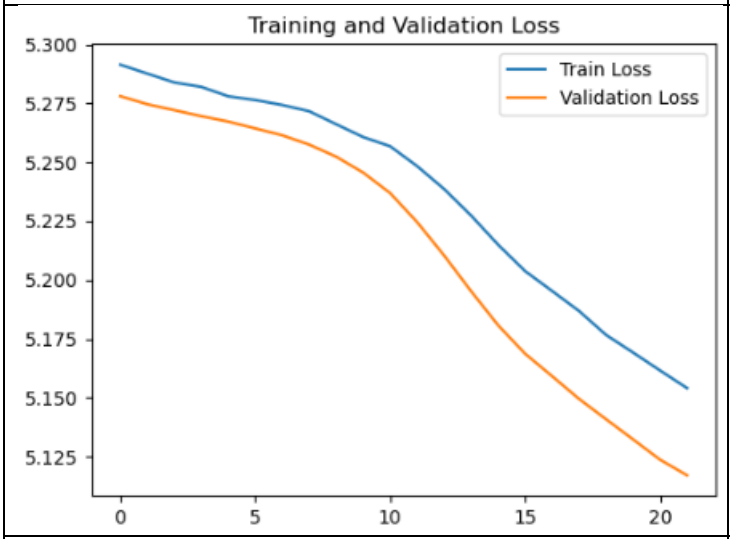
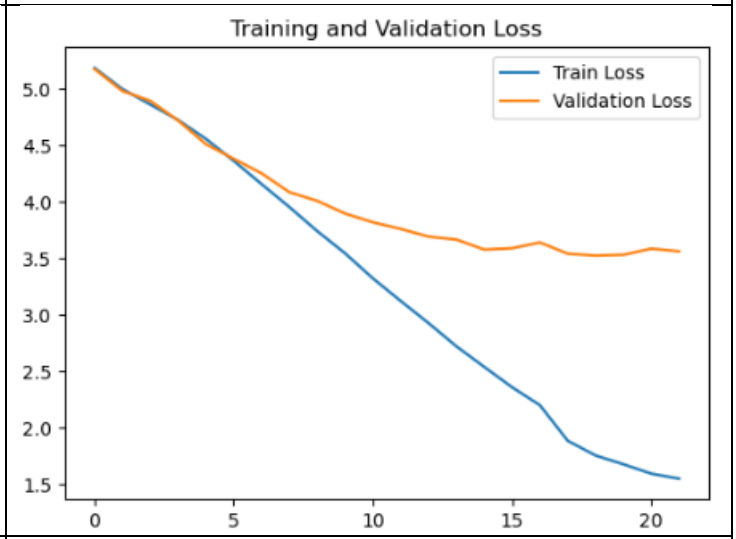
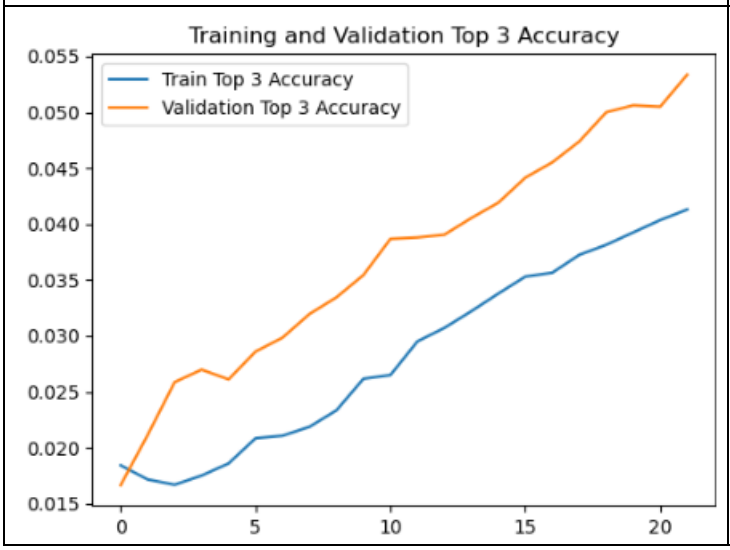
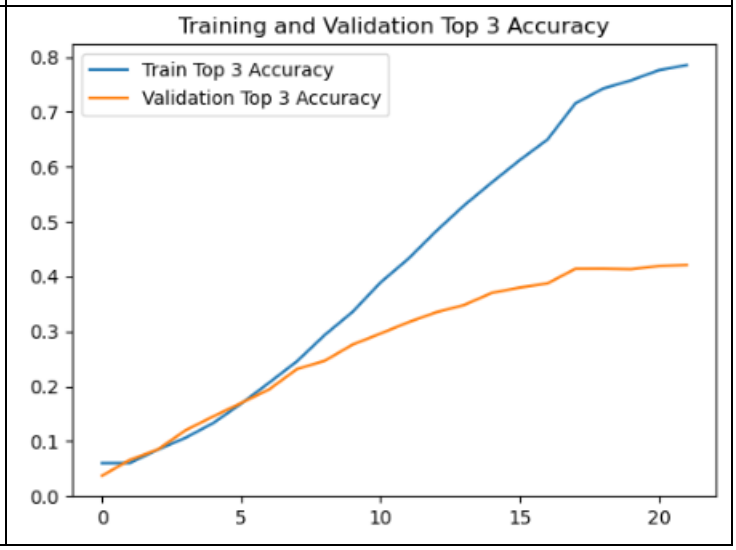
Execution time

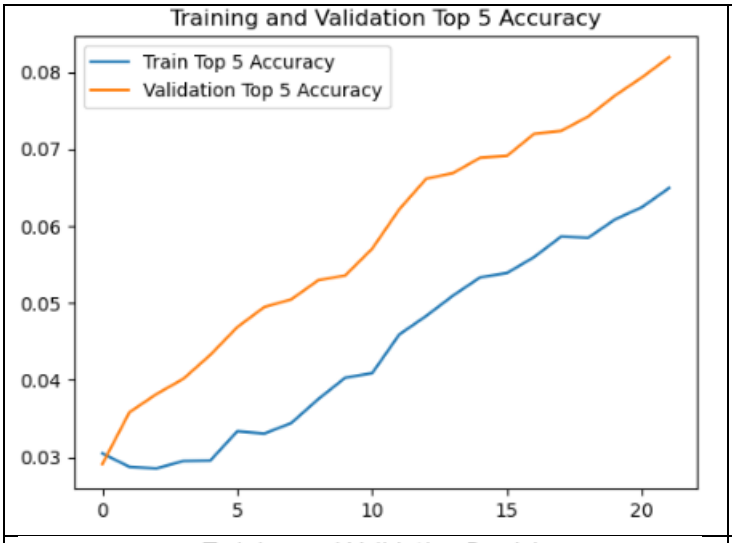
	Default hyper parameters		**Optimal hyper parameters**	
Data augmentation	**Execution time (minutes)**	**CPU Execution Time (minutes)**	**Execution time (minutes)**	**CPU Execution Time (minutes)**
Data augmentation	30.645	4.52	30.645	4.52
Hyper parameter Search	N/A	N/A	N/A	N/A
model1	187.98	90.26	185.31	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+

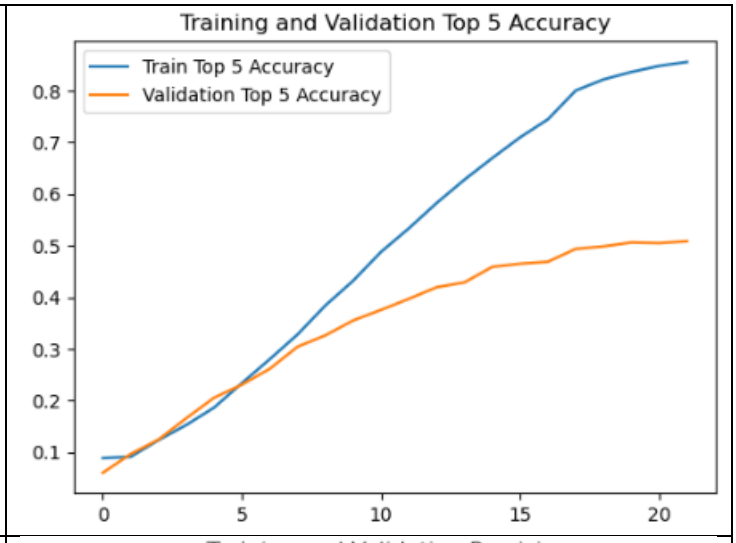
On an average, 30 hours were needed to run the notebook once and over 15 iterations had been made during the evolution of the code.

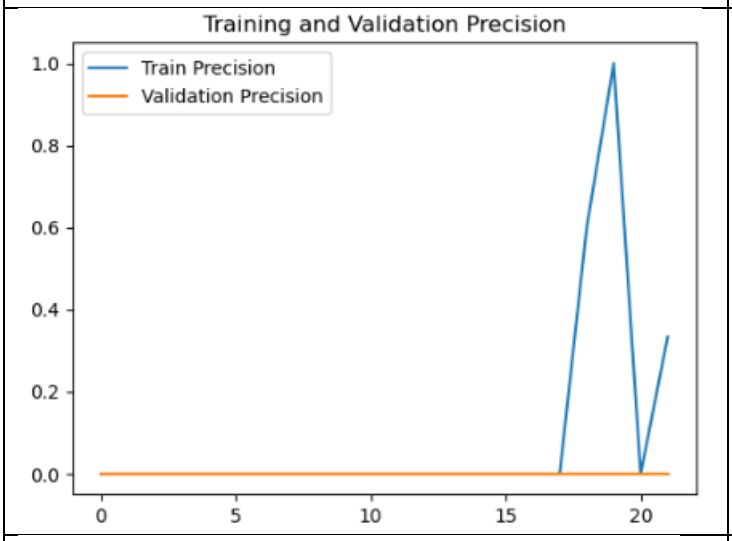
Results of the various models

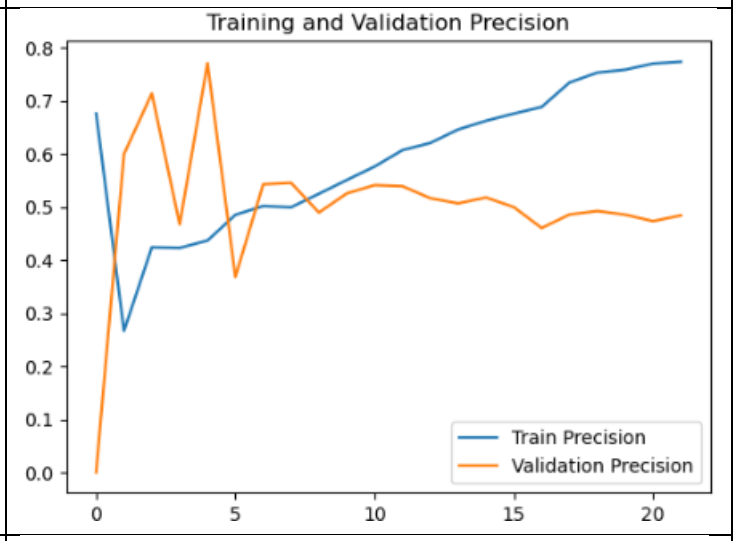
AlexNet Results

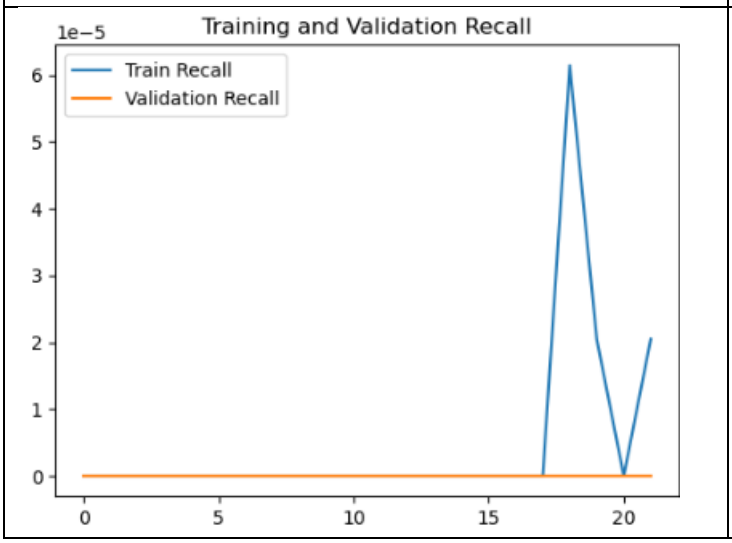
With Default hyper-parameters	With optimal hyper-parameters
	
	
	

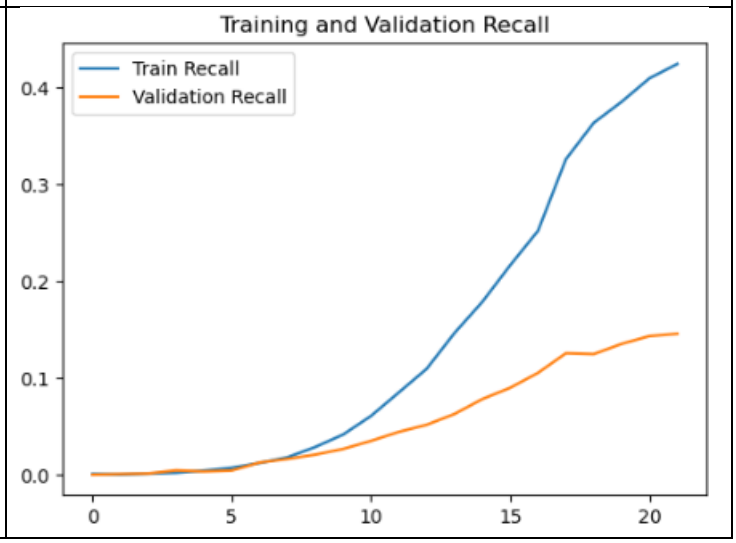

Training and Validation Top 5 Accuracy
Train Top 5 Accuracy
Validation Top 5 Accuracy

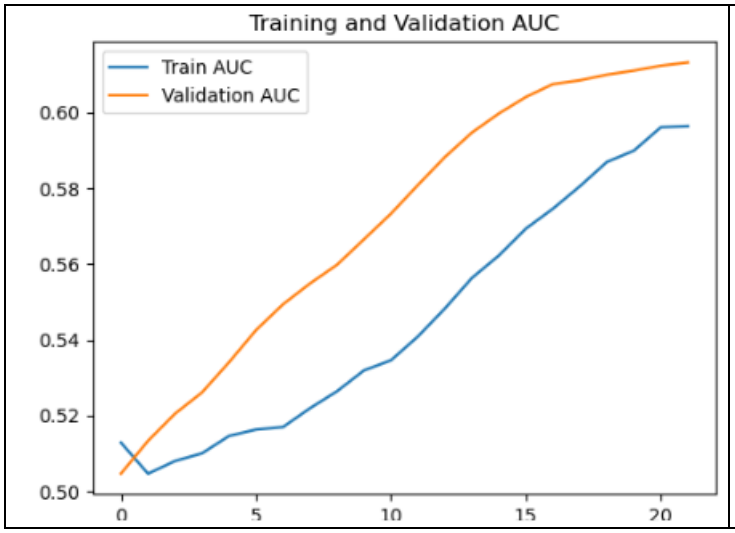

Training and Validation Top 5 Accuracy
Train Top 5 Accuracy
Validation Top 5 Accuracy

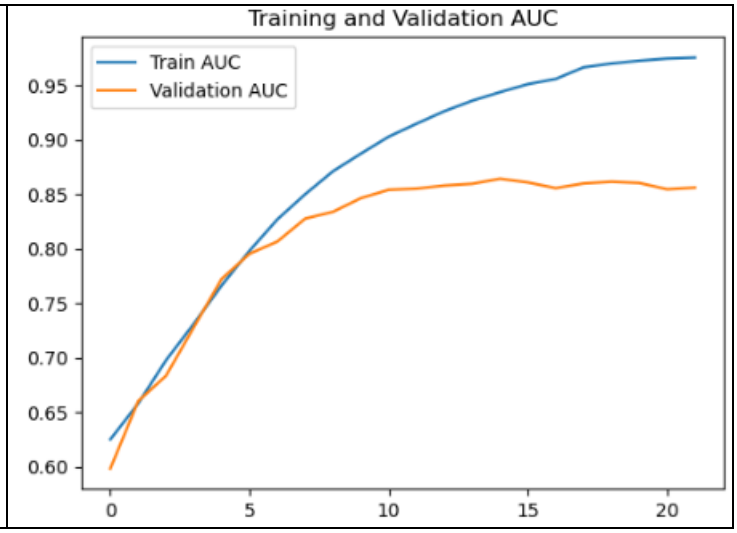

Training and Validation Precision
Train Precision
Validation Precision


Training and Validation Precision
Train Precision
Validation Precision

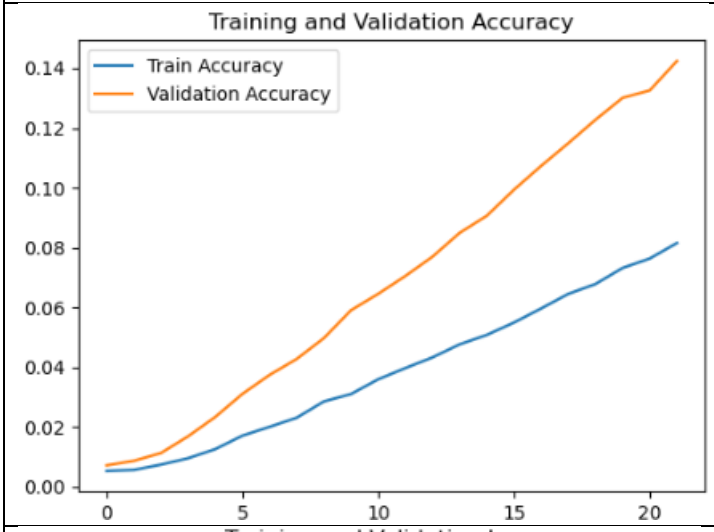

Training and Validation Recall
1e−5
Train Recall
Validation Recall

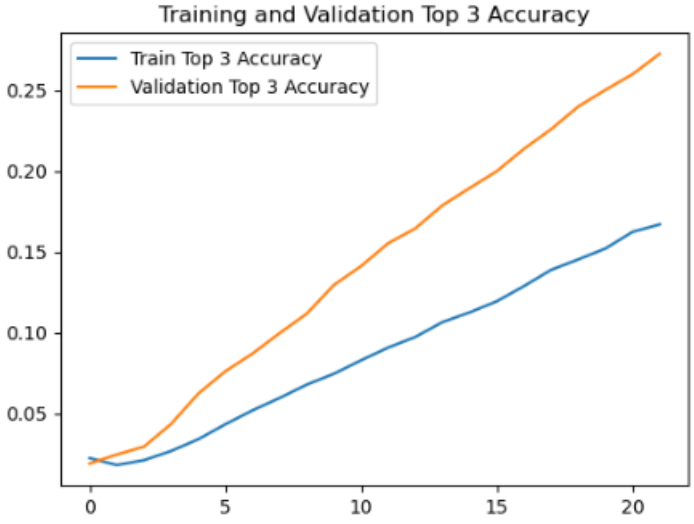

Training and Validation Recall
Train Recall
Validation Recall

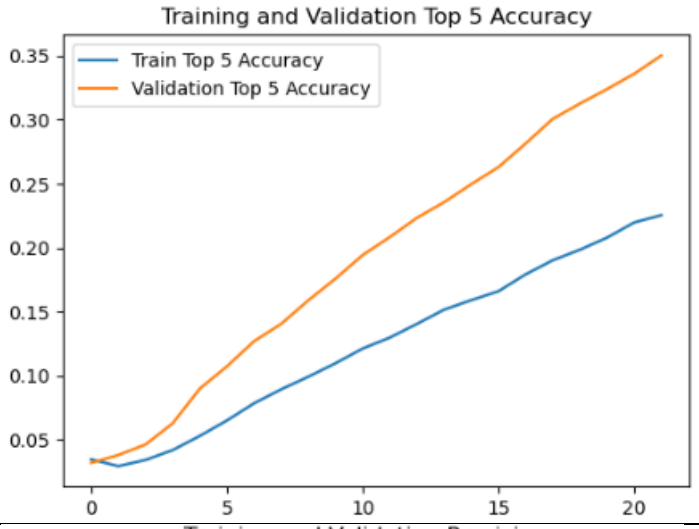


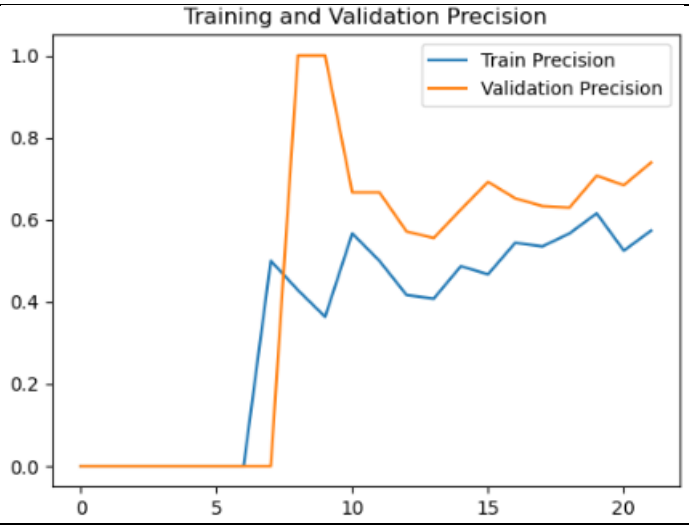


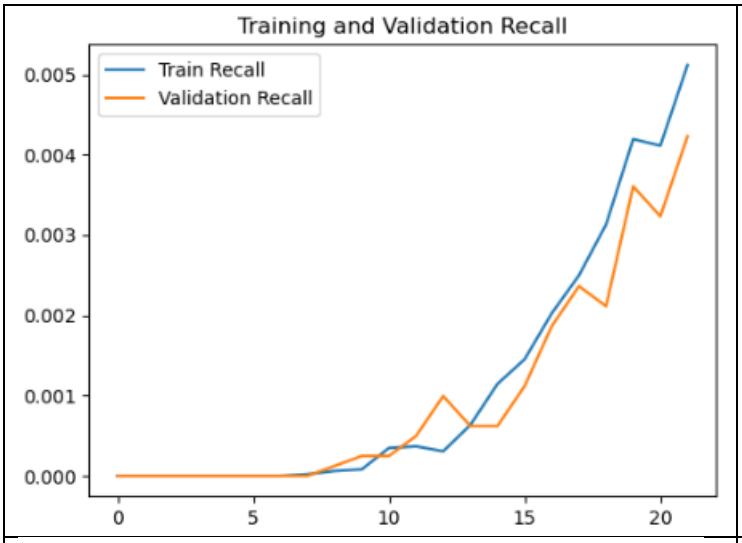
VGG16 Results

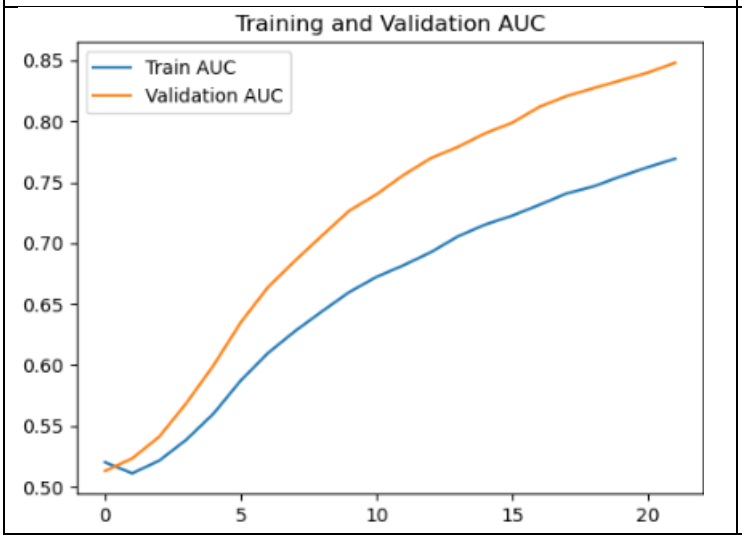
With Default hyper-parameters	With optimal hyper-parameters
	
	


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

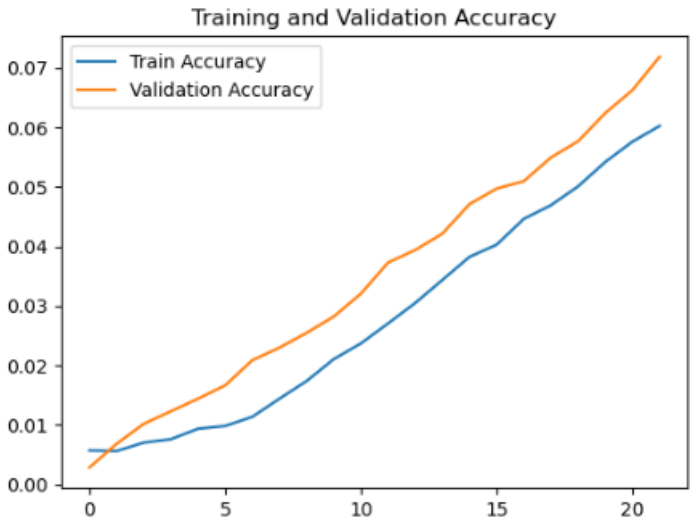
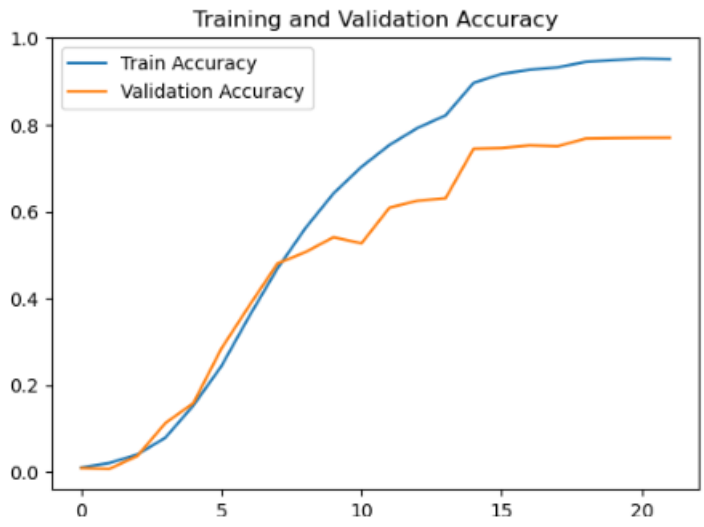

Training and Validation Top 5 Accuracy
Train Top 5 Accuracy
Validation Top 5 Accuracy
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0
5
10
15
20

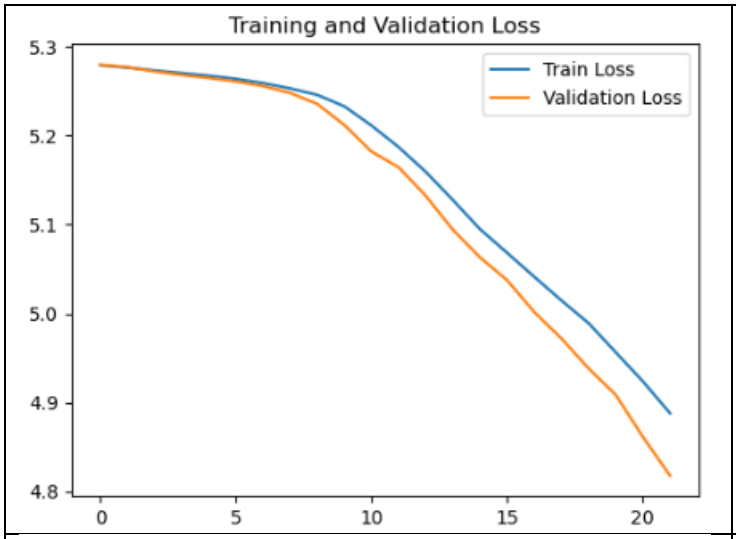

Training and Validation Precision
Train Precision
Validation Precision
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20

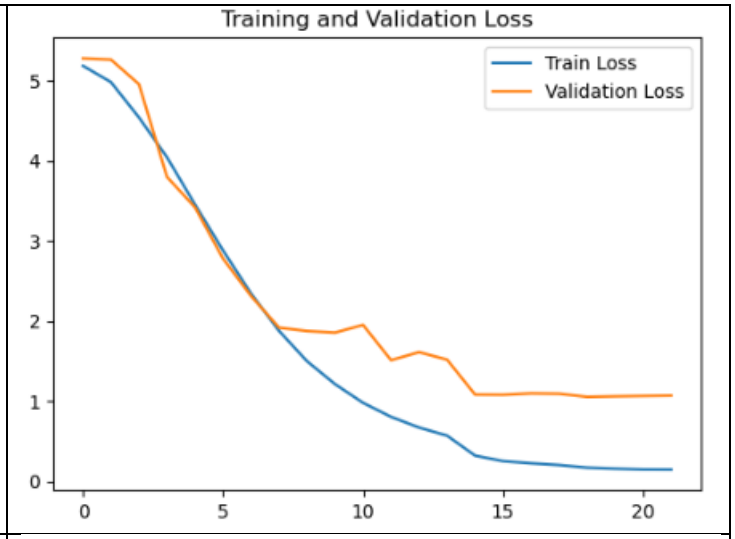


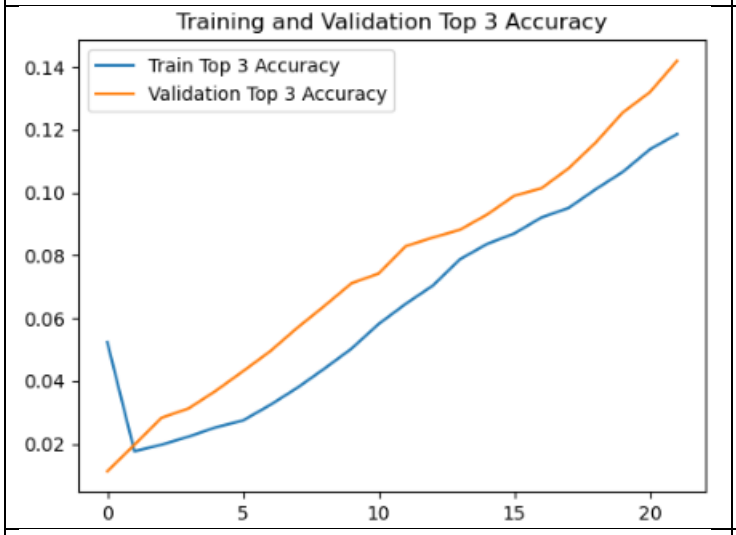


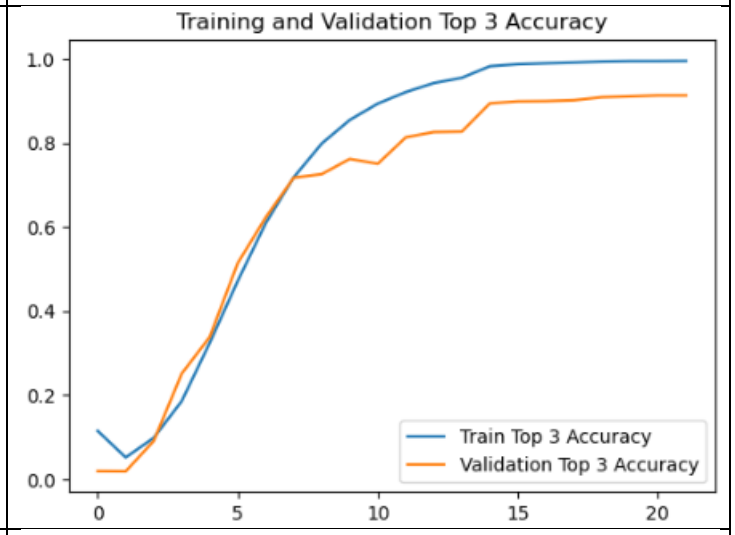
ResNet50 Results

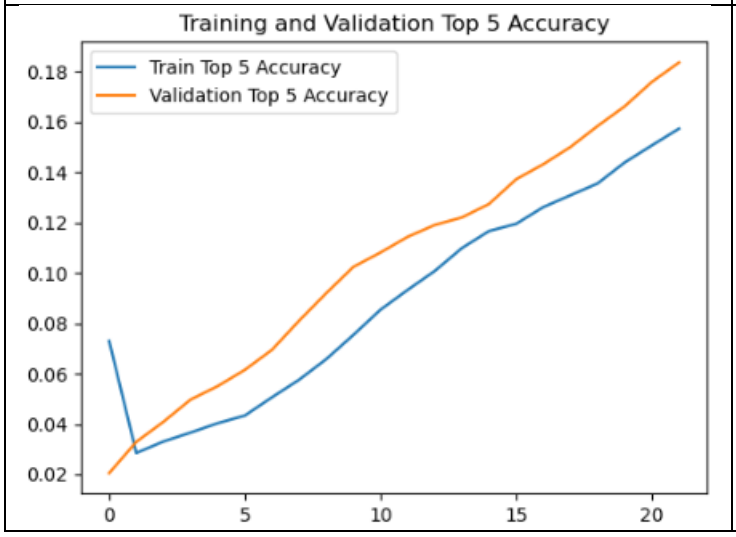
With Default hyper-parameters	With optimal hyper-parameters
	

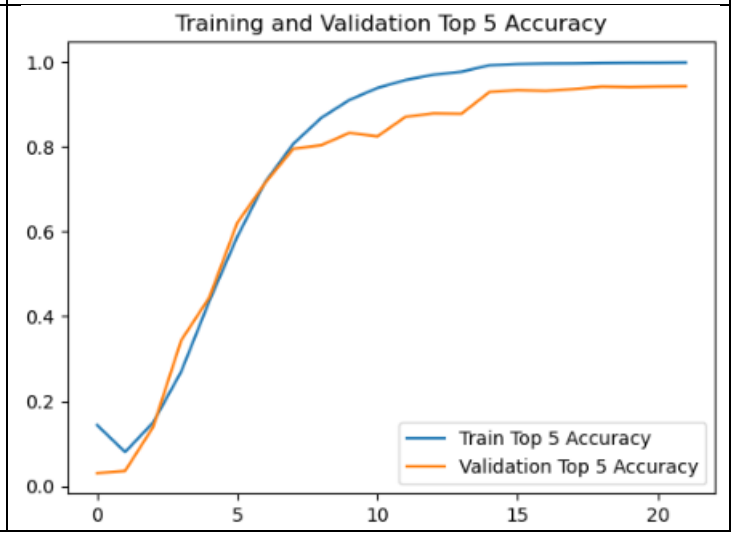

Training and Validation Loss
Train Loss
Validation Loss

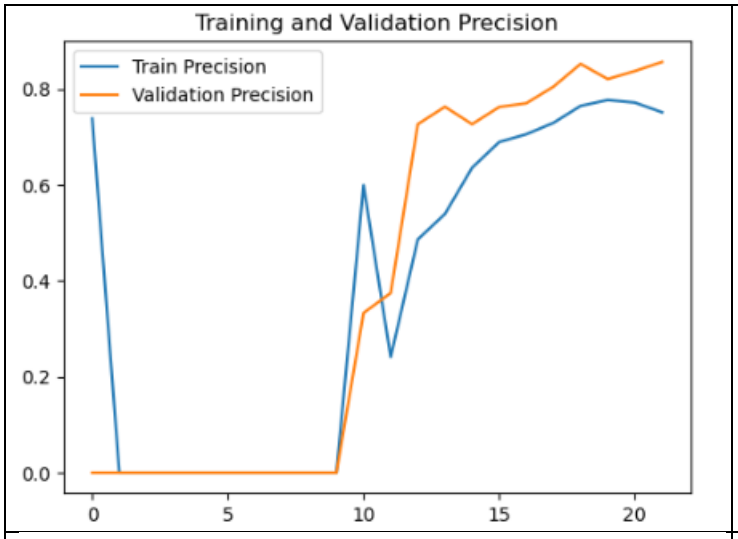

Training and Validation Loss
Train Loss
Validation Loss

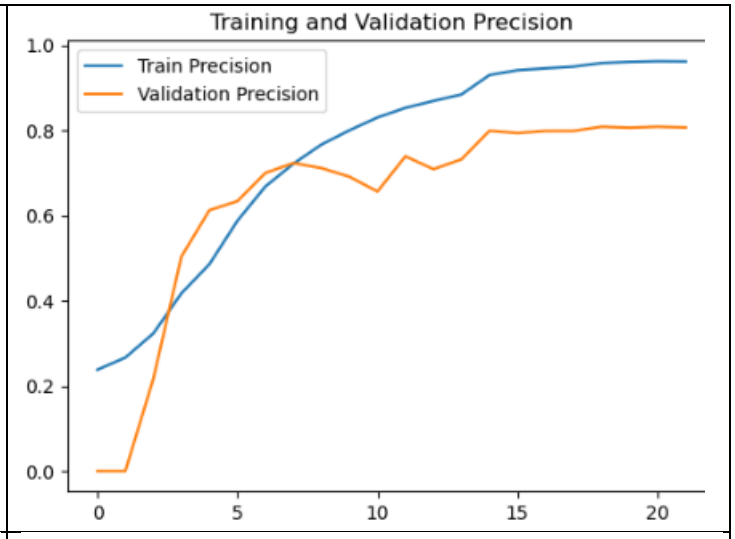

Training and Validation Top 3 Accuracy
Train Top 3 Accuracy
Validation Top 3 Accuracy

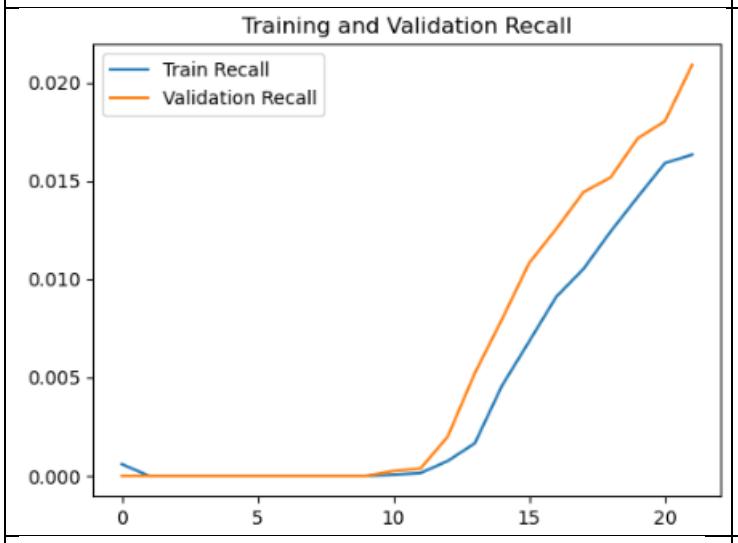

Training and Validation Top 3 Accuracy
Train Top 3 Accuracy
Validation Top 3 Accuracy

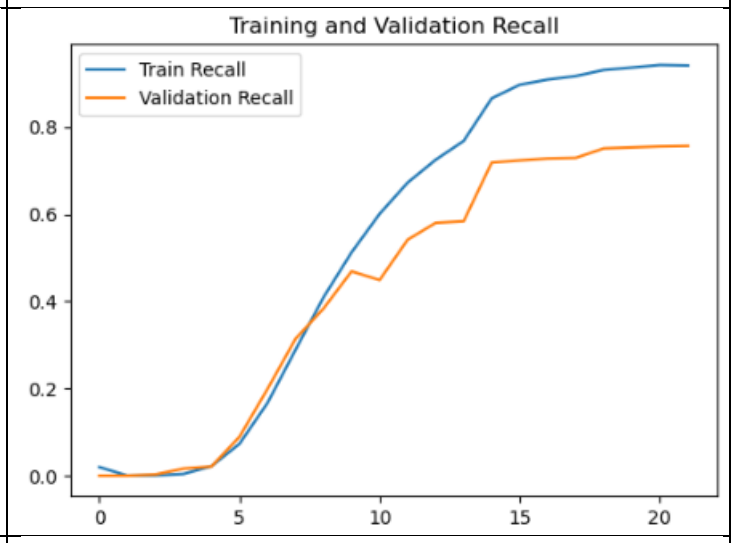

Training and Validation Top 5 Accuracy
Train Top 5 Accuracy
Validation Top 5 Accuracy

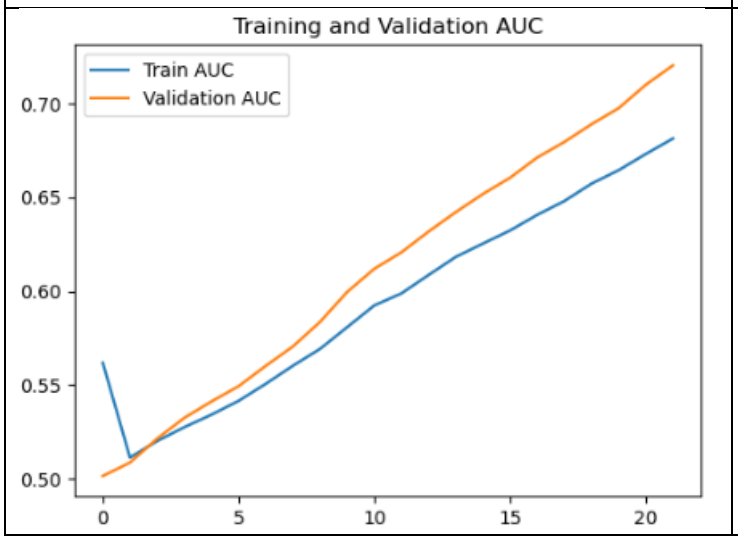

Training and Validation Top 5 Accuracy
Train Top 5 Accuracy
Validation Top 5 Accuracy

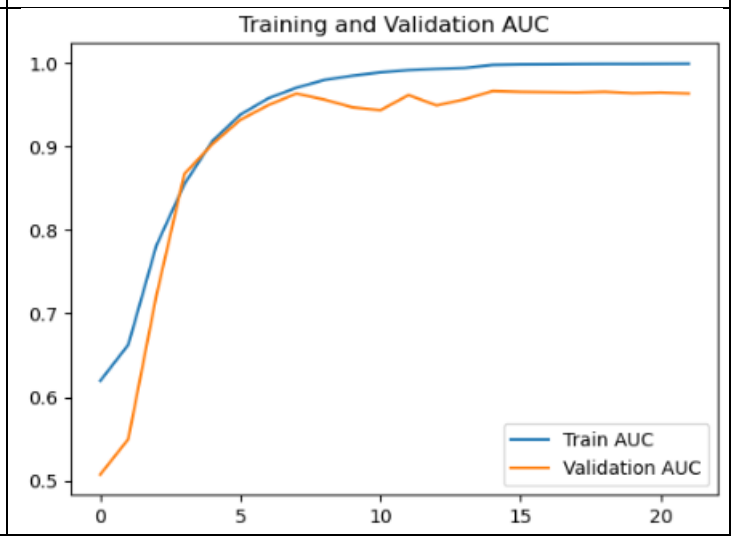

Training and Validation Precision
Train Precision
Validation Precision


Training and Validation Precision
Train Precision
Validation Precision

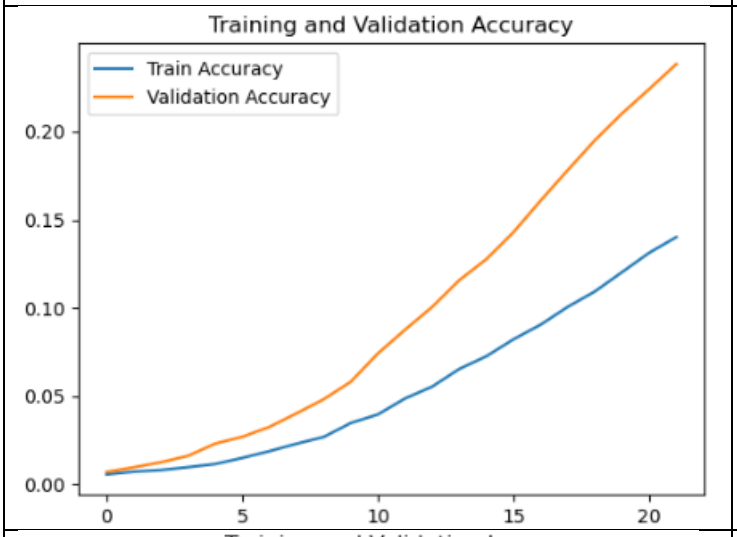
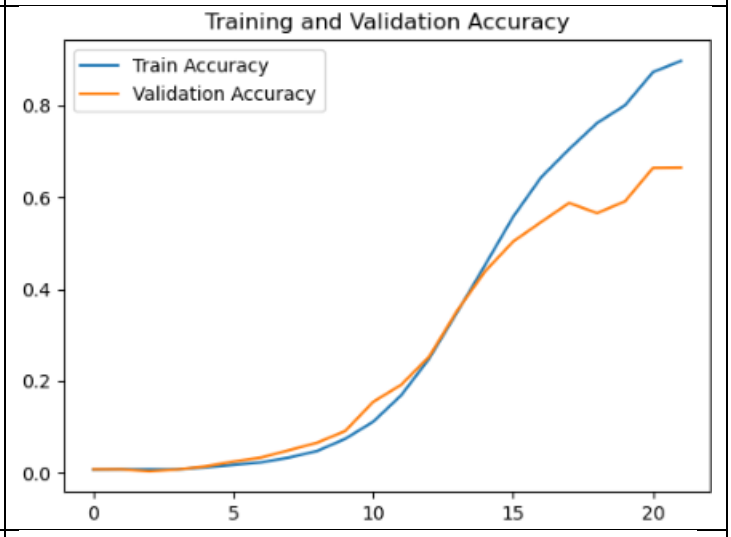
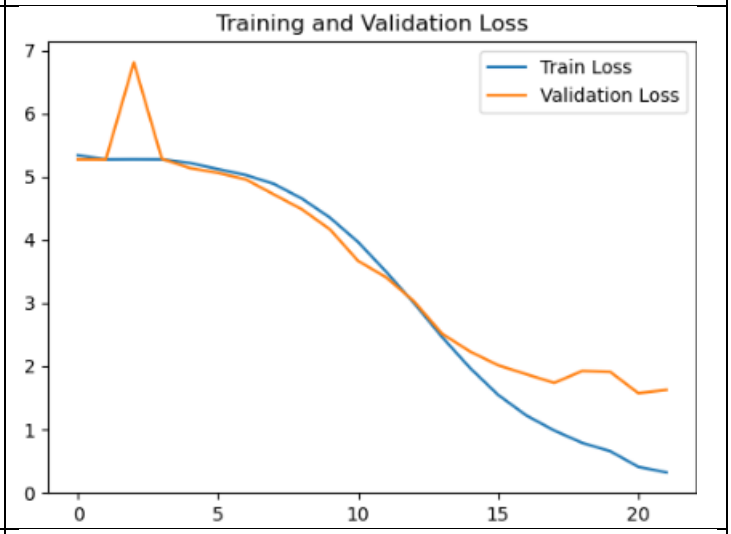
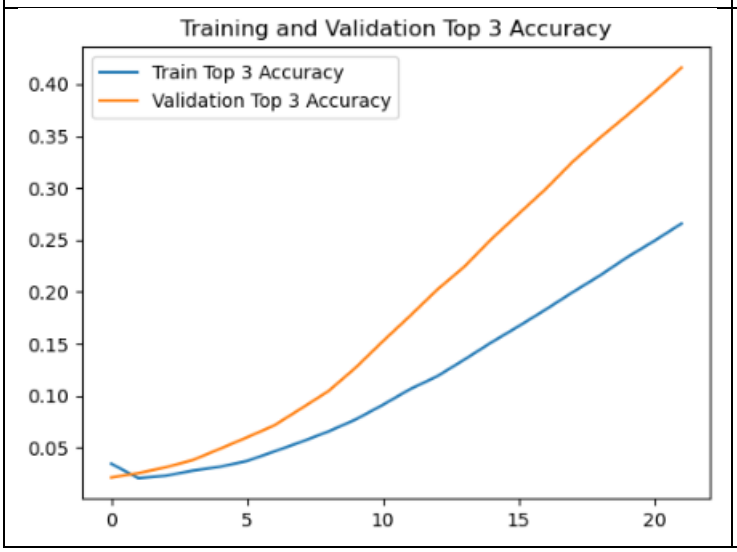
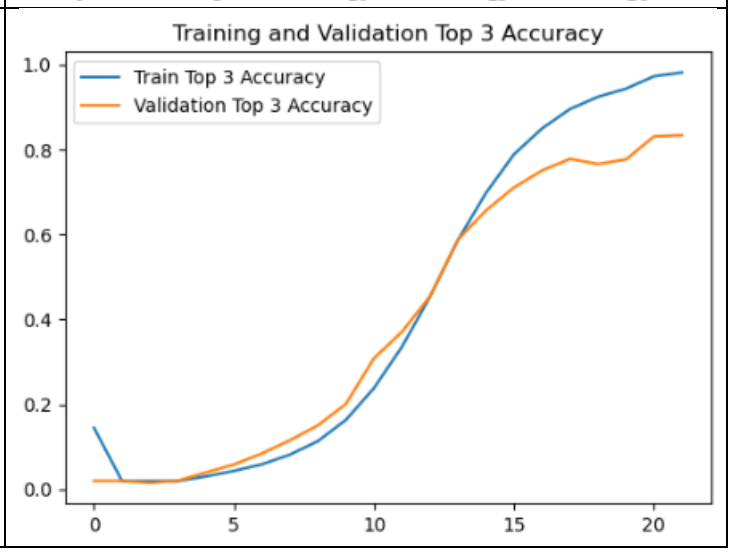

Training and Validation Recall
Train Recall
Validation Recall

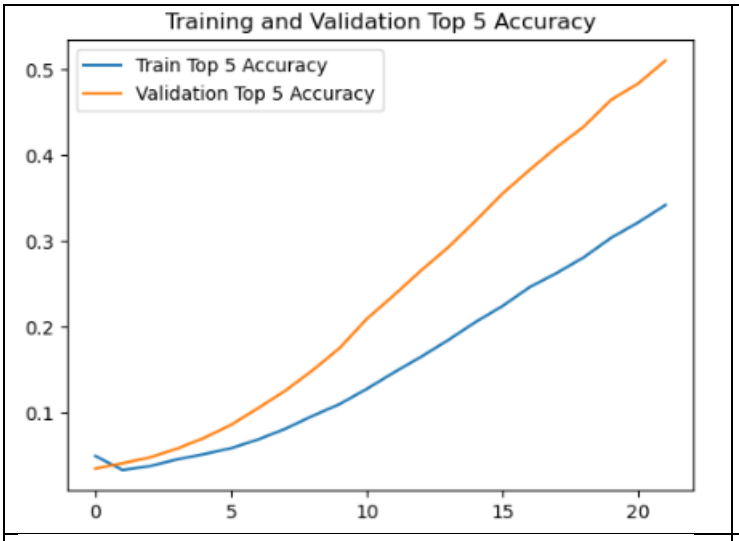

Training and Validation Recall
Train Recall
Validation Recall

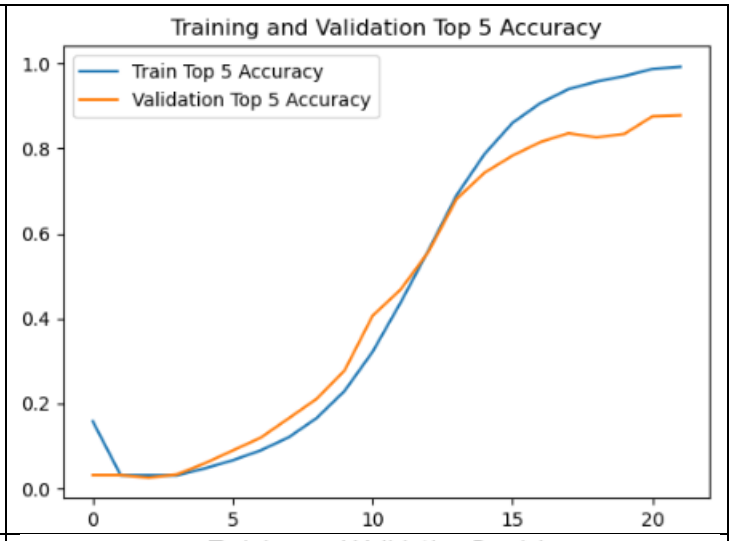

Training and Validation AUC
Train AUC
Validation AUC

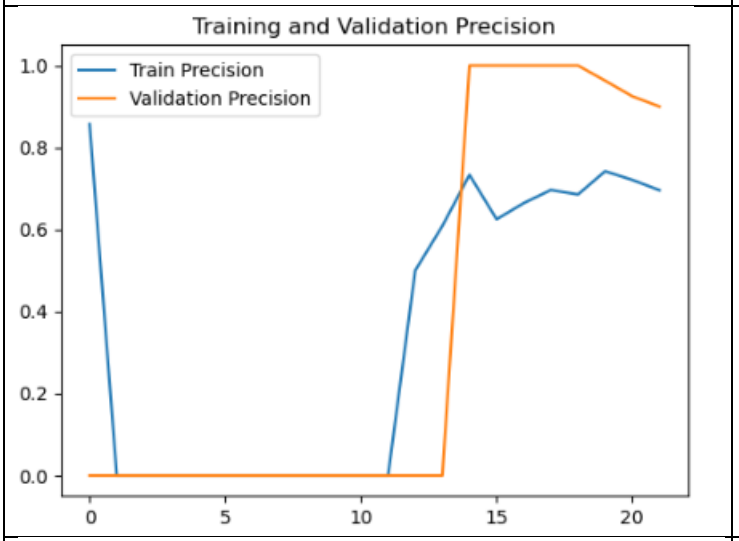

Training and Validation AUC
Train AUC
Validation AUC

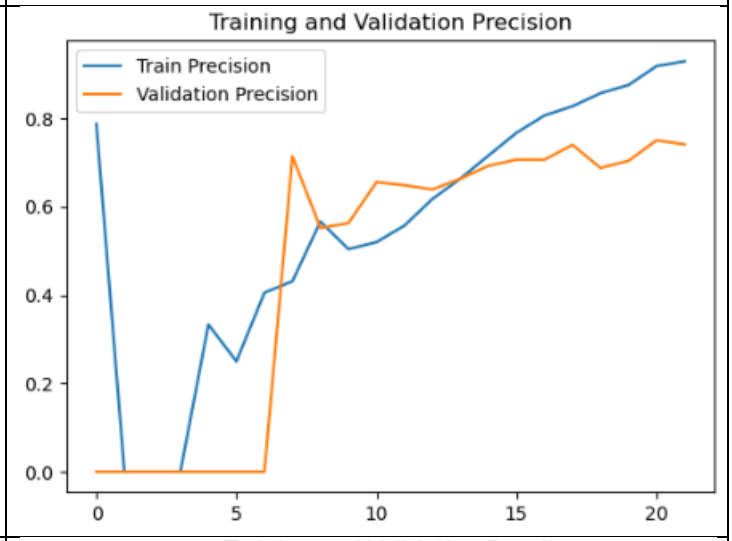
DenseNet121 Results

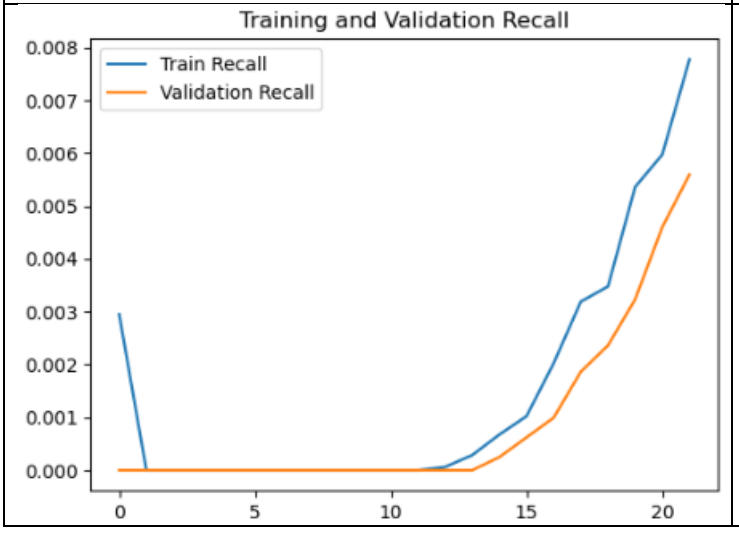
With Default hyper-parameters	With optimal hyper-parameters
	
	
	

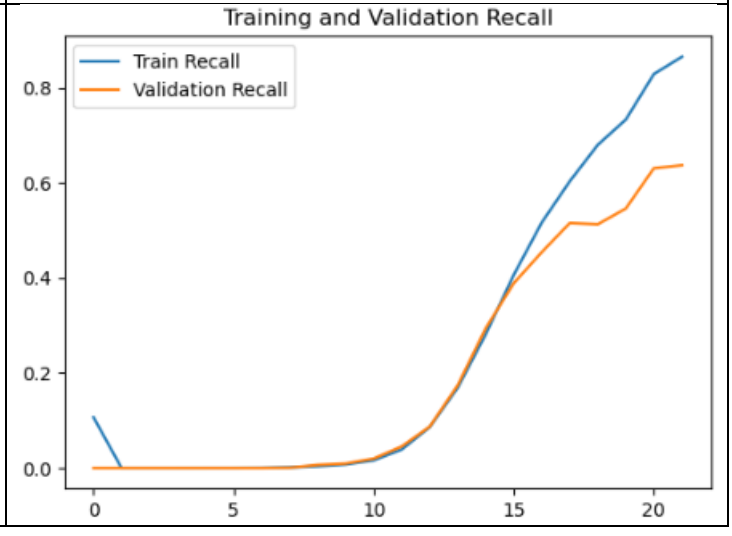

Training and Validation Top 5 Accuracy
Train Top 5 Accuracy
Validation Top 5 Accuracy

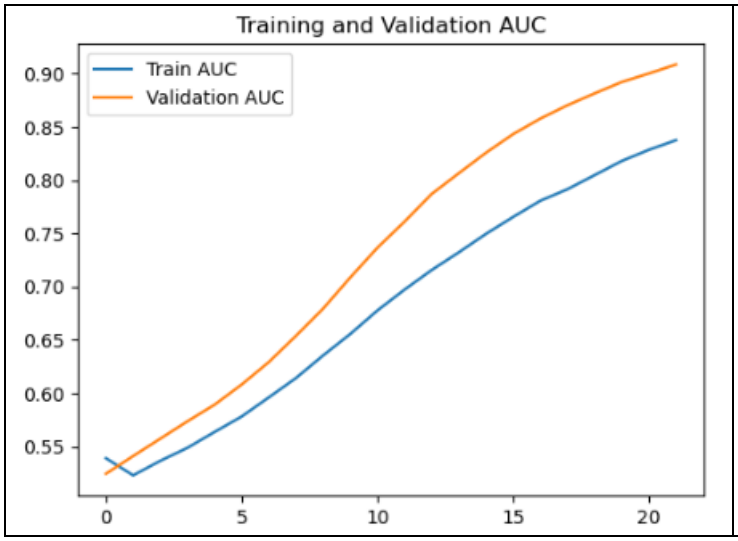

Training and Validation Top 5 Accuracy
Train Top 5 Accuracy
Validation Top 5 Accuracy

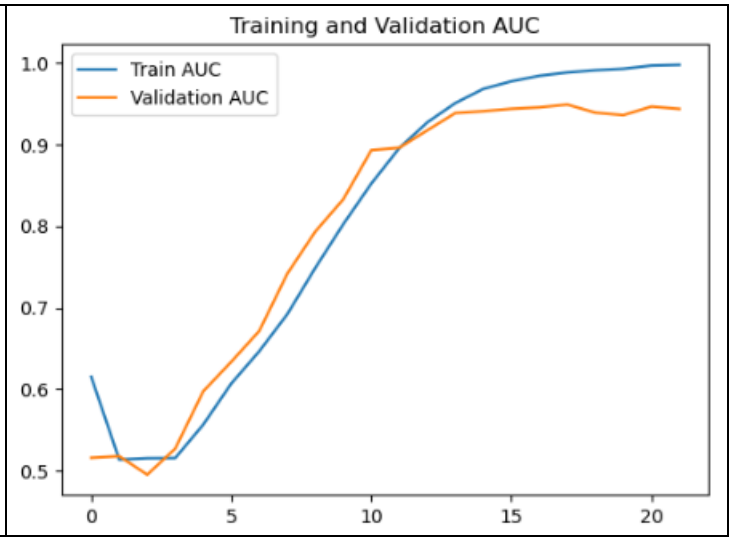

Training and Validation Precision
Train Precision
Validation Precision


Training and Validation Precision
Train Precision
Validation Precision


Training and Validation Recall
Train Recall
Validation Recall

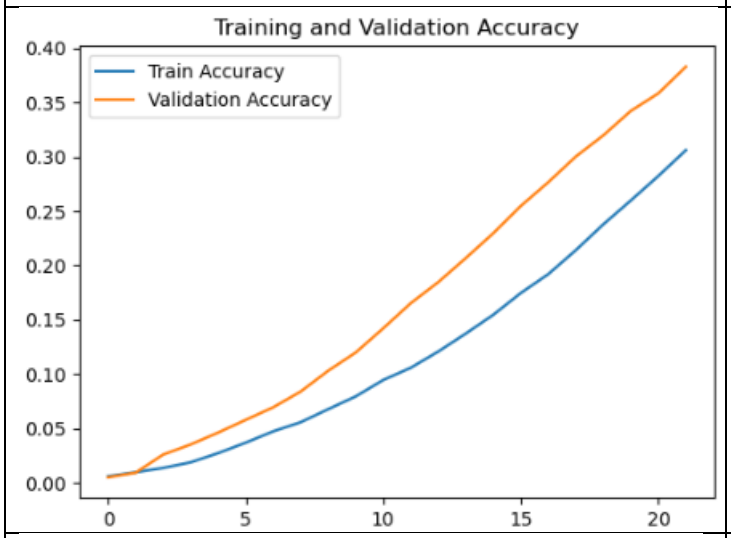

Training and Validation Recall
Train Recall
Validation Recall

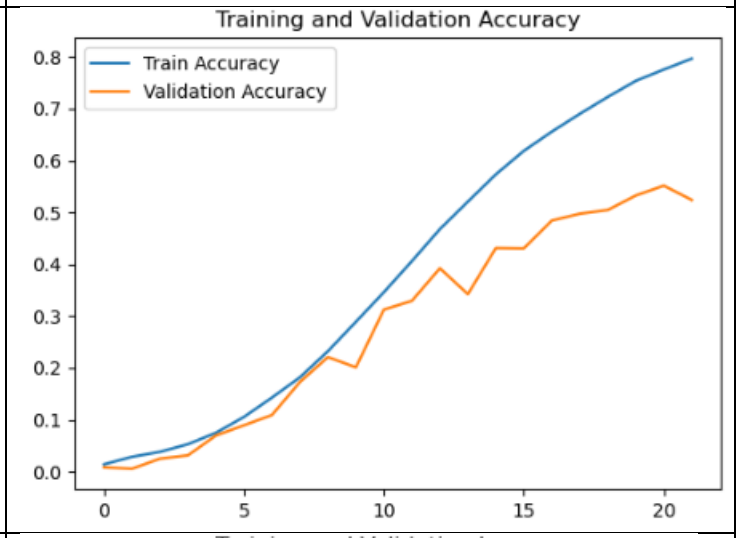


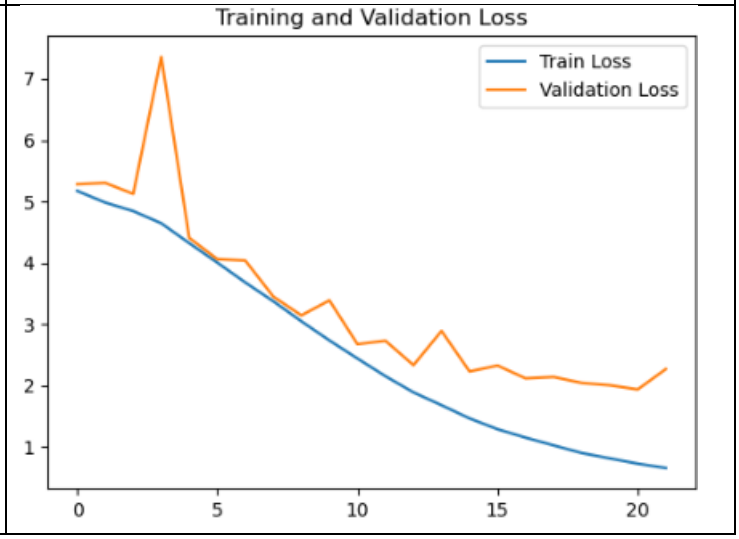


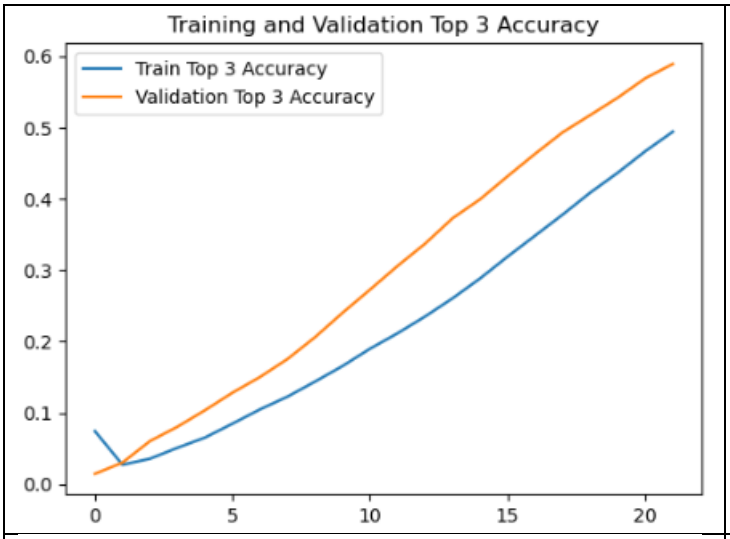
ImageNet Results

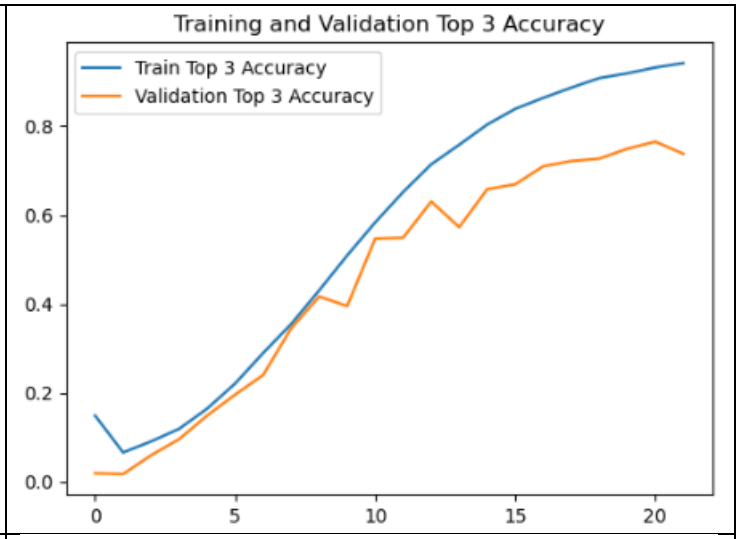
With Default hyper-parameters	With optimal hyper-parameters

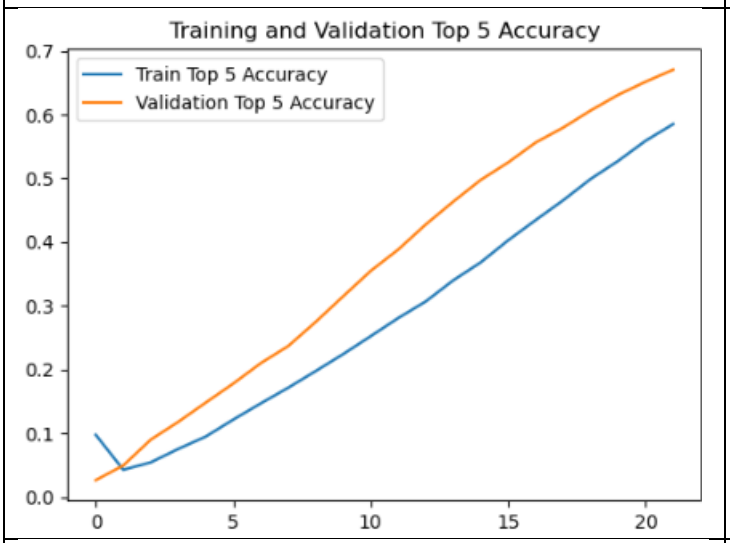


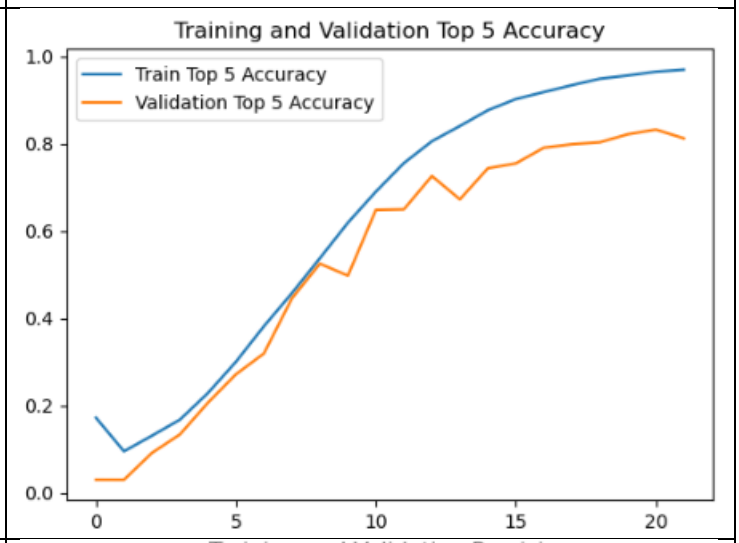


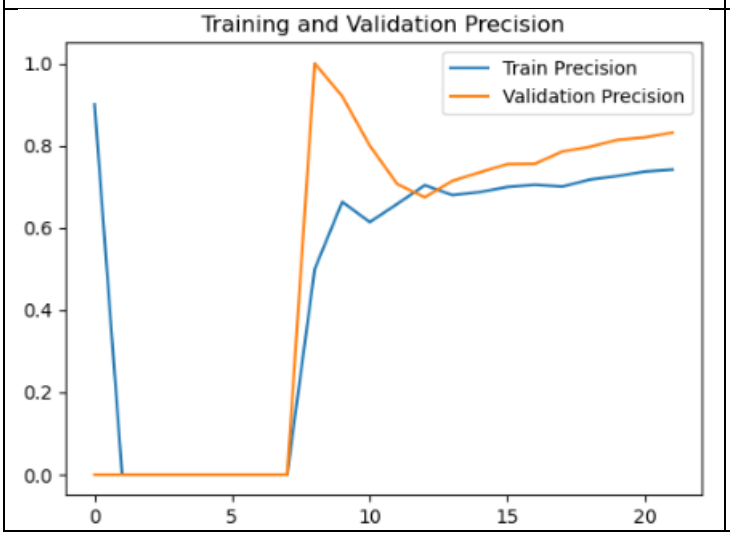


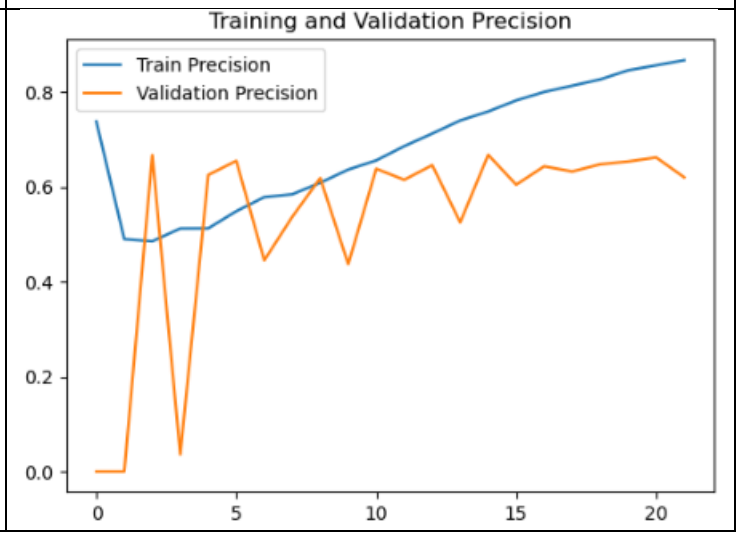

Training and Validation Top 3 Accuracy
Train Top 3 Accuracy
Validation Top 3 Accuracy

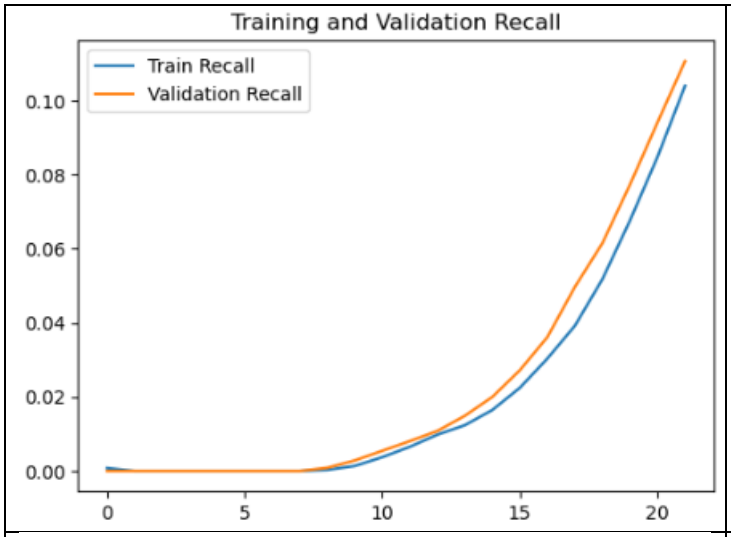

Training and Validation Top 3 Accuracy
Train Top 3 Accuracy
Validation Top 3 Accuracy

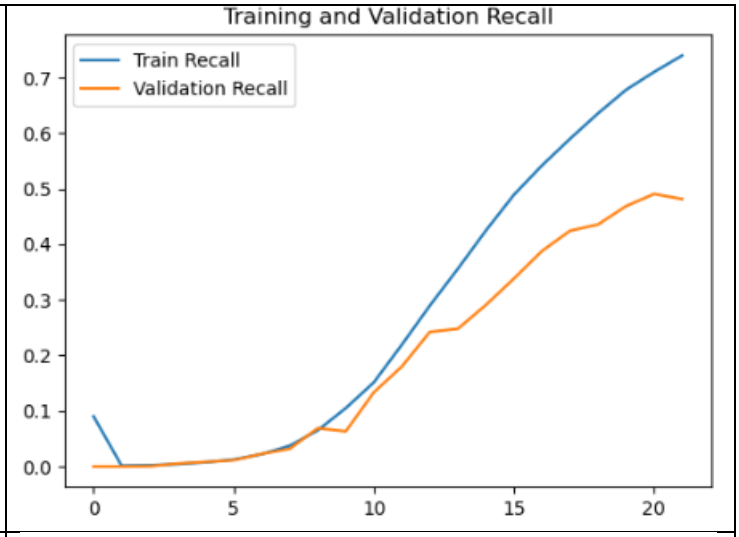

Training and Validation Top 5 Accuracy
Train Top 5 Accuracy
Validation Top 5 Accuracy

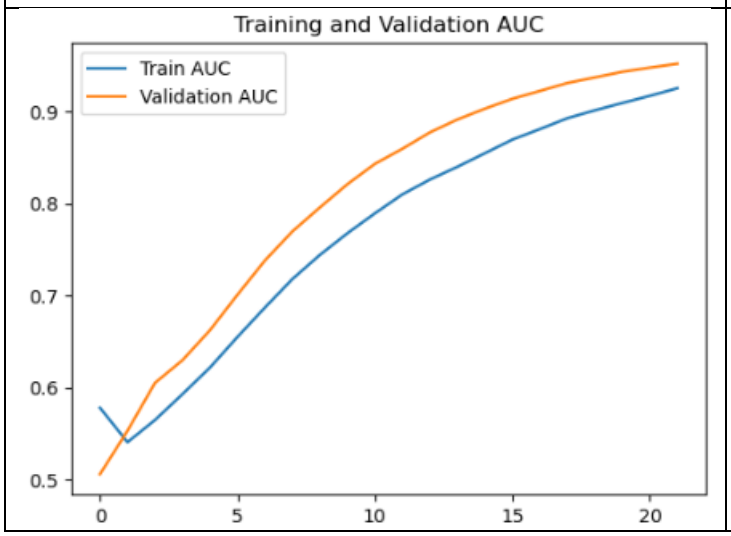

Training and Validation Top 5 Accuracy
Train Top 5 Accuracy
Validation Top 5 Accuracy

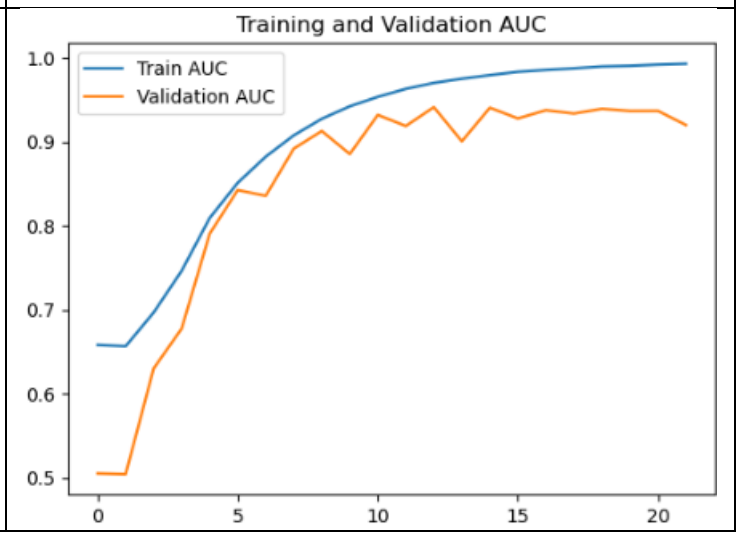

Training and Validation Precision
Train Precision
Validation Precision


Training and Validation Precision
Train Precision
Validation Precision



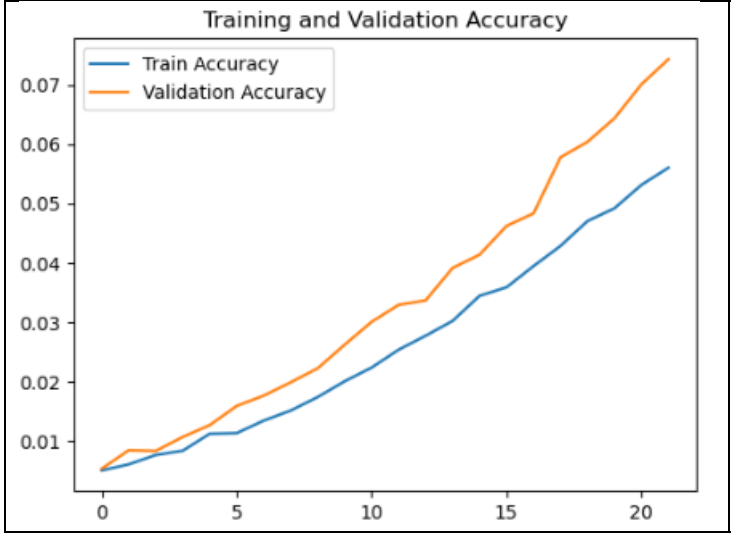


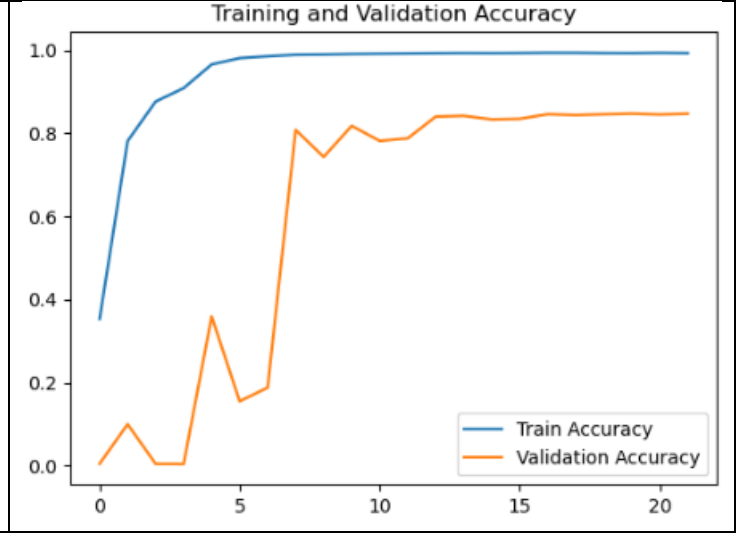


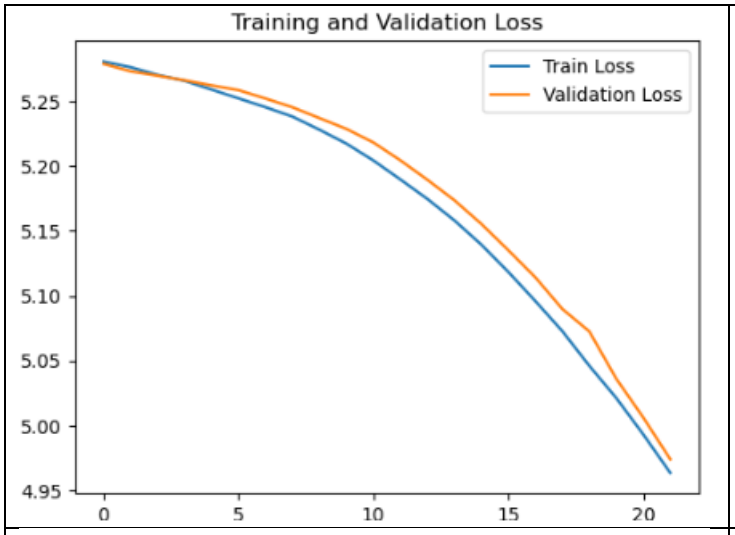


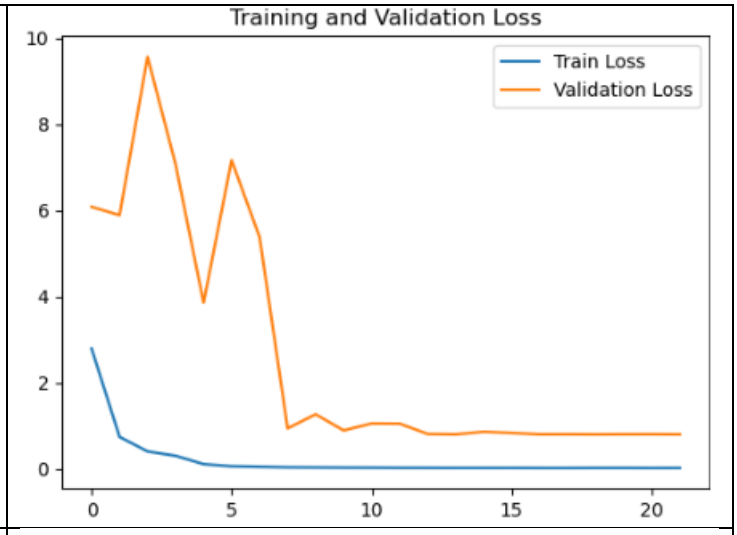
EfficientNetB0 Results

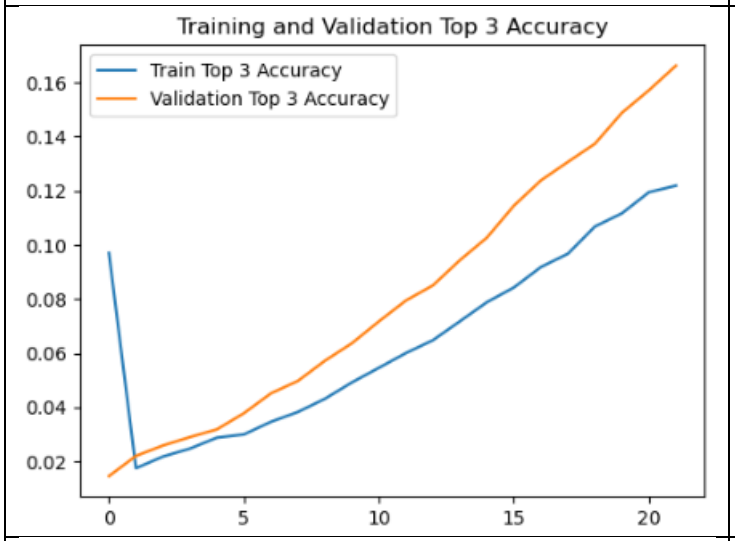
With Default hyper-parameters	With optimal hyper-parameters

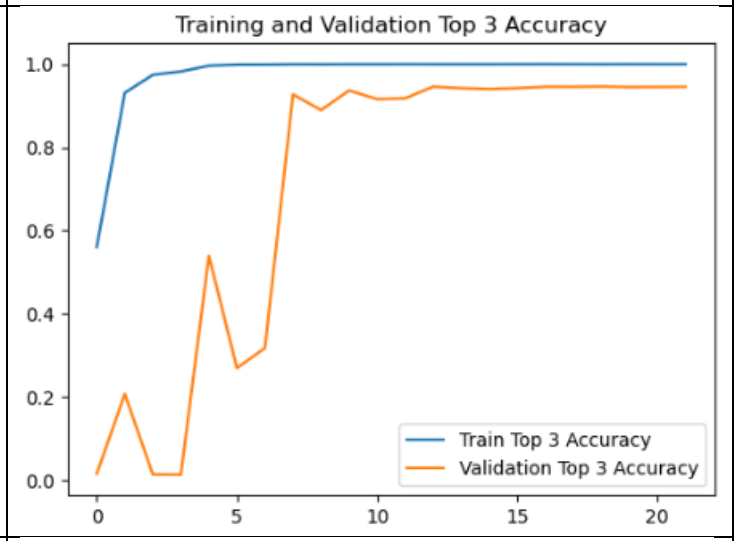


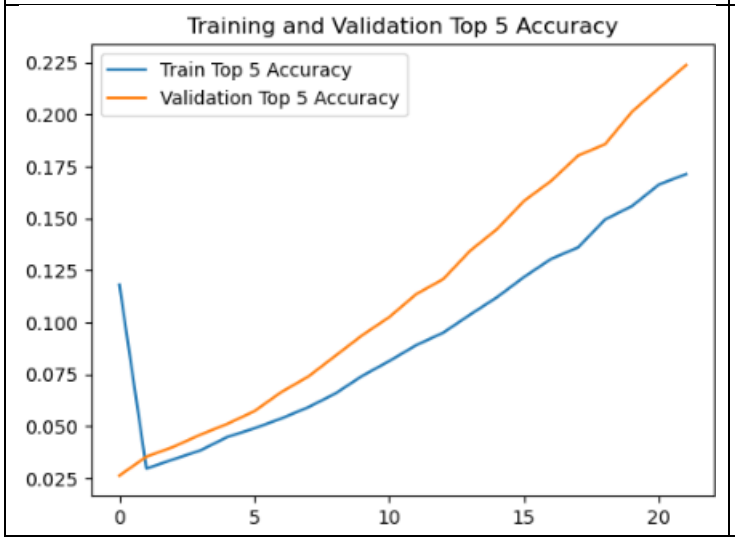


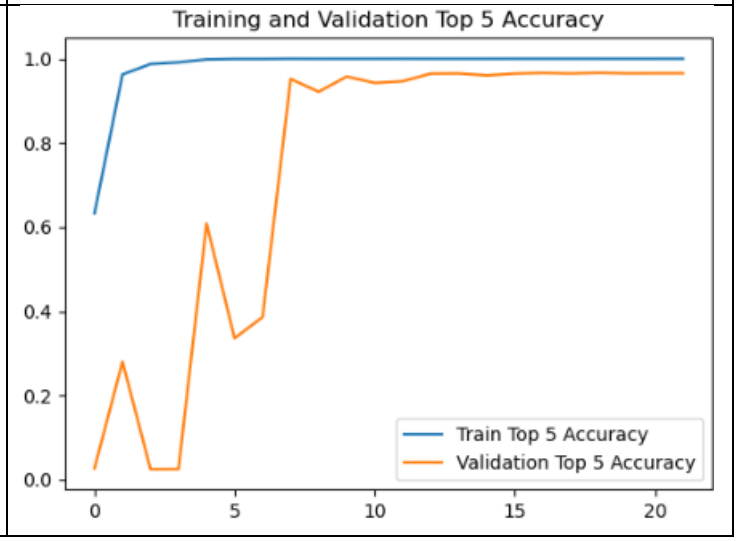

Training and Validation Loss
Train Loss
Validation Loss

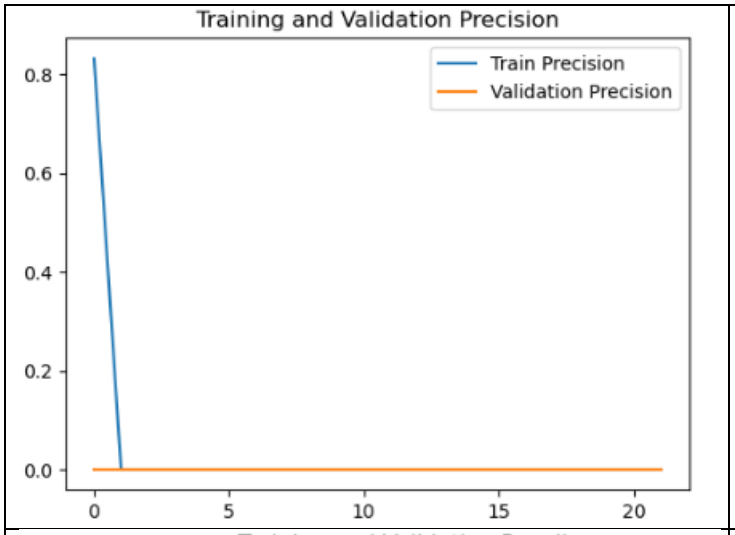

Training and Validation Loss
Train Loss
Validation Loss

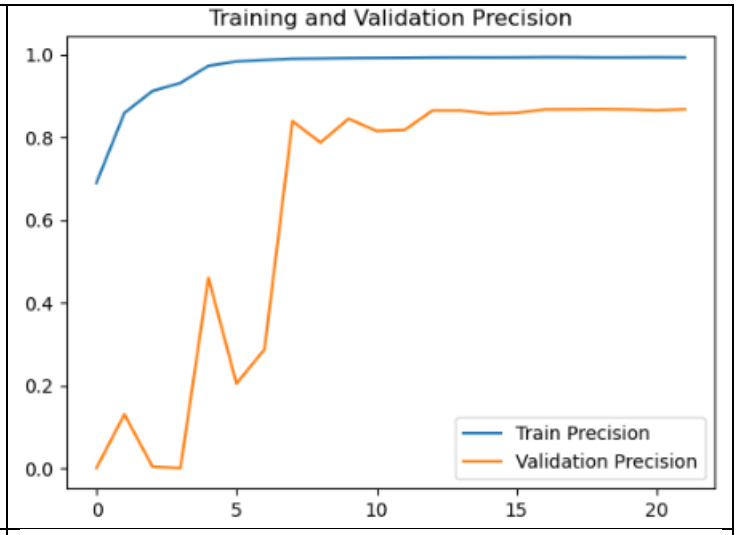

Training and Validation Top 3 Accuracy
Train Top 3 Accuracy
Validation Top 3 Accuracy

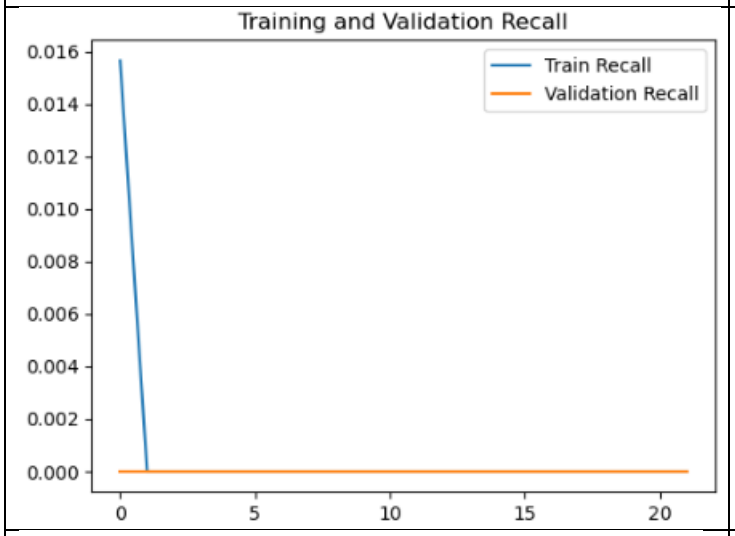

Training and Validation Top 3 Accuracy
Train Top 3 Accuracy
Validation Top 3 Accuracy

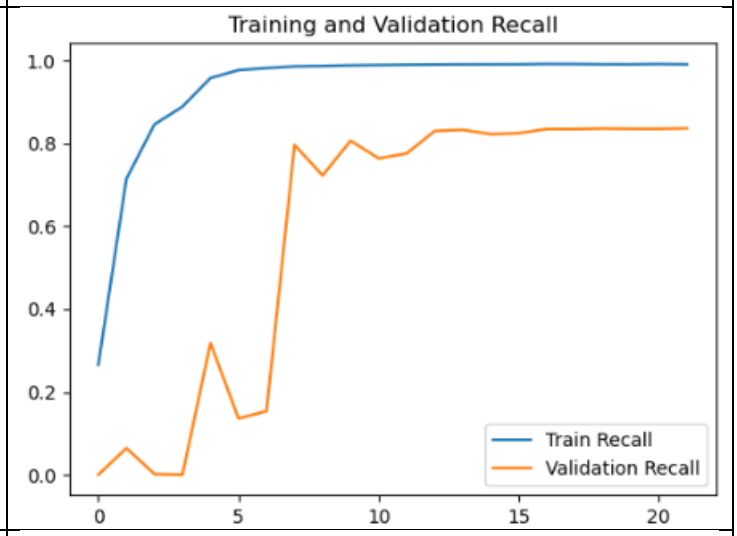

Training and Validation Top 5 Accuracy
Train Top 5 Accuracy
Validation Top 5 Accuracy

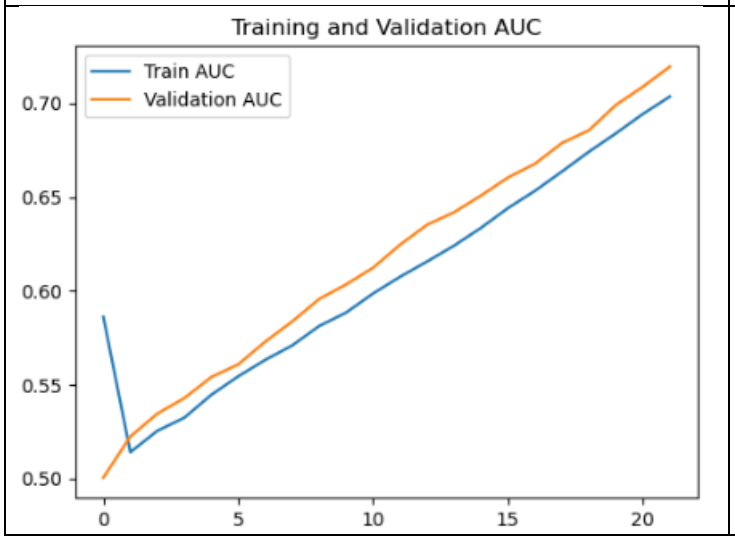

Training and Validation Top 5 Accuracy
Train Top 5 Accuracy
Validation Top 5 Accuracy

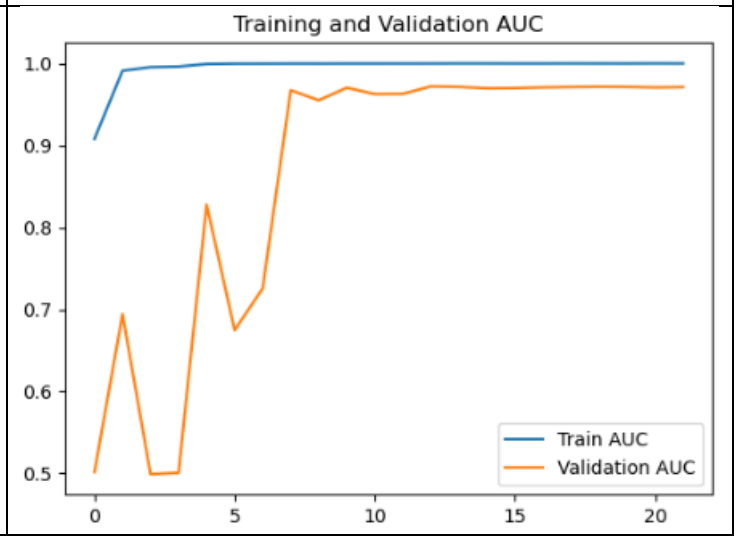






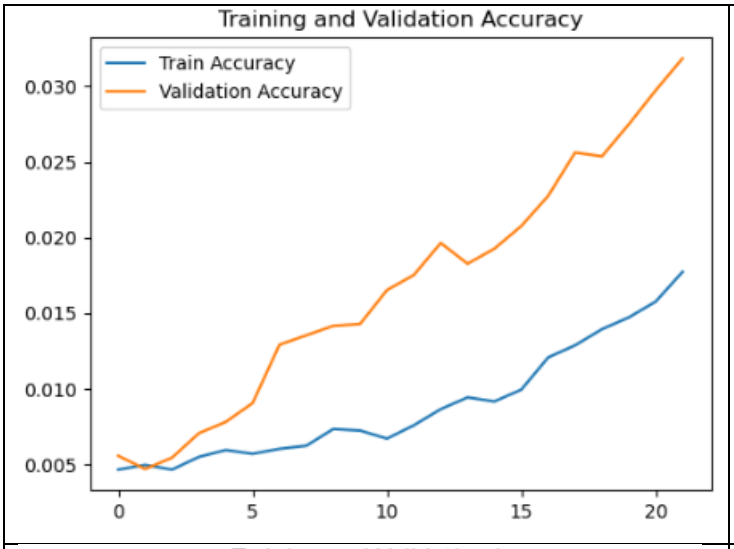


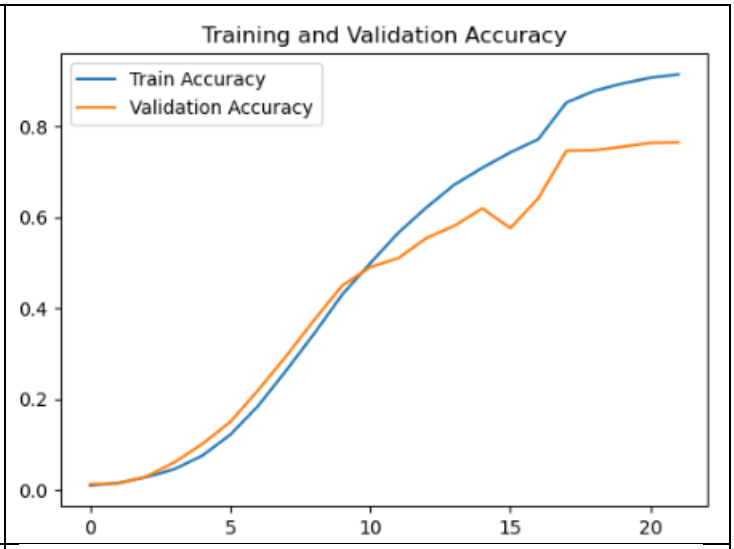


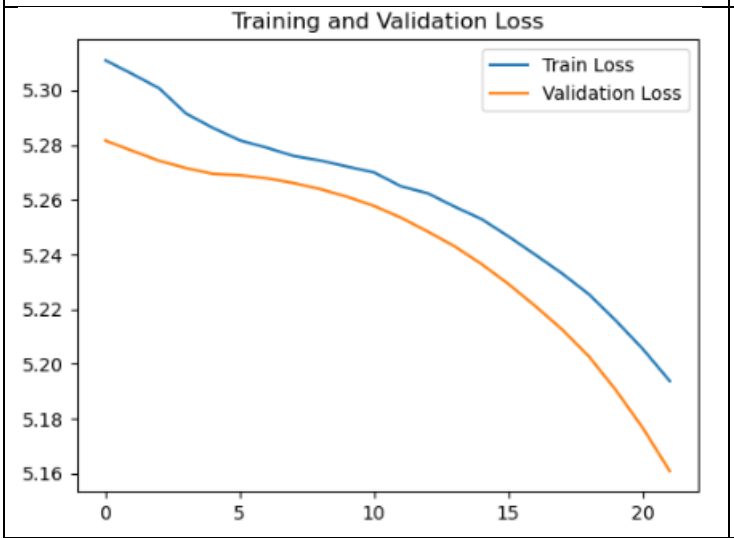


InceptionV3 Results

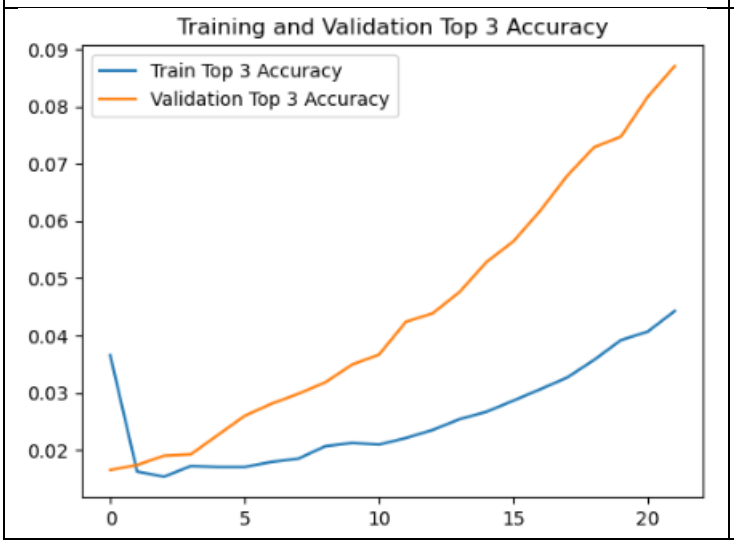
With Default hyper-parameters	With optimal hyper-parameters

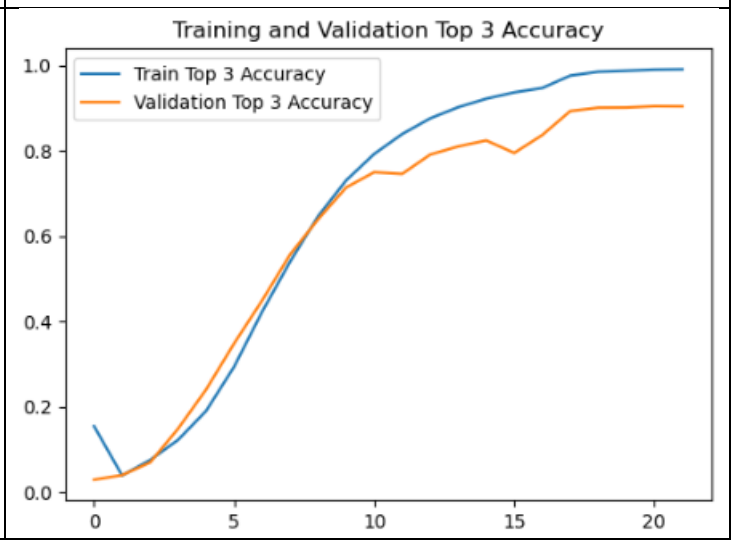

Training and Validation Accuracy
Train Accuracy
Validation Accuracy
0.030
0.025
0.020
0.015
0.010
0.005
0
5
10
15
20

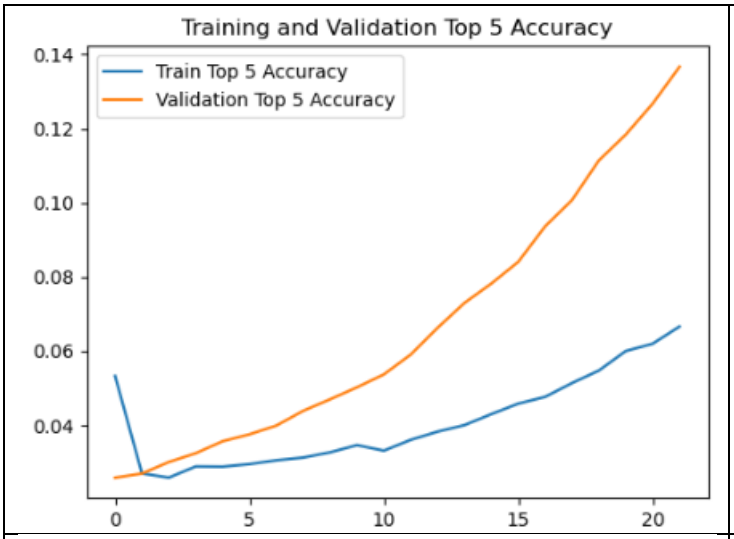

Training and Validation Accuracy
Train Accuracy
Validation Accuracy
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20

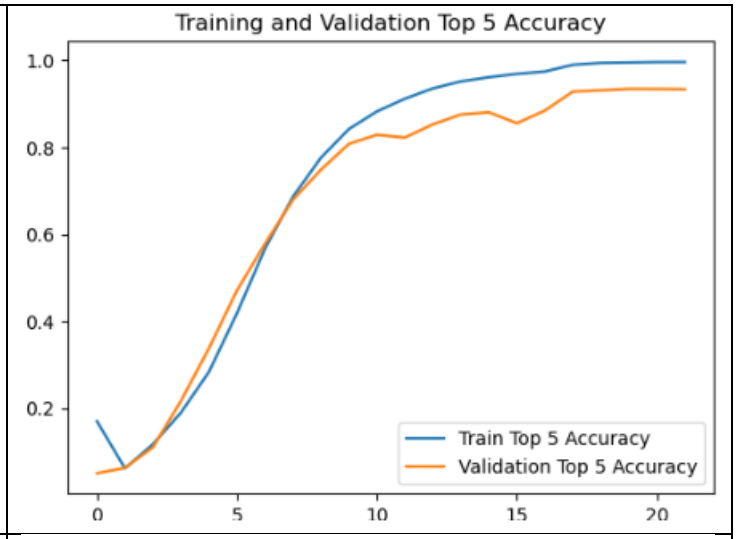

Training and Validation Loss
Train Loss
Validation Loss
5.30
5.28
5.26
5.24
5.22
5.20
5.18
5.16
0
5
10
15
20

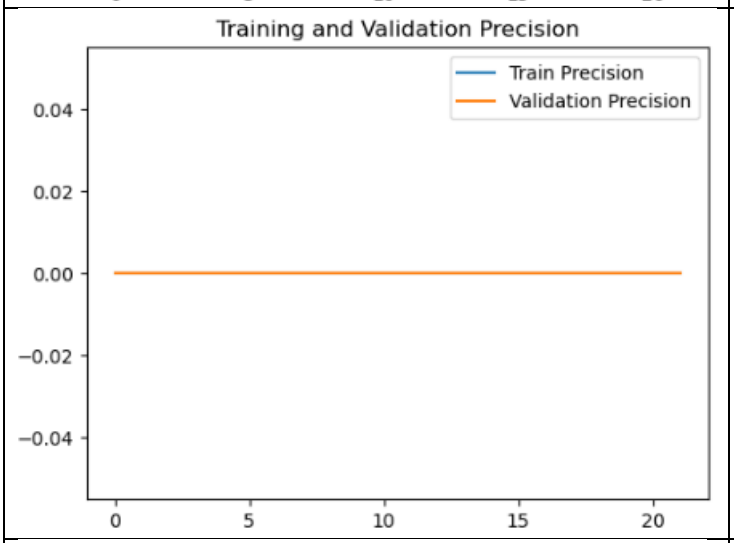

Training and Validation Loss
Train Loss
Validation Loss
8
6
4
2
0
0
5
10
15
20

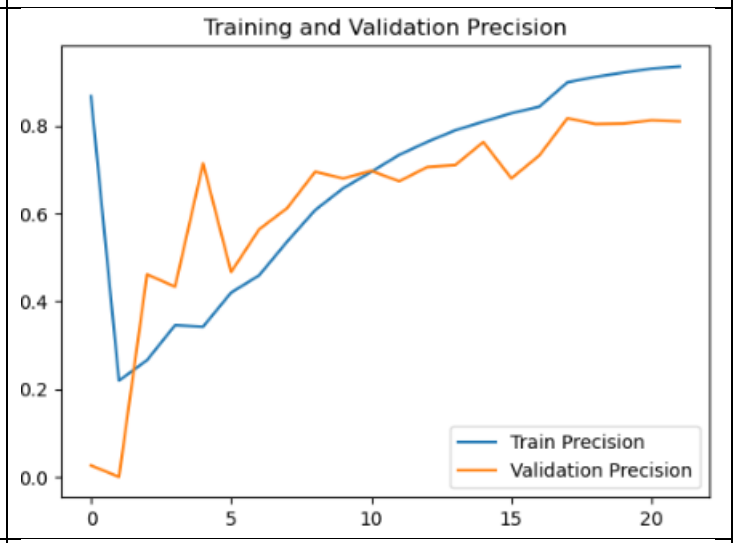

Training and Validation Top 3 Accuracy
Train Top 3 Accuracy
Validation Top 3 Accuracy
0.09
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0
5
10
15
20

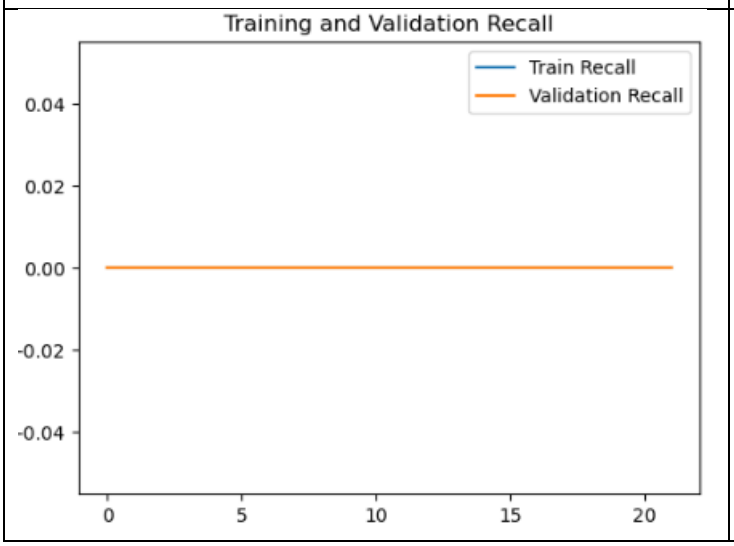

Training and Validation Top 3 Accuracy
Train Top 3 Accuracy
Validation Top 3 Accuracy
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20

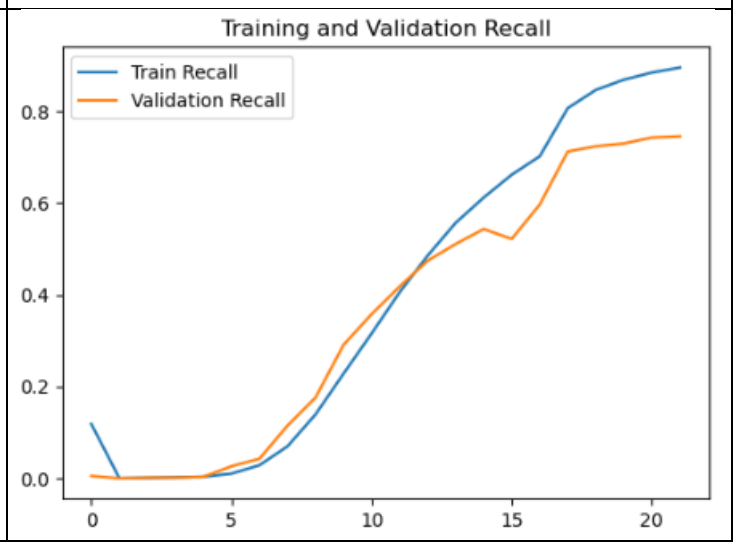

Training and Validation Top 5 Accuracy
Train Top 5 Accuracy
Validation Top 5 Accuracy
0.14
0.12
0.10
0.08
0.06
0.04
0
5
10
15
20

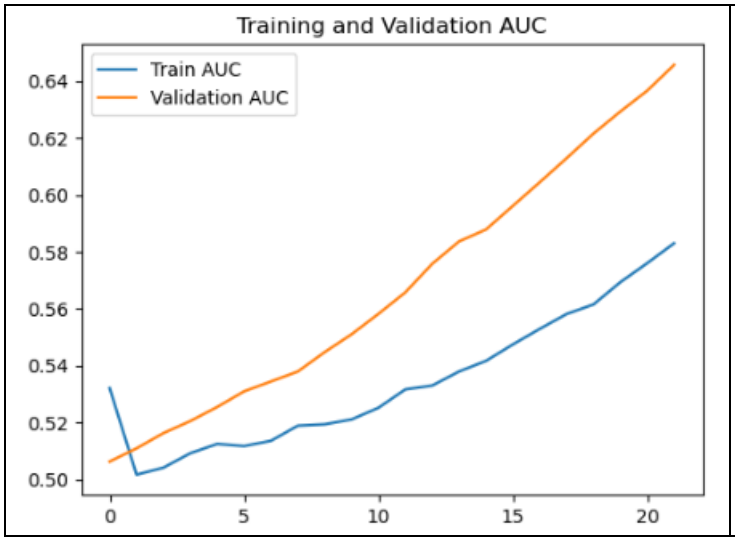

Training and Validation Top 5 Accuracy
Train Top 5 Accuracy
Validation Top 5 Accuracy
1.0
0.8
0.6
0.4
0.2
0
5
10
15
20

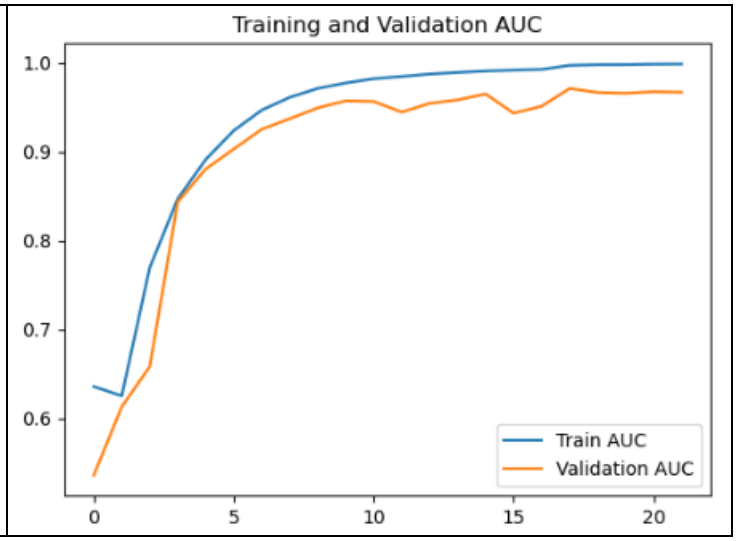

Training and Validation Precision
Train Precision
Validation Precision
0.04
0.02
0.00
−0.02
−0.04
0
5
10
15
20


Training and Validation Precision
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
0.04
0.02
0.00
-0.02
-0.04
0
5
10
15
20

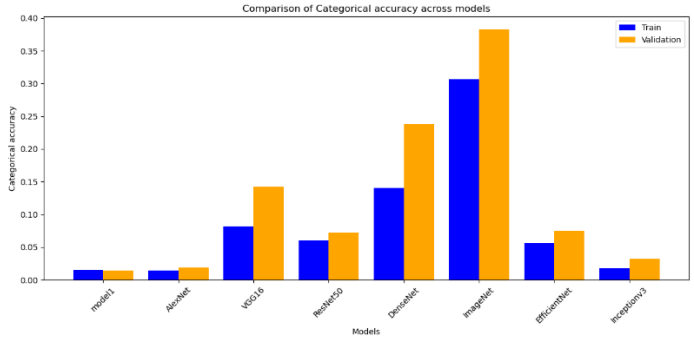

Training and Validation Recall
Train Recall
Validation Recall
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20

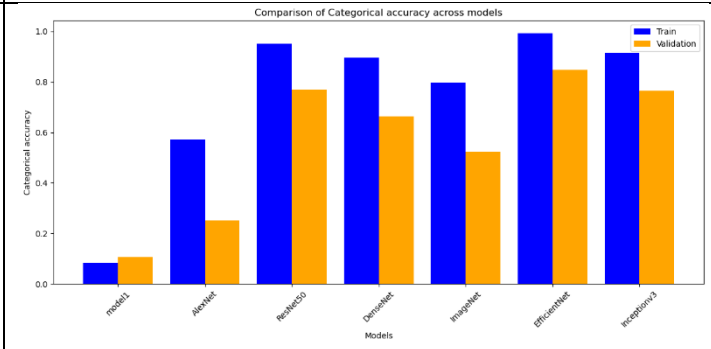


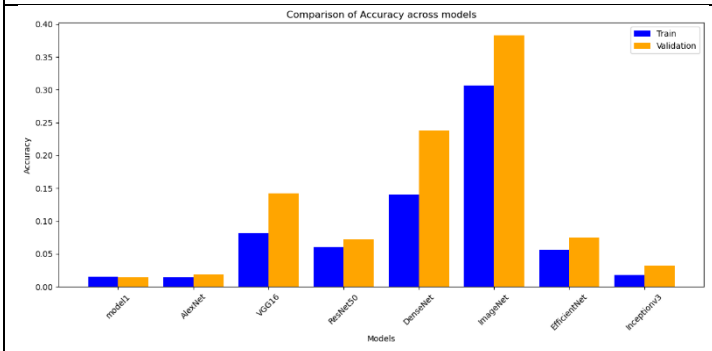


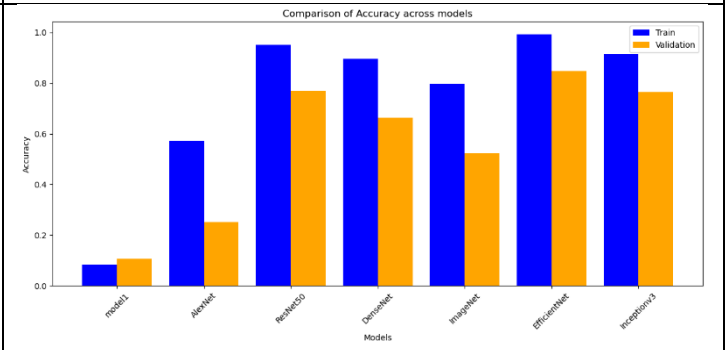
Comparison of models using default vs. optimal hyper-parameters

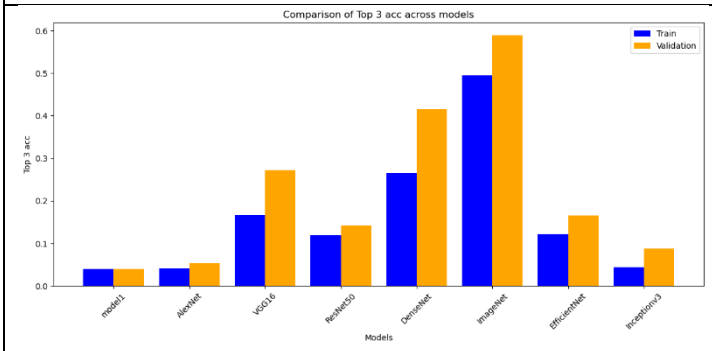
With default hyperparameters	With optimal hyper parameters

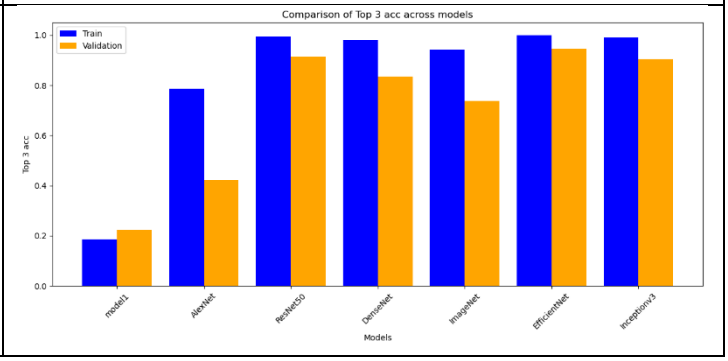


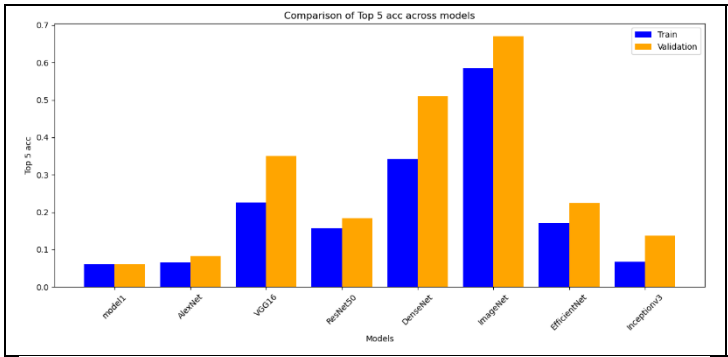


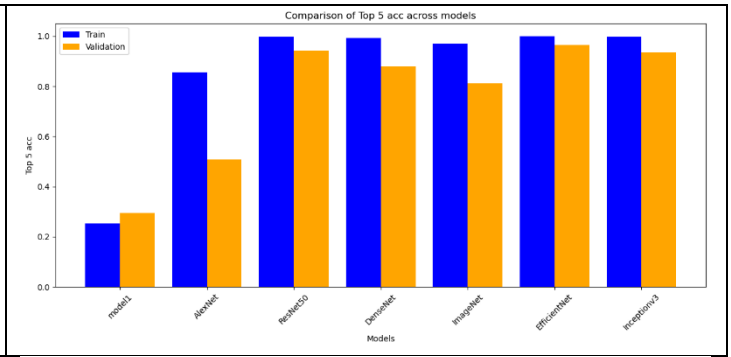


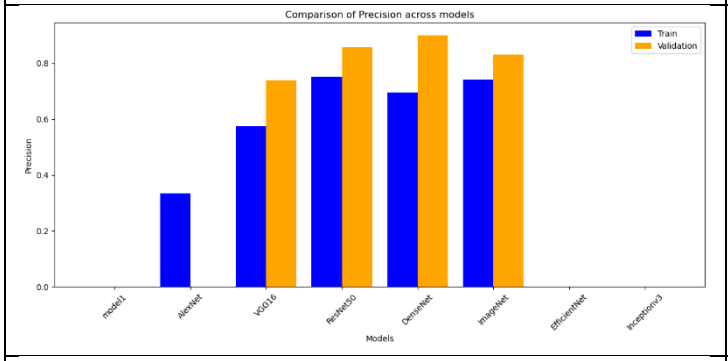


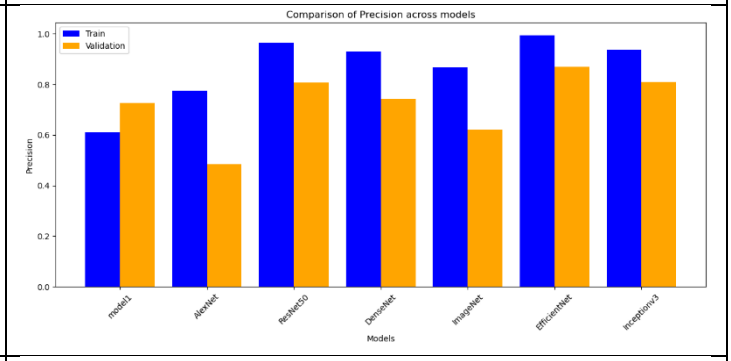


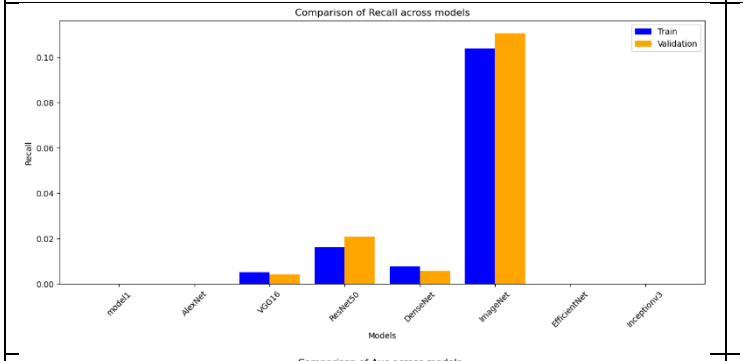


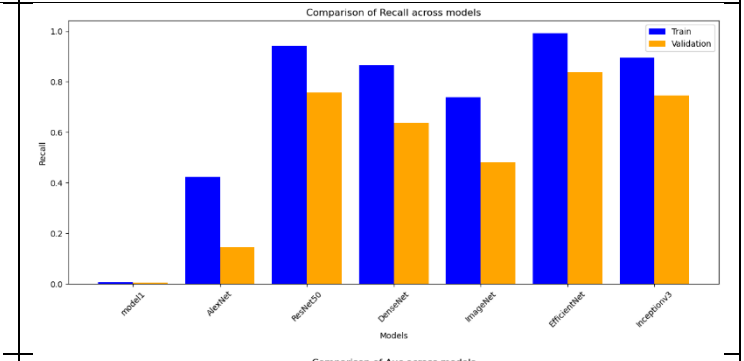

Comparison of Top 5 acc across models
Train
Validation
Top 5 acc
Models
model1
AlexNet
VGG16
ResNet50
DenseNet
ImageNet
EfficientNet
InceptionV3

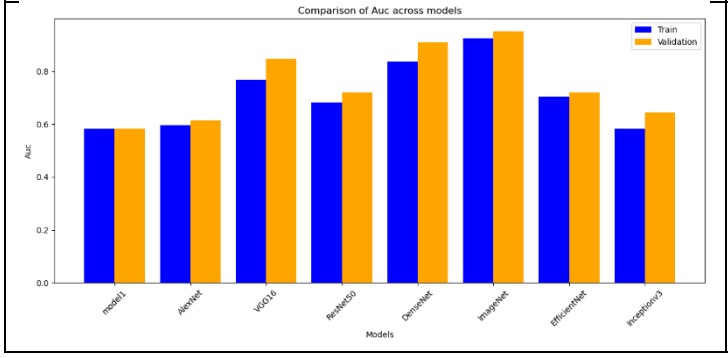

Comparison of Top 5 acc across models
Train
Validation
Top 5 acc
Models
model1
AlexNet
ResNet50
DenseNet
ImageNet
EfficientNet
InceptionV3

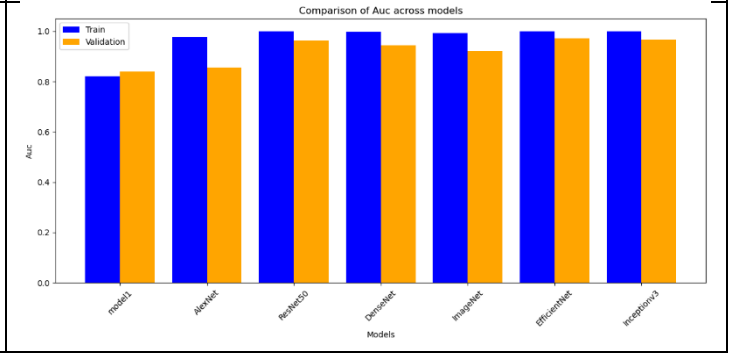

Comparison of Precision across models
Train
Validation
Precision
Models
model1
AlexNet
VGG16
ResNet50
DenseNet
ImageNet
EfficientNet
InceptionV3


Comparison of Precision across models
Train
Validation
Precision
Models
model1
AlexNet
ResNet50
DenseNet
ImageNet
EfficientNet
InceptionV3


Comparison of Recall across models
Train
Validation
Recall
Models
model1
AlexNet
VGG16
ResNet50
DenseNet
ImageNet
EfficientNet
InceptionV3


Comparison of Recall across models
Train
Validation
Recall
Models
model1
AlexNet
ResNet50
DenseNet
ImageNet
EfficientNet
InceptionV3


Comparison of Auc across models
Train
Validation
Auc
Models
model1
AlexNet
VGG16
ResNet50
DenseNet
ImageNet
EfficientNet
InceptionV3


Comparison of Auc across models
Train
Validation
Auc
Models
model1
AlexNet
ResNet50
DenseNet
ImageNet
EfficientNet
InceptionV3

Conclusions

The results of the various models are highly influenced by:

- Data augmentation techniques used
- 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 quire 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.