Python Implementations of Deep Convolutional Neural Network for Fine-Grained Image Classification

Colin (Zirui) Wang
Halıcıoğlu Data Science Institute
Department of Cognitive Science
University of California, San Diego
San Diego, CA 92093
zwcolin@ucsd.edu

Jerry (Yung-Chieh) Chan
Halıcıoğlu Data Science Institute
University of California, San Diego
San Diego, CA 92093
ychan@ucsd.edu

Bingqi Zhou
Halıcıoğlu Data Science Institute
Department of Mathematics
University of California, San Diego
San Diego, CA 92093
biz024@ucsd.edu

Abstract

In this part of the assignment, we built a pipeline for training, validating and testing baseline, custom, vgg16 and resnet18 CNN models with PyTorch on Caltech-UCSD Birds 200 dataset. We used standard techniques to pre-process data and yield a testing accuracy of 0.277 (at epoch 25 with default hyperparameters) and 0.293 (at epoch 50 with default hyperparameters) for the baseline model, 0.500 (at epoch 50) for the custom model, 0.857 after fine-tuning for the vgg16 model, and 0.866 after fine-tuning for the resnet18 model. We also did hyperparameter tuning for models such as learning rate, weight decay, epsilon, amsgrad [1], image standardization mode, and weight initialization mode. Finally, we visualized weight maps and feature maps for the custom, vgg16 and resnet18 model.

1 Introduction

With the advancement of computing power in recent years, people are able to construct complex deep learning models to make predictions on image classification. Using Convolutional Neural Networks (CNN), researchers are competing to reach higher testing accuracy for various image datasets. In this assignment, we used Caltech-UCSD Birds 200 (CUB-200) dataset [2] to test our models. CUB-200 is a dataset annotated with 200 bird species. According to the dataset's description, CUB-200 was created to enable the study of subordinate categorization, which is not possible with other popular datasets that focus on basic level categories.

Early research on the model was not based on pure image classification with deep learning due to constraints in computing power. Branson et. al [3] used this dataset and built an interactive human-computer method to help people without expertise to recognize a certain bird species by minimizing the number of questions asked. Welinder and Perona [4] used this dataset and built a model that utilized crowd-sourcing to help label data in general.

Here, we used this dataset to train, validate, and test our model in predicting bird categories. Because of limitations in computing power, we only used a subset of 20 categories as our dataset. Each

category includes 60 images. We used 27 images from each category to train our model, 3 images from each category to validate our model, and 30 images to test our model. Table 1 shows how we split train-validation-test data. We converted each image to RGB scale, resize it to (224×224) pixels, and standardized in three different methods. Table 2 shows image standardization methods. We used single-precision floating-point format to represent image data. We also performed image augmentation as our dataset contains relatively few images. Table 3 describes how we augmented our training data, and Table 4 describes how we transformed out test data.

CATEGORY	TRAIN SIZE	VALIDATION SIZE	TEST SIZE
Frigatebird	27	3	30
Northern_Fulmar	27	3	30
Gadwall	27	3	30
American_Goldfinch	27	3	30
European_Goldfinch	27	3	30
Boat_tailed_Grackle	27	3	30
Eared_Grebe	27	3	30
Horned_Grebe	27	3	30
Pied_billed_Grebe	27	3	30
Western_Grebe	27	3	30
Blue_Grosbeak	27	3	30
Evening_Grosbeak	27	3	30
Pine_Grosbeak	27	3	30
Rose_breasted_Grosbeak	27	3	30
Pigeon_Guillemot	27	3	30
California_Gull	27	3	30
Glaucous_winged_Gull	27	3	30
Heermann_Gull	27	3	30
Herring_Gull	27	3	30
Ivory_Gull	27	3	30
TOTAL	**540**	**60**	**600**

Table 1: Dataset Statistics

VARIABLES/METHOD	Z-SCORE	IMAGENET	MIN-MAX
R_MEAN	0	0.485	ORIGINAL_MEAN/225
G_MEAN	0	0.456	ORIGINAL_MEAN/225
B_MEAN	0	0.406	ORIGINAL_MEAN/225
R_STD	1	0.229	ORIGINAL_STD/225
G_STD	1	0.224	ORIGINAL_STD/225
B_STD	1	0.225	ORIGINAL_STD/225

Table 2: Standarization Methods & Statistics

Augmentation	Parameter
Resize	256
Center Crop	224
Random Horizontal Flip	0.5
Random Vertical Flip	0.5
Random Rotation	30
Normalize	See Table 2

Table 3: Train Data Augmentation

Transformation	Parameter
Resize	256
Center Crop	224
Normalize	See Table 2

Table 4: Test Data Transformation

We used 2-folds cross validation to train and validate all of our models (baseline, custom, vgg16 [5], resnet18 [6]). Training and validation statistics from all plots for accuracy and loss within this assignment in each epoch are the average of 2-folds. We prevented overfitting by saving the model with lowest validation loss and changing weight decay for the custom, vgg16 and resnet18 model. We also performed hyperparameter tuning over these three models to enhance accuracy for the test set. Finally, we visualized weight maps and feature maps for them.

2 Related Work

Developments in deep learning during recent years often involve very deep convolutional neural networks to detect features of images to make accurate predictions. Here, besides our baseline and custom model to train, validate and test on CUB-200 dataset, we also included two pretrained models to help determine our performance. Vgg16 [5] developed by Simonyan and Zisserman aims to enhance the accuracy of image predictions by pushing the depth to 16-19 weight layers. The other model resnet18 [6] developed by He et. al aims to use a residual learning framework to ease the training of networks that are substantially deeper than those used previously, such as vgg16.

There are also several deep learning models that are developed based on the characteristics of the CUB-200 dataset. Huang et. al [7] proposed a novel Part-Stacked CNN architecture that explicitly explains the fine-grained recognition process by modeling subtle differences from object parts, and proved its model to be effective using the CUB-200 dataset. Zhang et. al [8] proposed a unified framework based on a mixture of experts, and promoted diversity among experts by combing an expert gradually-enhanced learning strategy and a Kullback-Leibler divergence based constraint. Experiments from their work showed that their resulting model improves the classification performance on the CUB-200 dataset. Korsch et. al [9] further improved the model by Zhang et. al by integrating a Fisher vector encoding of part features into convolutional neural networks, reaching an accuracy of 0.9095 in predicting birds categories from CUB-200 dataset.

3 Models

3.1 Baseline Model Description

The baseline model used a simple CNN structure to learn features of images and make predictions. Specifically, this model starts with 3 convolutional layers of increasing number of channels but same receptive field size (3×3). Then, it has a max pooling layer of field size (3×3) following another convolutional layer with 256 output channels and a stride of 2. After that, it has an adaptive average pooling layer that transforms the last 256 output channels into 256 features. The forward function will flatten these 256 features and pass them into two fully connected layers. In the first fully connected layer, 256 features will be passed into a dropout layer and them mapped into 1024 features. In the second fully connected layer, these 1024 features will be mapped into the number of classes, which is 20 based on this assignment. The model used ReLU as its activation function, cross entropy as its loss function, an Adam optimizer [10], uniform xavier initialization for all the weights of linear and convolutional layers, and 0 bias for these layers. Table 5 shows the architecture of the baseline model.

LAYER	IN_SHAPE	OUT_SHAPE	PARAM #
Conv2d	[-1, 3, 224, 224]	[-1, 64, 222, 222]	1,792
BatchNorm2d	[-1, 64, 222, 222]	[-1, 64, 222, 222]	128
ReLU	[-1, 64, 222, 222]	[-1, 64, 222, 222]	0
Conv2d	[-1, 64, 222, 222]	[-1, 128, 220, 220]	73,856
BatchNorm2d	[-1, 128, 220, 220]	[-1, 128, 220, 220]	256
ReLU	[-1, 128, 220, 220]	[-1, 128, 220, 220]	0
Conv2d	[-1, 128, 220, 220]	[-1, 128, 218, 218]	147,584
BatchNorm2d	[-1, 128, 218, 218]	[-1, 128, 218, 218]	256
ReLU-9	[-1, 128, 218, 218]	[-1, 128, 218, 218]	0
MaxPool2d	[-1, 128, 218, 218]	[-1, 128, 72, 72]	0
Conv2d	[-1, 128, 72, 72]	[-1, 256, 35, 35]	295,168
BatchNorm2d	[-1, 256, 35, 35]	[-1, 256, 35, 35]	512
ReLU-13	[-1, 256, 35, 35]	[-1, 256, 35, 35]	0
AvgPool2d	[-1, 256, 35, 35]	[-1, 256, 1, 1]	0
Linear	[-1, 256, 1, 1]	[-1, 1024]	263,168
Dropout	[-1, 1024]	[-1, 1024]	0
ReLU	[-1, 1024]	[-1, 1024]	0
Linear	[-1, 1024]	[-1, 20]	20,500

Table 5: Baseline Model Architecture. Here, the first dimension of input/output shape is the batch size. -1 will be replaced according to how many data samples are present in each batch.

3.2 Custom CNN Architectural Details

In this part, we presented our custom model as a variant of Alexnet [11]. Alexnet was a CNN model designed in 2012 for image classification. Many people used it on general image classification such as everyday items (cars, people, apple, numbers, etc). However, our dataset consists of bird with different species. Image classification in this dataset is harder in original Alexnet design because Alexnet cannot deal well with fine-grained image classification. After we explored on the Internet for potential solutions, we found a paper written by Cai et. al [12] where their team used super-resolution CNN layers to transform the image and feed both the original image and transformed data into the first CNN layer of Alexnet. Since Alexnet already has 5 layers, we added 1 super-resolution CNN layer into our model to satisfy the requirements of our assignment. Figure 1 shows how we modified the original Alexnet.

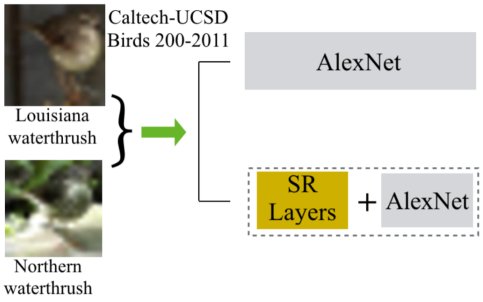

Figure 1: Basic architecture of our model: how we modified Alexnet to make it perform better on fine-grained images

Besides changes in convolutional layer architecture to ensure that our varient can better deal with fine-grained images, we also added batch normalization layers after each convolutional layers in the original Alexnet to enhance the model stability. Our specific model architecture is presented in Table 6 and Table 7

3.3 Custom Model Architectural Description

Our final model starts with a convolutional layer that outputs features having the same size of our images. We named this layer as SConv2d for its capability of creating super-resolution. We performed element-wise summation of the original image (the residual) and the output of the layer as the input for second layer. Other design features are similar to Alexnet. Starting from second convolutional layer, we have decreasing size of receptive field (from 11 to 5 to 3) and decreasing strides (from 4 to 2 to 1) to capture both global and local features. We add paddings to the last few convolutional layers to prevent from loss of information as the length and width have become very small. We added max-pooling layers after early convolutional layers to reduce the complexity of our model. After the last convolutional layers, we added a average pooling layer to flatten the results, and send these results into our classifier network. Our classifier network consists of two dropout layers to prevent the model from overfitting at early epochs. We have three linear layers that map 9216 features into 4096 features and finally 20 features. Our final model used batch normalization after each convolutional layer to enhance the stability, and ReLU as the activation function for our entire model. We have tried adding a softmax layer as activation function for the last layer of our network, but our results proved that this layer may not be necessary (discussed in Piazza Post @593_f3). As a result, we used argmax to calculate our accuracy. The final model used cross-entropy as loss function, xavier uniform initialization for all convolutional and linear layers, and an Adam optimizer with a scheduled learning rate.

For model history, we have tried removing the average pooling layer and adding to additional convolutional layers based on the baseline model, but this led to an exploded number of parameters in the fully connected layer, where GPU cannot handle these parameters given limited memory. We also tried the positioning of two max-pooling layers, and we found that max-pooling layers worked best between two convolutional layers of increasing number of channels. This probably helped transform our data from depth in size to depth in channels.

3.4 Custom CNN Architectural Encapsulation

LAYER	IN_SHAPE	OUT_SHAPE	K	S	P	D	PARAMS
(1) SConv2d	[-1, 3, 224, 224]	[-1, 3, 224, 224]	5	0	2		228
BatchNorm2d	[-1, 3, 224, 224]	[-1, 3, 224, 224]					6
ReLU	[-1, 3, 224, 224]	[-1, 3, 224, 224]					0
(2) Conv2d	[-1, 32, 224, 224]	[-1, 64, 54, 54]	11	4	2		23,296
BatchNorm2d	[-1, 64, 54, 54]	[-1, 64, 54, 54]					128
ReLU	[-1, 64, 54, 54]	[-1, 64, 54, 54]					0
MaxPool2d	[-1, 64, 54, 54]	[-1, 64, 26, 26]	3	2	0		0
(3) Conv2d	[-1, 64, 26, 26]	[-1, 192, 11, 11]	5	2	0		307,392
BatchNorm2d	[-1, 192, 11, 11]	[-1, 192, 11, 11]					384
ReLU	[-1, 192, 11, 11]	[-1, 192, 11, 11]					0
MaxPool2d	[-1, 192, 11, 11]	[-1, 192, 5, 5]	3	2	0		0
(4) Conv2d	[-1, 192, 5, 5]	[-1, 384, 5, 5]	3	1	1		663,936
BatchNorm2d	[-1, 384, 5, 5]	[-1, 384, 5, 5]					768
ReLU	[-1, 384, 5, 5]	[-1, 384, 5, 5]					0
(5) Conv2d	[-1, 384, 5, 5]	[-1, 256, 5, 5]	3	1	1		884,992
BatchNorm2d	[-1, 256, 5, 5]	[-1, 256, 5, 5]					512
ReLU	[-1, 256, 5, 5]	[-1, 256, 5, 5]					0
(6) Conv2d	[-1, 256, 5, 5]	[-1, 256, 5, 5]	3	1	1		590,080
BatchNorm2d	[-1, 256, 5, 5]	[-1, 256, 5, 5]					512
ReLU	[-1, 256, 5, 5]	[-1, 256, 5, 5]					0
AvgPool2d	[-1, 256, 5, 5]	[-1, 256, 6, 6]					0

Table 6: Custom Model Architecture. -1 will be replaced according to how many data samples are present in each batch. K, S, P, D represents Kernel size, Stride, Padding, and Dropout probability respectively.

LAYER	IN_SHAPE	OUT_SHAPE	K	S	P	D	PARAMS
Flatten	[-1, 256, 6, 6]	[-1, 9216]					0
Dropout	[-1, 9216]	[-1, 9216]				.5	0
Linear	[-1, 9216]	[-1, 4096]					37,752,832
ReLU	[-1, 4096]	[-1, 4096]					0
Dropout	[-1, 4096]	[-1, 4096]				.5	0
Linear	[-1, 4096]	[-1, 4096]					16,781,312
ReLU	[-1, 4096]	[-1, 4096]					0
Linear	[-1, 4096]	[-1, 20]					81,940

Table 7: Custom Model Classifier Architecture. -1 will be replaced according to how many data samples are present in each batch. K, S, P, D represents Kernel size, Stride, Padding, and Dropout probability respectively.

3.5 VGG16 & ResNet18 Architecture

We download the pretrained ResNet and VGG model from torchvision.models. To adapt the model to our data, we remove the last fc layer from both model and added our own fc layer which has the same number of units as the category of our dataset.

3.5.1 VGG model

For this transfer learning model, we are using pretrained VGG16 with batch normalization. Vgg16 is a model compose of a sequence of modules, which includes two to three convolution layers following by a maxpooling layer, with a adaptive average pooling and three linear layers at the end. The kernel size and number of channels for all convolution layers in a module increase by a factor of 2 than the previous module. The model has 1000 output features each representing a class (label). The model is one of the state-of-the-art model in the ImageNet challenge. Its success shows that a very deep convolution network with pooling layers can achieve a low error on image classification. Batch normalization is a technique of adding trainable layer that normalize each batch between hidden layers. The technique can booth the training and enhance model performance.

The origin output layer (the last layer in the classifier) is a linear layer with 4096 in features and 1000 out features. We remove the layer and add a linear layer that takes in 4096 features and output 20 features and initialized it with xavier uniform weight initialization to fit our application.

3.5.2 ResNet model

For this transfer learning model, we are using pretrained ResNet18. ResNet is also another deep convolution model for image classification. It is composed of multiple basicblocks. Each basic blocks contains convolution layers, ReLU activation layers, and batch normalization layers. The output of each basicblocks is the output of those layers plus the input, which enable the gradient to be passed further back to the model in back propagation. In ResNet18, there are 4 sets of layers, each contains 2 basicblocks that contains 2 convolution layers with ReLU and Batch Normalization between them. The convolution parts of the network is followed by an adaptive average pooling layer and a fully connected linear layer for classification. The model structure is designed to avoid the problem of vanishing gradients in deep networks. This idea opens up the probability of training very deep models.

The origin output layer (the last layer) is a linear layer with 512 in features and 1000 out features. We remove the layer and add a linear layer that takes in 512 features and output 20 features and initialized it with xavier uniform weight initialization to fit our application.

4 Experiments

4.1 Baseline Model Experimentation

Since we're asked not to change any hyperparameters even including those that are responsible for preventing the model from overfitting (reference: Piazza Post @568_f1), this model starts to overfit since 10th epoch. Overfit began at early epoch because our training set is relatively small that the model can easily *remember* these images. However, since we save our model not based on training loss but validation loss, we ensured that the saved model has the best weights for the *unseen* data. Table 8 shows the default hyperparameters that we used to train, validate, and test the model. Table 9 shows the model's accuracy on test data after 25, 50, and 100 training epochs. Figure 2, 3 and 4 shows plots for training and validation accuracy and loss over 25, 50, and 100 epochs. Please note that we have been consistently using 2-Fold cross validation during the training and validation process. Calculations of training and validation statistics over each epoch are based on the average value over the 2-Folds.

HYPERPARAMETER	VALUE
Batch Size	20
Learning Rate	1e-4
Weight Initialization	Xavier Uniform
Weight Decay	0
Epsilon	1e-8
Folds	2
AMS Grad	False
Image Standardization	IMAGENET

Table 8: Baseline Model Hyperparameter

EPOCH	ACCURACY
25	0.277
50	0.293
100	0.302

Table 9: Baseline Model Accuracy on Test Set

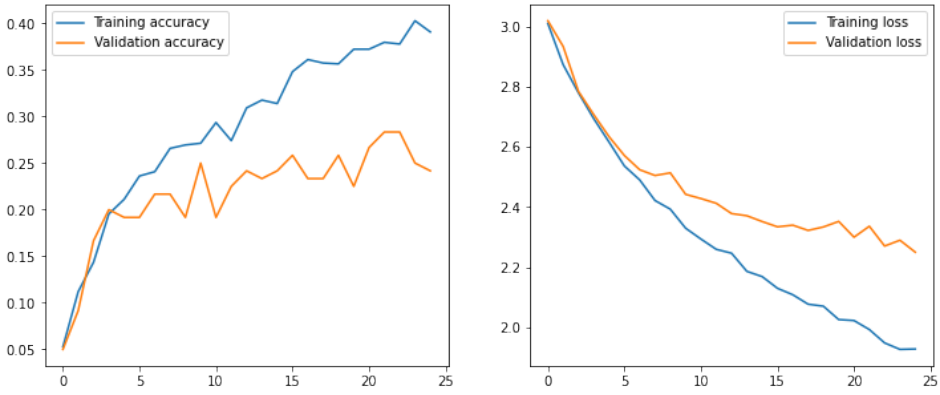

Figure 2: Training and validation accuracy & loss on baseline model with 25 epochs.

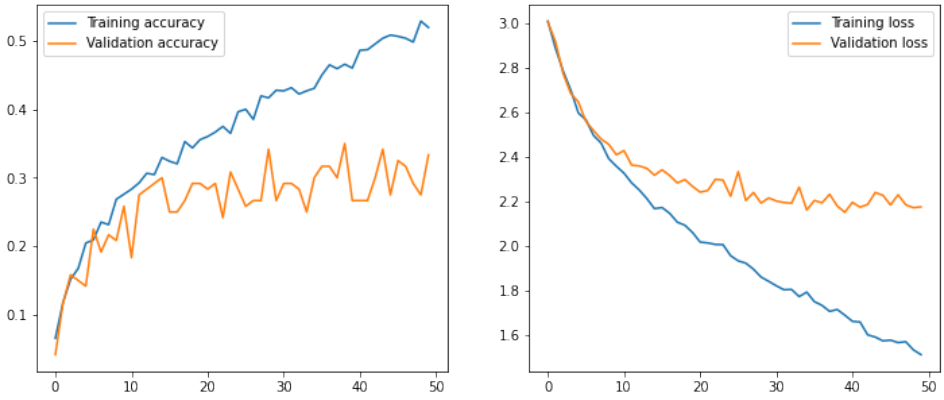

Figure 3: Training and validation accuracy & loss on baseline model with 50 epochs.

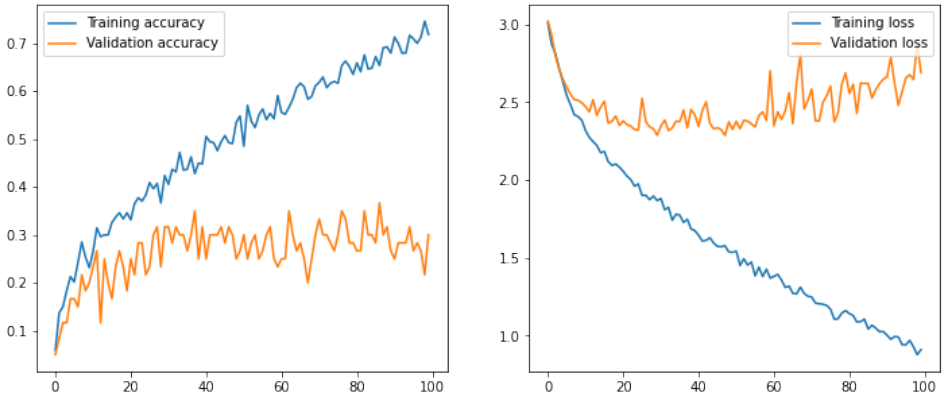

Figure 4: Training and validation accuracy & loss on baseline model with 100 epochs.

4.2 Custom Model Experimentation

Based on our custom model and the best set of hyperparameters, we achieved an accuracy of 0.500 for this dataset. Table 10 shows the hyperparameters that we used to achieve the highest accuracy, Table 11 shows the accuracy of different experiments that involve changing model architecture, tuning hyperparameters, changing data augmentation. Figure 5 shows the training and validation accuracy and loss for the best testing set accuracy (0.500).

HYPERPARAMETER	VALUE
Batch Size	20
Learning Rate	1e-4
Weight Initialization	Xavier Uniform
Weight Decay	5e-4
Epsilon	1e-8
Folds	2
AMS Grad	False
Image Standardization	IMAGENET
Image Augmentation	True

Table 10: Custom model hyperparameters with best performance

Experimentation	Accuracy	Overfit
VA + 1SR + HT + AG	**0.500**	**some**
VA + 1EC + HT + AG	0.486	some
BL + 2EC + HT + AG	0.389	slight
VA + 1SR + HT	0.348	significant
VA + 1EC + HT	0.322	significant
BL + 2EC + HT	0.312	some
VA + 1SR + AG	0.449	some
VA + 1EC+ AG	0.428	some
BL + 2EC + AG	0.366	slight
VA + 1SR	0.318	significant
VA + 1EC	0.315	significant
BL + 2EC	0.303	some

Table 11: Custom model experimentation with 50 epochs each. VA represents Variant Alexnet, BL represents Baseline model. SR represents Super-Resolution. EC represents extra convolutional layer. HT represents if hyperparameter tuning is done, and AG means if data augmentation is done. Here, Variant Alexnet with Super Resolution is our final model. Note that this net is not a pretrained version of Alexnet, and some layers besides the addition of SR Layer in original Alexnet have been modified to better fit our data.

We have three custom model being experimented. Alexnet with SR Layer has an architecture that creates a Super-Resolution version of an image and feeds both the output of the SR layer and the input into the second layer, where process starts here is Alexnet-alike. Alexnet with 1 Extra Conv has an architecture that starts with convolutional layer of receptive field size of 3 and a channel of 32. This aims to help capture fine details about the image before sending it into the second layer. Processes starting from the second layer is Alexnet-alike. Baseline with 2 Extra Conv has the same architecture as the baseline model except that 2 more convolutional layers are added to the end.

Here, augmentation is an approach to increase the variance of input images to reduce overfit problems. Augmentation details can be found in Table 3. Tuning is an approach to further increase the accuracy of our model by changing hyperparameters like weight decay, learning rate, weight initialization, etc. However, given limited size of our dataset, hyperparameter tuning had relatively little effect compared to that brought by data augmentation. We have three levels for overfitting: significant, some, and slight. When there is significant overfitting, it means that the training accuracy over certain epochs reached 1.00 and never goes down again, but validation accuracy is relatively low. When there is some overfitting, it means that the training accuracy over certain epochs is visibly higher than validation accuracy, but still makes space for validation accuracy to go up. When there is slight overfitting, it means that training and validation accuracy are similar over the entire training process.

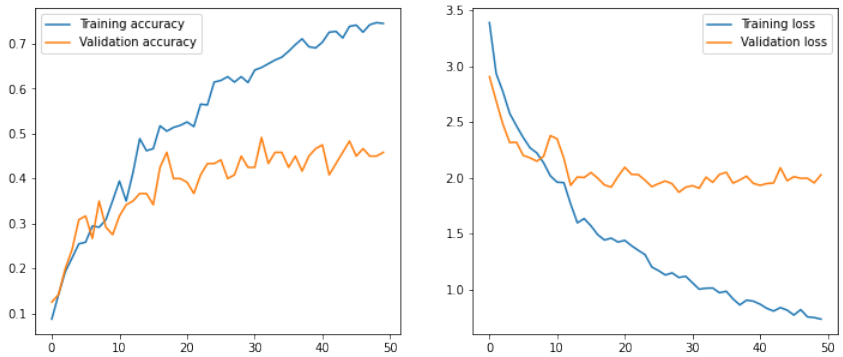

Figure 5: Average training and validation accuracy & loss on custom model with data augmentation and best set of hyperparameters for 50 epochs over 2 folds

4.3 VGG & ResNet Experimentation

4.3.1 Freezing Pretrained model

In the first step of out experimentation, we freeze the pretrained layers by not applying optimizer on it and doesn't back prop the gradient to them. The model is named freezed VGG and freezed ResNet. We train both model under the following hyperparameters in Table 12:

HYPERPARAMETER	VALUE
Batch Size	20
Learning Rate	1e-3
Weight Initialization	Xavier Uniform
Weight Decay	0
Epsilon	1e-8
Folds	2
AMS Grad	False
Epoch	25
Image Standardization	IMAGENET

Table 12: Hyperparameter for freezed VGG and freezed ResNet

The results are presented in Figure 6 and Figure 7. The accuracy is presented in Table 13

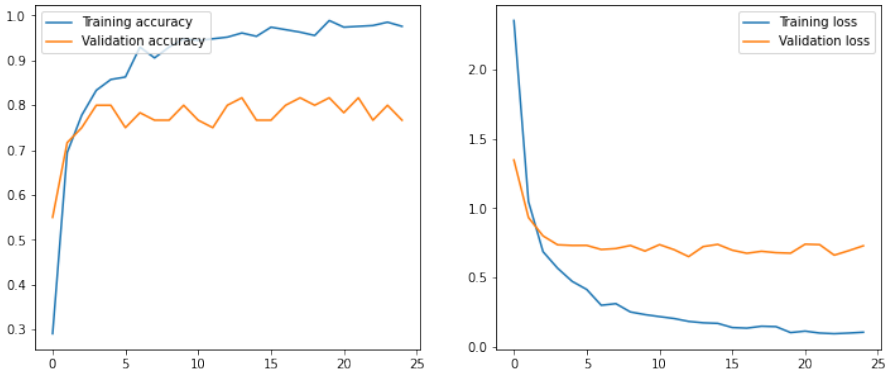

Figure 6: Training and validation accuracy & loss on freezed VGG model.

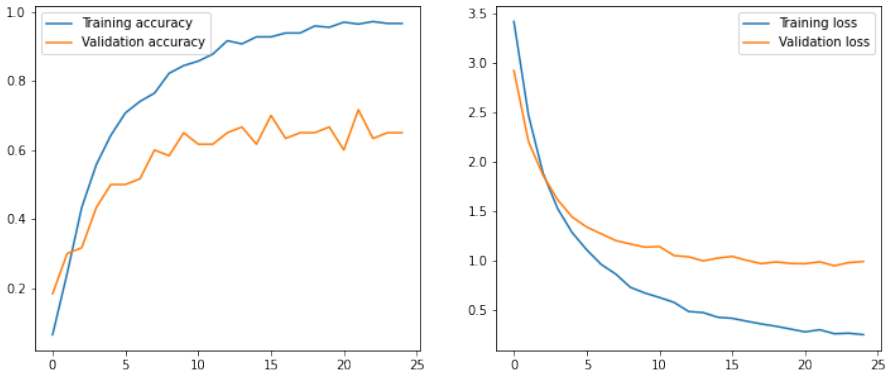

Figure 7: Training and validation accuracy & loss on freezed ResNet model.

MODEL	ACCURACY	LOSS
Freezed VGG	0.747	0.771
Freezed ResNet	0.698	0.879

Table 13: Freezed VGG and Freezed ResNet Model Accuracy and Loss on Test Set

Freezed VGG model scores a high accuracy at 74% while Freezed ResNet scores 69%. We can see that there are some overfitting issues in both models. We thinks this is caused by the small training dataset (540 images) compare to a large input features to the output layer (4096 for VGG model and 512 for ResNet model). We will try to solve this problem in next section.

4.3.2 Reduce overfitting

We have tried multiple techniques to reduce overfitting, including dropout layer, l2, schedule learning, and dataset augmentation. We test our model with different combination of those technique. The following Table 14 are the configuration we used to get the best preforming model:

METHOD	DESCRIPTION
Image Augmentation	Randomized cropping, horizontial flip, vertical flip, and rotation.
Scheduled Learning Rates	Learning rate decrease by a factor of 0.5 for every 4 epochs
L2 Penalty	Set the weight decay of the optimizer to 1e-3

Table 14: Overfitting Alleviation Methods

We used the transforms module from torchvision to transform the input data while training. Therefore, the number of images feed to the model in a single epoch will be the same. We double the number of epochs since each input image are getting transformed randomly in each epoch.

All the hyperparameter are the same as last section except for the parameters mentions above. The result is presented in Figure 8, Figure 9, and Table 15.

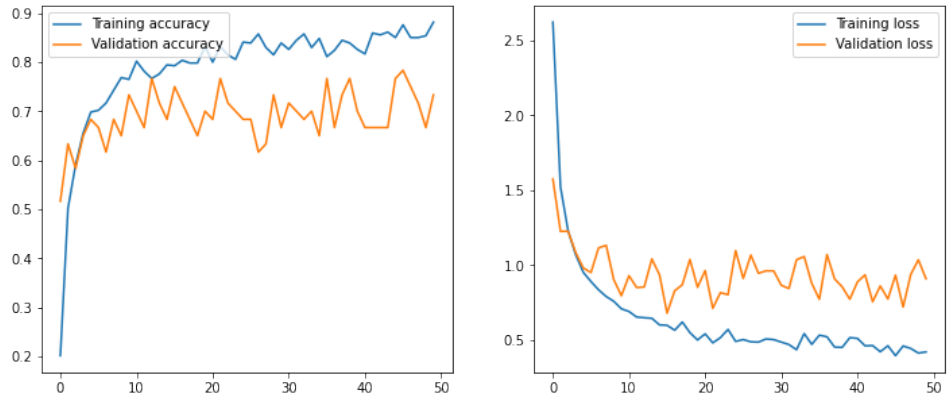

Figure 8: Training and validation accuracy & loss on freezed VGG model with augmented dataset.

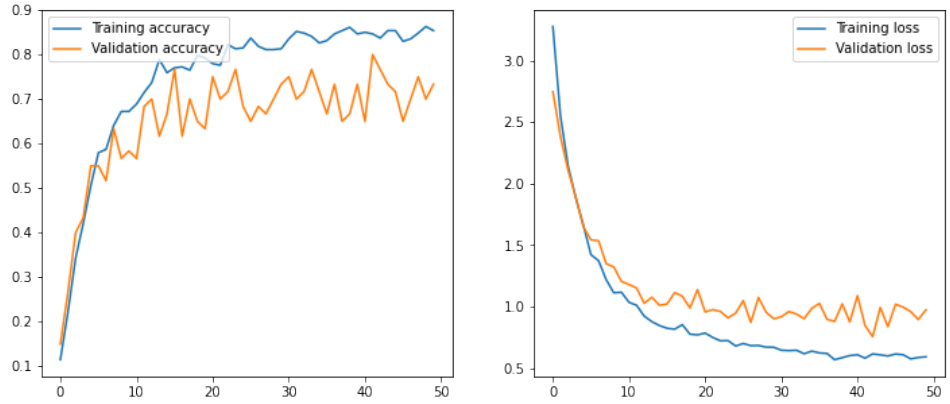

Figure 9: Training and validation accuracy & loss on freezed ResNet model with augmented dataset.

MODEL	ACCURACY	LOSS
Freezed VGG	0.793	0.605
Freezed ResNet	0.752	0.800

Table 15: Freezed VGG and Freezed ResNet Model Trained with Augmented Dataset Accuracy and Loss on Test Set

The accuracy increases by 5%. As we can see from the graphs above, overfiting is reduced by a lot. Because of it excellent performance, we are going to stick with this setting in the next section.

4.3.3 Fine-tuning

In this experiment, we propagate the gradient back to the pretrained part of the model. The pretrained parameters are given a lower learning rate since we are just fine-tuning them. After some testing, we find that a high learning rate will cause server overfitting since the model is very complicated compare to use dataset. We decrease the number of epochs back to 25 to prevent overfitting. The following Table 16 is the hyperparameter used in this section:

HYPERPARAMETER	VALUE
Batch Size	20
Learning Rate (last fc)	1e-3
Learning Rate (pretrain)	1e-4
Weight Initialization	Xavier Uniform
Weight Decay	1e-3
Epsilon	1e-8
Folds	2
AMS Grad	False
Epoch	25
Image Standardization	IMAGENET

Table 16: Hyperparameter for Fine-Tuning VGG and ResNet

The results are presented in Figure 10, Figure 11, and Table 17

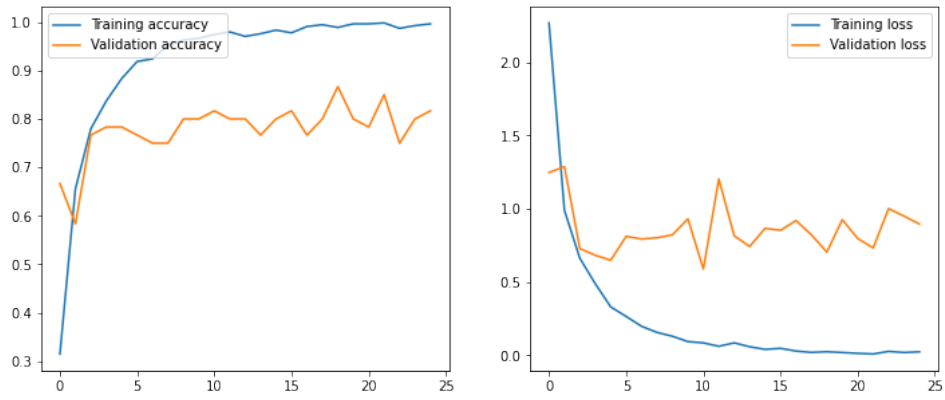

Figure 10: Training and validation accuracy & loss on fine-tuning VGG model.

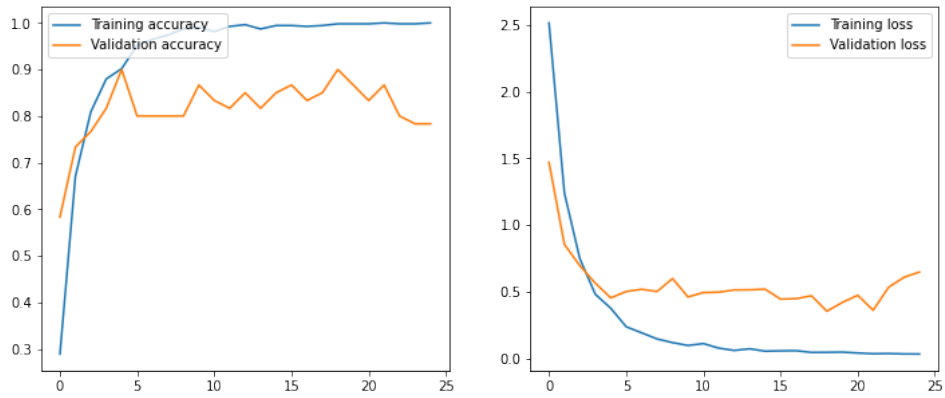

Figure 11: Training and validation accuracy & loss on fine-tuning ResNet model.

MODEL	ACCURACY	LOSS
Freezed VGG	0.866	0.460
Freezed ResNet	0.857	0.506

Table 17: Freezed VGG and Freezed ResNet Model Accuracy and Loss on Test Set

Both model achieve a descent accuracy above 85% and a 0.2 drops on losses. More discussion of the results will be included in the discussion section.

5 Feature Map and Weights Analysis

5.1 Convolutional Layer Weight Maps

5.1.1 Custom Model

DISCLAIMER: Because the first convolutional layer of our custom model is used to create super resolution, we are unable to visualize the weights with given architecture. Therefore, in this part, we removed the SR convolutional layer and the results are from the first convolutional layer without the SR layer.

The first convolution layer (non-super-resolution) of custom model has 64 11x11 kernel sets. There weights are scaled back to (0, 255), kernels in the same set are combined.(Each corresponding to R, G, B), and presented in the figure below.

The weights map seems very random. We think this is because there are not enough training data. The pretrained model (in the following sections) are trained with ImageNet dataset which has 14 million images. Therefore our weight maps aren't as representative as the one from VGG and ResNet. But still we're able to yield an accuracy of 0.500 with this weight map.

Figure 12: VGG weight maps

5.1.2 VGG Model

The first convolution layer of VGG model has 64 3x3 kernel sets. There weights are scaled back to (0, 255), kernels in the same set are combined.(Each corresponding to R, G, B), and presented in the Figure 13 below. Following are the visual observations:

- Kernel (2, 8) and (5, 7) detects vertical image boundaries. Kernel (2, 8) reacts to image patches that is lighter on the left side of the boundaries and darker on the right side. Kernel (5, 7) does the opposite.
- Kernel (5, 2) and (4, 8) detects horizontal image boundaries. Kernel (5, 2) reacts to image patches that is lighter on the upper side of the boundaries and darker on the lower side. Kernel (4, 8) does the opposite.
- Kernel (3, 3) and (4, 5) detects dots. Kernel (3, 3) reacts to a light dot and (4, 5) reacts to a dark dot
- Kernels like (2, 3), (3, 2) and (2, 5) detects the corresponding colors. The react when the input patch has similar colors as themselves.
- Kernels (7, 1) and (6, 1) contains all the different colors. I'm guessing that it detects the brightness of the image patches. Suppose near by pixels have similar colors, the kernel will treat patches with different color the same since all the color appears on the weight. If a image is brighter, it's going to give a larges value.

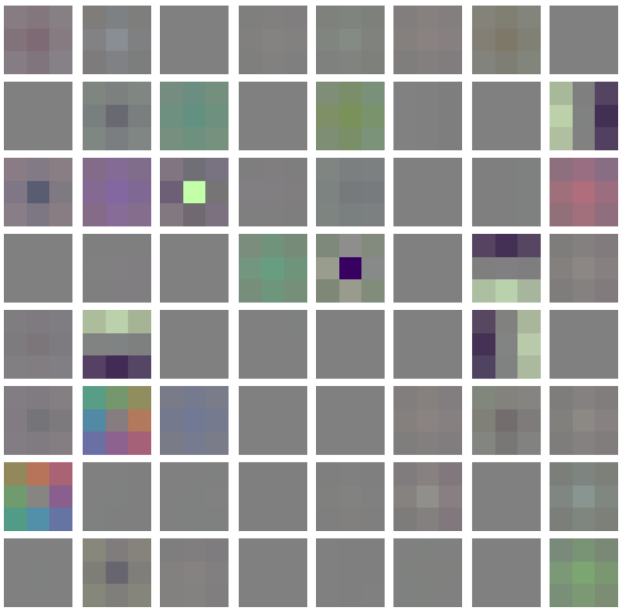

Figure 13: VGG weight maps

5.1.3 ResNet Model

The first convolution layer of ResNet model has 64 7x7 kernel sets. We preform the same operations as to the VGG model, and the result presented in the Figure 14 below. Following are the visual observations:

- Kernel like (0, 0), (6, 3), and (1, 6) with black and white / blue and orange strips are Gabor filters with different angles, color, and frequency. They are good at detecting edges and texture changes.

- Kernel (4, 3) and (4, 7) detects vertical image boundaries. Kernel (4, 7) reacts to image patches that is lighter on the left side of the boundaries and darker on the right side. Kernel (4, 3) does the opposite.

- Kernel (2, 7) and (3, 7) detects horizontal image boundaries. Kernel (3, 7) reacts to image patches that is lighter on the upper side of the boundaries and darker on the lower side. Kernel (2, 7) does the opposite.

- kernels like (3, 1), (5, 1), (4, 4), (4, 5) and (5, 2) detects the corresponding colors. The react when the input patch has similar colors as themselves.

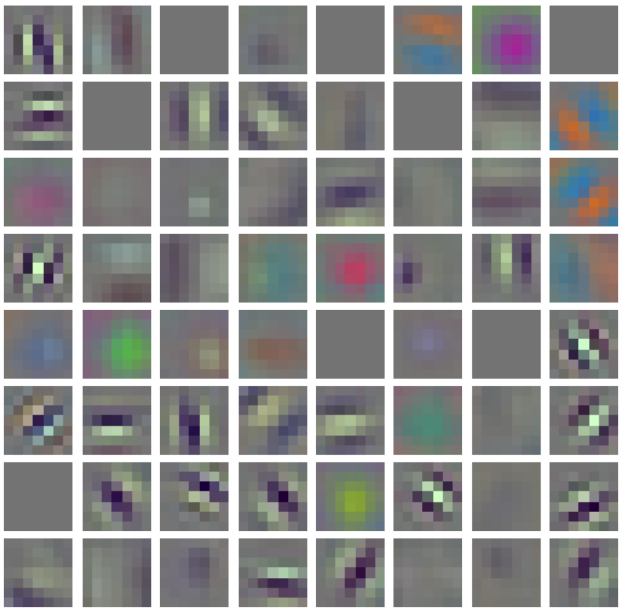

Figure 14: resnet weight maps

5.2 Feature Maps

Generally speaking, from the features maps, we can see that the deeper we go, the more opaque those feature maps would be. For example, we can clearly see the features and silhouette of the bird on the first layer. However, we can barely tell that it's a bird if we go to the last layer.

Figure15, Figure16 and Figure17 are feature maps of the first convolution layer, a middle convolution layer and the final convolution layer of the custom model respectively. Note that the first layer of our custom model is used to create super-resolution. Therefore, it has to have the same number of channels as the original image. That's why we have a total of three feature maps for the first layer of our custom model. Here, the first layer feature map of our custom model has a clear shape of the bird sample, and this is used to combine with three channels of the input image to send into the second layer. The middle layer feature map of our custom model has more channels, and we can still see the shape of the bird sample. However, here you will see more diversity in pixel intensity distribution, meaning that each map shows a different feature of the bird. Tha last layer feature map of our custom model only has a vague shape of the bird sample. This feature map has a low resolution, but with more channels it can capture more details about the bird's characteristics.

Figure 15: The first layer feature map of the custom model

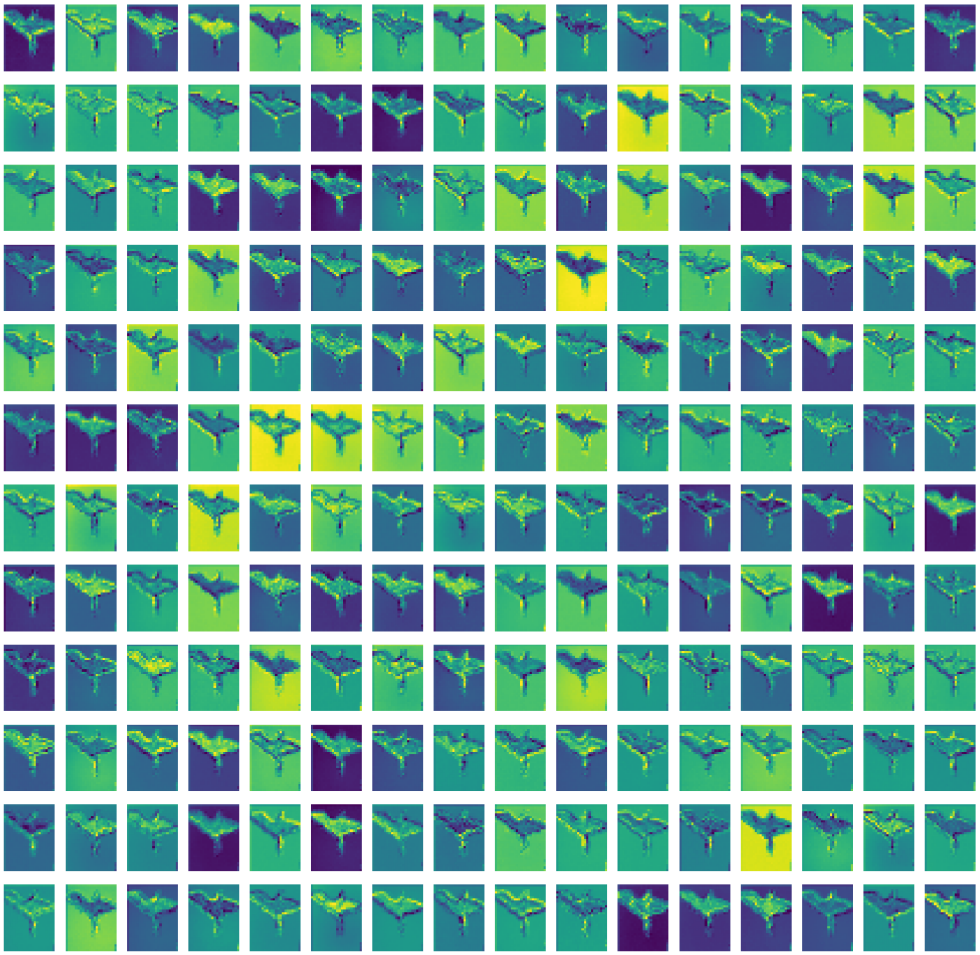

Figure 16: A middle layer feature map of the custom model

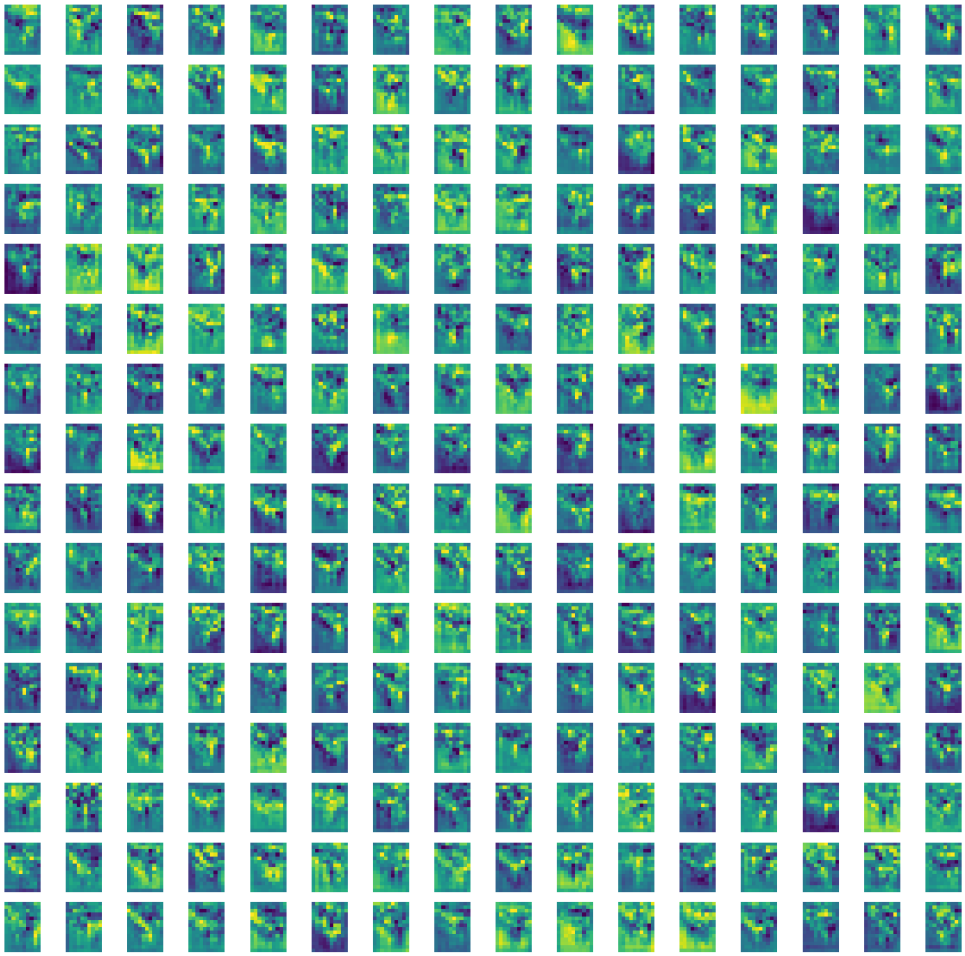

Figure 17: A last layer feature map of the custom model

Figure18, Figure19 and Figure20 are feature maps of the first convolution layer, a middle convolution layer and the final convolution layer of the VGG16 model respectively. In the feature map of the first layer, we can similarly see a very clear shape of the bird sample as we see from the feature map of our custom model's first layer. But since we have more channels, we can see different features presented here. For instance, in cell (6, 0), we can see that this feature captures only the edge of the bird. On the other hand, in cell (0, 6), we can see that this feature captures the shape of the bird. In the feature map of the middle layer, different from what we see in the middle layer of our custom model that were still capturing shape, we discovered that the middle layer of vgg16 mostly only captures the edge of the bird, which can be seen that most, if not all, feature maps have uniform pixel intensity distribution except for the edge of our bird sample. Feature maps of the last layer in vgg16 rather showed some diversity. For instance, cell (0, 2) captures the neck detail of the bird sample, while cell (3, 0) captures the tail detail of the bird sample.

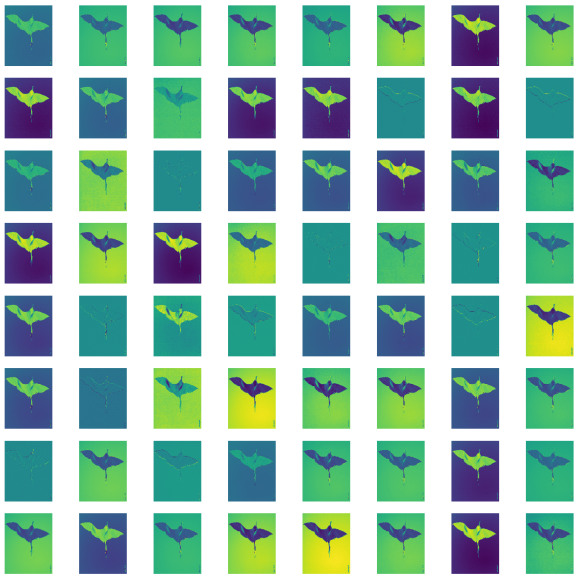

Figure 18: The first layer feature map of the vgg16 model

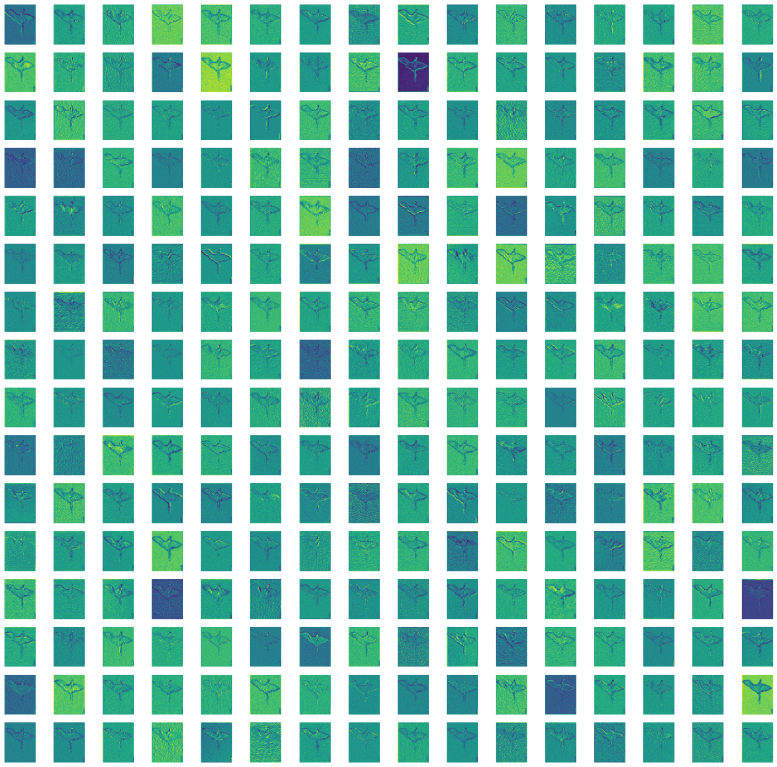

Figure 19: A middle layer feature map of the VGG16 model

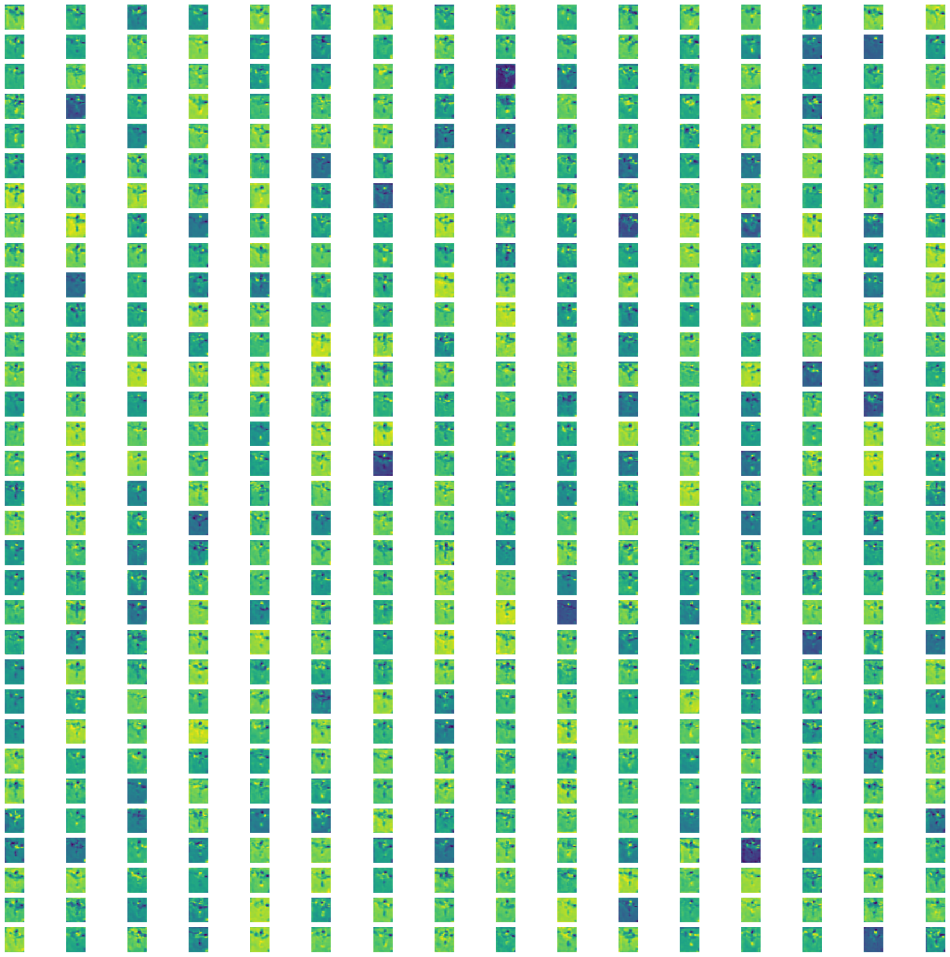

Figure 20: A last layer feature map of the VGG16 model

Figure21, Figure22 and Figure23 are feature maps of the first convolution layer, a middle convolution layer and the final convolution layer of the ResNet18 model respectively. In the feature map of the first layer, we see that most channels have already targeted edges of the bird sample. However, in the feature map of the middle layer, these edges were still preserved. We believe that this was likely caused by the fact that resnet takes the residual from a previous layer and directly combines with the output to feed into the input for next layer. That's why we are able to see a very apparent similarity between cell (2, 2) in the feature map of the first layer and cell (1 ,5) in the feature map of the second layer. Feature maps for the last layer have much more diversity than the feature maps we saw from vgg16 and our custom model. We believe that this is likely caused by the fact that resnet18 has two more depth than vgg16, which enhanced the diversity of different feature maps.

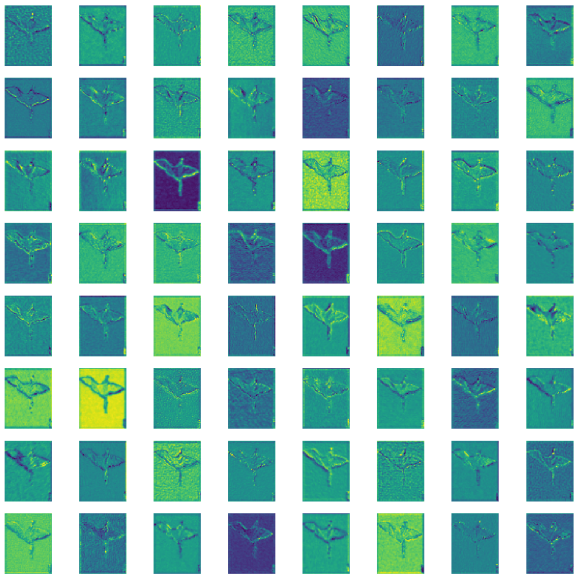

Figure 21: The first layer feature map of the resnet18 model

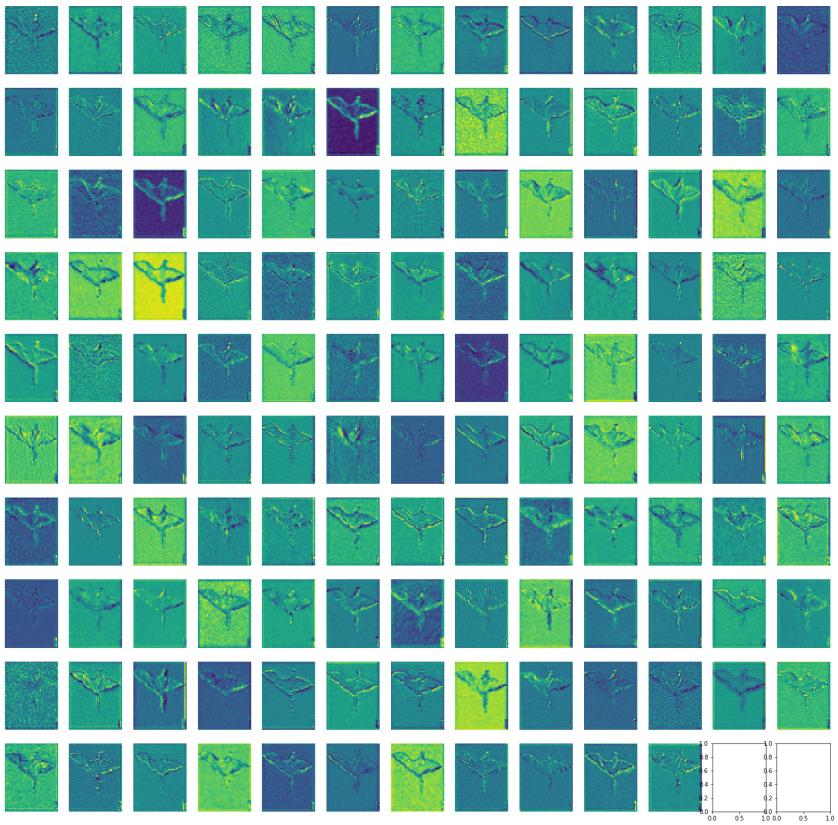

Figure 22: A middle layer feature map of the ResNet18 model

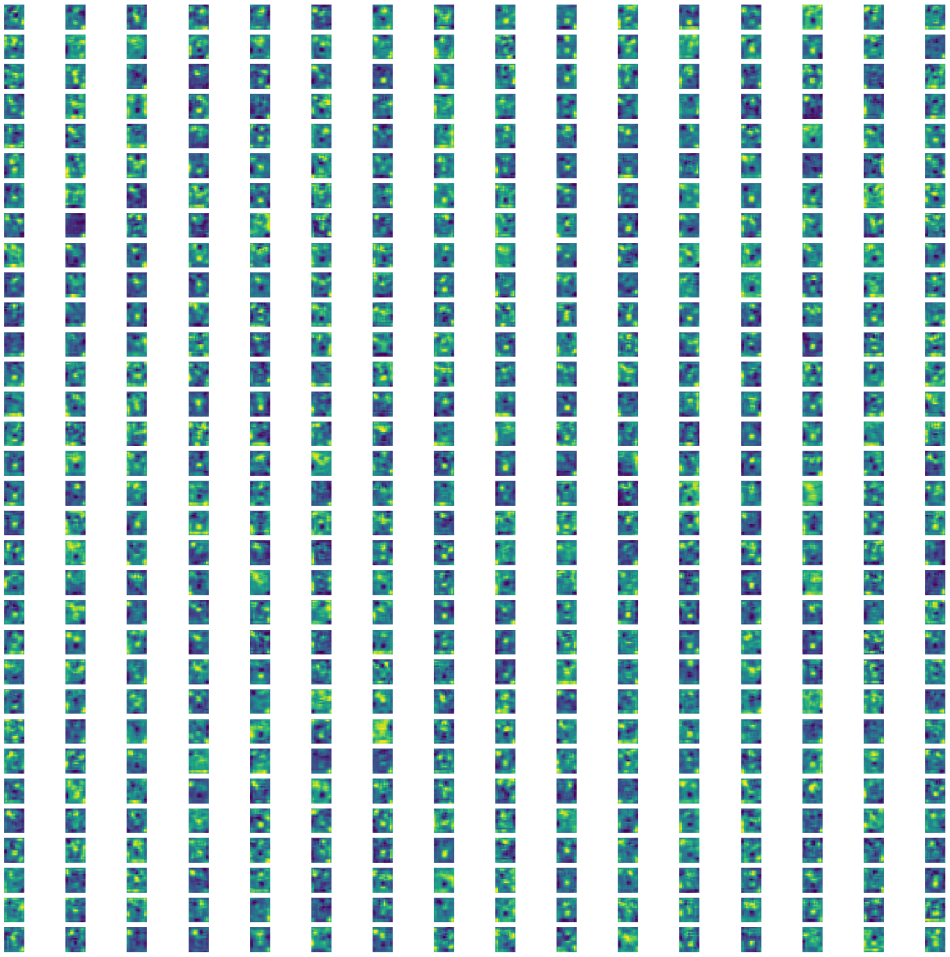

Figure 23: A last layer feature map of the ResNet18 model

6 Discussion

6.1 Comparison across Baseline, Custom, VGG16 and & ResNet18 Models, and Further Improvements

In this assignment, we implemented a deep learning pipeline to test model performance on CUB-200 dataset with 20 classes. Table 18 shows performance statistics for each of these models.

Model	Accuracy
Baseline	0.293
Custom + HT + AG	0.500
VGG16 + HT + AG	0.793
RESNET18 + HT + AG	0.752
VGG16 + HT + AG + FT	0.857
RESNET18 + HT + AG + FT	**0.866**

Table 18: Model performance measured by test accuracy over 50 epochs. Here, HT represents hyperparameter tuning, AG means data augmentation, FT represents fine tuning.

From the performance results, we can see that pretrained vgg16 model outperformed the rest of the models, following by pretrained resnet18. The custom model has an accuracy of 0.5, which is in-between the performance of the baseline model and pretrained vgg16 and resnet18 model.

First, we think that the high performance of vgg16 and resnet18 is largely because they are pre-trained with many images and their convolutional layers can capture many details and patterns in images. As a result, we can simply train the last layer of their classifier network to yield a desired result.

Second, we think that vgg16 outperformed resnet18 for this dataset in particular is because vgg16 preserves more separate features when passed into the classifier than resnet18 does. For instance, There are 102,764,544 parameters in the first linear layer of vgg16 whereas there are only 10,260 parameters in the first linear layer of resnet18. Although this difference can be explained by the use of a residual network, our performance result suggests that a traditional very deep convolutional neural network works better on this dataset by a small margin. On the other hand, after we unfreezed all layers and performed fine tuning, resnet18's performance beat vgg16 by a small margin. We believe that this is caused by the design that resnet18 avoided vanished gradient problem when training convolutional layers.

Third, we think that the major difference between the basline model and our custom model is that our custom model not only used data augmentation and hyperparameter tuning to prevent it from overfitting at early epochs, but layers with difference kernel sizes, strides, padding to capture both global and find structures. We also have one more linear layer and dropout layer to enhance the performance of our custom model. Finally, we introduced a Super-Resolution convolutional layer in our custom model, which finally pushed the accuracy of our custom model to above 0.5.

There are many ways to further improve our custom model given a fixed amount of convolutional layers. However, given time constraint on this assignment, we finished up our model with an accuracy of 0.500 even though we had many better ideas on how to further improve it. As the task for this assignment is to use CNN to classify bird species, it belongs to the field of fine-grained image classification.

First, we can add attention mechanism into our convolutional neural network. As the labels for birds are based on themselves, but not their surroundings, we can let our CNN pay more attention to the birds themselves in the input image rather than their background. Recently, Ma et. al [13] proposed a novel DCANet that can learn connected attentions for convolutional neural networks. Figure 24 is a sample attention mechanism from their paper.

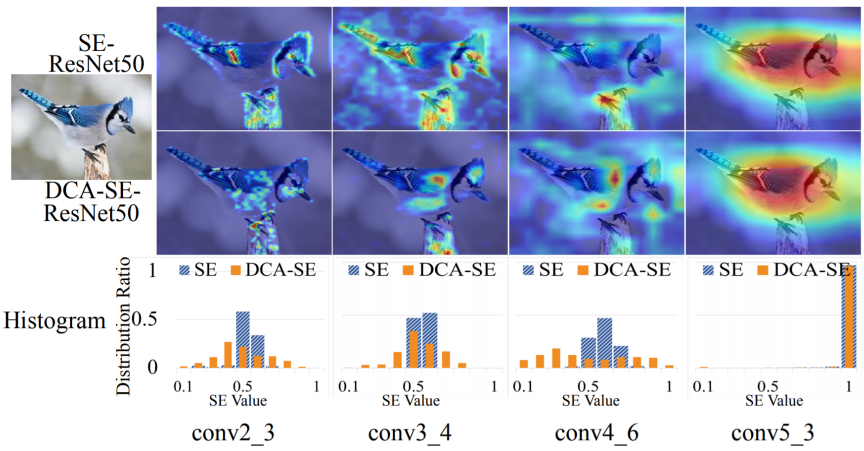

Figure 24: Attention Mechanism in CNN for fine-grained image classification

Second, we can add a semantic grouping module after our convolutional neural networks that help specify semantics of sub-features for a global feature. This method is proposed by Luo et. al [14]

in a recent paper to address an effective yet cheap tool to enhance the performance of fine-grained image classification. Figure 25 is a visualization of their mechanism

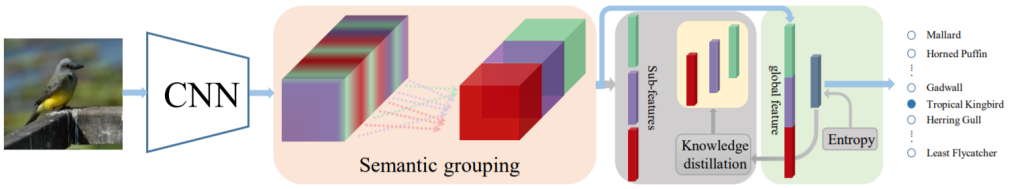

Figure 25: Semantic Grouping Mechanism in CNN for fine-grained image classification

6.2 Improvements based on Hyperparameters Tuning

6.2.1 Learning Rate

This hyperparameter helps the model speed up the learning process of convolutional neural networks by increasing the amount of changes in weights and bias in each batch. We started with a learning rate of $1e-4$ for most models, but found out that the training and validation accuracy is still low over certain epochs. When we turned our learning rate into $1e-3$, models quickly learned features and showed a decent performance, but this time it has some fluctuations between epochs. As a result, we used a learning rate scheduler that starts at $1e-3$ and decreased the learning rate as we have more epochs to stabilize the training process.

6.2.2 Weight Decay

This hyperparameter helps the model prevent from overfitting by punishing big weights using L2 regularization. Using this method, the model is less likely to *remember* training images because big weights are punished. However, given limited number of images in the dataset, accuracy improvements on weight decay can be hardly seen without the help of image augmentation, which increases training set variance. After we used image augmentation, switching weight decay from 0 to a value between $1e-3$ and $1e-4$ (depending on models) has some effects in increasing test set accuracy.

6.2.3 Weight Initialization

Using an xavier uniform weight initialization helps the model learn faster during early epochs because weights are initialized with a non-zero floating points. Because of these initial weights, gradients would be larger for early epochs compared to the case when initial weights are all set to 0. In one of our experimentation for vgg16 where we only unfreezed last layer in the classifier and initialized it with xavier uniform, we found that the validation accuracy pumped to 0.5 right after the first epoch!

6.2.4 Epoch

Generally, the more epochs, the better the model. On the other hand, since our model contains few images, our model can easily converge well below 100 epochs (most of the time, below 50 epochs). This hyperparameter only helps when our model's learning rate is low, which needs more epochs to converge.

6.2.5 Batch Size

Generally, the larger the batch size, the better the performance of the model and the faster the model converges. This is a larger batch size could potentially improve the effectiveness of the optimization steps resulting in more rapid convergence of the model parameters. Also, a larger batch size means that one batch could contain images of more categories, which add the diversity of the batch for the model to learn, for the loss function to calculate loss and subsequently for the optimizer to update weights. We always used a batch size of 20 whenever possible to ensure that each batch contains

as many categories of birds as possible, which can be ideally one image from each category. We observed slower training speed and a slight decrease in accuracy when we decreased the batch size to, say, 2-4.

6.3 Improvements based on Other Design Choices

6.3.1 Image Augmentation

Research from recent years has proved the effectiveness of image augmentation. Wang and Perez [15] have used several techniques to show that image augmentation can effectively solve overfitting problems by creating variance of training set, which is equivalent to increase the training set size. While there are many creative ways to augment existing image dataset (including one of my research collaborator's recent work that uses game engines to augment image data), we only used basic techniques such as **center crop, random horizontal flip, random vertical flip, and random rotation** to augment images from the training set. Surprisingly, this augmentation significantly reduced the overfit problems for larger models (custom, vgg, resnet), where these models' training accuracy can easily go up to 1 without image augmentation techniques.

6.3.2 Super-Resolution & Residual Learning

We used a convolutional layer that outputs data having the same dimension with our input image in our custom model. This layer of convolutional layer aims to create super resolution to enhance the model's performance on find-grained image classification problems. We referred to Cai et. al's paper [12] on this technique. By replacing a normal convolutional layer of 32 channels with a SR convolutional layer of 3 channels and add the residual as the input for next convolutional layer, we improved the accuracy of our custom model from 0.486 to 0.500.

6.3.3 Batch Normalization

Research from recent years has proved that adding a batch normalization after each convolutional layer can accelerate deep network training by reducing internal covariate shift. Ioffe and Szegedy [16] used batch normalization techniques on inception model and decreased Top-5 Error rate below 5%. As we borrowed the structure of Alexnet as our custom model, we added batch normalization layers after each of its convolutional layers. Along with our image normalization, this batch normalization technique made the training process more stable and converge faster.

6.3.4 Selective Freezing

We used the pre-trained vgg16 and resnet18 model by freezing all layers except the last layer. After reaching an accuracy of 0.75 - 0.8, we fine tuned the pre-trained vgg16 and resnet18 model by unfreezing the rest of the layers to further improve the performance. After we fine-tuned these two models, they are able to reach an accuracy of 0.85-0.9 in predicting bird categories.

6.3.5 Learning Rate Scheduler

Learning rate scheduler is a useful tool to control the learning rate while the model is approaching convergence to improve model stability as well as increase accuracy performance. Recent research done by Xu et. al [17] proposed a reinforcement learning based framework that can automatically learn an adaptive learning rate schedule by leveraging the information from past training histories. While we don't use their approach to control our learning rate, we used a step learning rate scheduler with step_size of epochs//4, and gamma of 0.5. This increases the performance on pretrained models by around 0.05.

7 Team Contributions

Colin: Implemented dataset.py, train.py, util.py, model.py for the final custom model and a pipeline built upon jupyter notebook (training.ipynb) and documented dataset.py, util.py, and pipeline.py. Wrote report for abstract, introduction, related work, discussion, references, as well as everything about the baseline and custom model.

Jerry: Implemented model.py for vgg16 and resnet18, fixed minor bugs in other .py files, documented train.py, and code for weight maps. Tune and applied training techniques on vgg16 and resnet18. Wrote report for all vgg16 and resnet18 related sections.

Bingqi: Attempted implementing model.py for custom model. Implemented code for weight and feature maps. Contributed to the report for feature map.

References

[1] S. J. Reddi, S. Kale, and S. Kumar, "On the convergence of adam and beyond," in *International Conference on Learning Representations*, 2018.

[2] P. Welinder, S. Branson, T. Mita, C. Wah, F. Schroff, S. Belongie, and P. Perona, "Caltech-UCSD Birds 200," Sept. 2010.

[3] S. Branson, C. Wah, F. Schroff, B. Babenko, P. Welinder, P. Perona, and S. Belongie, "Visual Recognition with Humans in the Loop," in *Computer Vision – ECCV 2010* (K. Daniilidis, P. Maragos, and N. Paragios, eds.), Lecture Notes in Computer Science, (Berlin, Heidelberg), pp. 438–451, Springer, 2010.

[4] P. Welinder and P. Perona, "Online crowdsourcing: Rating annotators and obtaining cost-effective labels," in *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Workshops*, pp. 25–32, June 2010. ISSN: 2160-7516.

[5] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *arXiv:1409.1556 [cs]*, Apr. 2015. arXiv: 1409.1556.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *arXiv:1512.03385 [cs]*, Dec. 2015. arXiv: 1512.03385.

[7] S. Huang, Z. Xu, D. Tao, and Y. Zhang, "Part-Stacked CNN for Fine-Grained Visual Categorization," *arXiv:1512.08086 [cs]*, Dec. 2015. arXiv: 1512.08086.

[8] L. Zhang, S. Huang, W. Liu, and D. Tao, "Learning a Mixture of Granularity-Specific Experts for Fine-Grained Categorization," in *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 8330–8339, Oct. 2019. ISSN: 2380-7504.

[9] D. Korsch, P. Bodesheim, and J. Denzler, "End-to-end Learning of a Fisher Vector Encoding for Part Features in Fine-grained Recognition," *arXiv:2007.02080 [cs]*, July 2020. arXiv: 2007.02080.

[10] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[11] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

[12] D. Cai, K. Chen, Y. Qian, and J.-K. Kämäräinen, "Convolutional Low-Resolution Fine-Grained Classification," *arXiv:1703.05393 [cs]*, Oct. 2017. arXiv: 1703.05393.

[13] X. Ma, J. Guo, S. Tang, Z. Qiao, Q. Chen, Q. Yang, and S. Fu, "Dcanet: Learning connected attentions for convolutional neural networks," 2020.

[14] W. Luo, H. Zhang, J. Li, and X.-S. Wei, "Learning semantically enhanced feature for fine-grained image classification," 2020.

[15] L. Perez and J. Wang, "The effectiveness of data augmentation in image classification using deep learning," 2017.

[16] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," 2015.

[17] Z. Xu, A. M. Dai, J. Kemp, and L. Metz, "Learning an adaptive learning rate schedule," 2019.