Classifying Birds Using Convolutional Neural Networks

Mitchell Zhang
miz134@ucsd.edu

Jun (Moses) Oh
j6oh@ucsd.edu

Abstract

In this project, we explored deep learning tools in PyTorch to create and train deep convolutional networks to classify a range of birds. We used 20 categories of birds in the Caltech-UCSD birds dataset to train and test our models. We trained the given baseline model with Cross Entropy as the loss function, Xavier initialization for the weights, Adam optimizer. Our training procedure consisted of using 2-way cross validation with two holdout sets with 60 images each. Training the given model resulted in an accuracy of 26%. We were able optimize the baseline model by adding extra convolutional layers and experimenting with variety of hyper parameters. Through this custom model, we obtained a score of 43.2%. We explored other pre-trained models such as VGG16 and Resnet18 through PyTorch, in which we fine tuned by replacing the last fully connected layer with an appropriate-sized one for this specific problem. Our accuracy on these pretrained models peaked at 75.5% and 79.2% respectively.

1 Introduction

In this project, we use PyTorch to create a convolutional neural network to learn to classify types of birds. The Caltech-UCSD birds dataset consists of 12,000 images of 200 classes of birds. However, we focus on only 20 classes (600 images) of these. We use 3 of each class randomly sampled to create our validation set. We used two-fold cross validation, creating our two validation sets randomly to be 60 images each total. Our baseline model consisted of 4 convolutional layers, 1 max pool layer, 1 average pool layer, and 2 fully connected linear layers. This model gave us an accuracy of 26.2%. This model gave us an accuracy better than chance, but was still not very accurate. We looked to improve this model by adding our own custom layers, such as 2 convolutional layers and a fully connected layer, improving accuracy, peaking at 43.2%. We attributed the general lack of accuracy (¡50%) to our small dataset. Since our validation set consisted of only 3 images per class, there was a noticeable fluctuation across tests in the validation loss and accuracy. We also noticed discrepancies between the first fold and second fold, despite resetting the model in between the folds. We also attributed this to variance across the image datasets. The test set accuracy was noted to occasionally outperform the best validation performance, supporting our conclusion that the dataset size was a major factor in these discrepancies. We also tested the dataset on pretrained models, such as vgg16 and resnet 18. We tuned these to fit our dataset by freezing certain layers. These models greatly outperformed the baseline and custom models we trained, obtaining accuracies of 76.1% and 80.4%. These models were both pretrained and were deeper, which would explain some of the difference. Overfitting to a degree was observed in nearly all models, but we used learning rate, learning rate scheduling, weight decay, and dropout to limit it.

2 Related Works

[1] ”Bird Species Categorization Using Pose Normalized Deep Convolutional Nets” paper implements a CNN which follows Krizhevsky’s network structure that computes features from the object’s pose. The model separates the bird into head and body in which they feed into the CNN. The output is aggregated to classify which category the bird is in.

[2] ”Learning Branched Networks for Fine-grained Representations” paper realizes the limitation of relying only on the pose of the bird. So they take advantage of descriptions of the birds in which they feed into BERT NLP model to combine with their CNN which results in a more accurate classifier.

[4] ”Ideas on how to fine-tune a pre-trained model in PyTorch” article describes the fine-tuning process for pre-trained models and provides guidance on which hyperparamaters to tune such the learning rate and weight decay.

3 Models

The baseline model consisted of 4 convolutional layers, 1 max pool layer, 1 average pool, and 2 fully connected linear layers.

$$conv1- > conv2- > conv3- > maxpool1- > conv4- > avgpool- > fc1- > fc2$$

Layer	Input	Output	Parameters
Convolutional Layer 1	[3, 224, 224]	[64, 222, 222]	Kernel-size = 3x3, Activation = ReLU
Convolutional Layer 2	[64, 222, 222]	[128, 220, 220]	Kernel-size = 3x3, Activation = ReLU
Convolutional Layer 3	[128, 218, 218]	[128, 218, 218]	Kernel-size = 3x3, Activation = ReLU
Max pool	[128, 72, 72]	[128, 72, 72]	Kernel-size = 3x3, Stride = 3
Convolutional Layer 4	[256, 35, 35]	[256, 35, 35]	Kernel-size = 3x3, Activation = ReLU
Average Pool	[256, 35, 35]	[256, 1, 1]	None
FC Linear Layer 1	256	1024	Dropout of 50%, Activation = ReLU
FC Linear Layer 2	1024	20	None

The first input/output number represents the number of channels. For example, in convolutional layer 1, the input is [3, 224, 224] and output is [64, 222, 222]. The input number of channels is 3 and the output number of channels is 64. All convolutional layers also had Batch Normalization of the same output size and no padding was used.

The hyperparameters we used to obtain our test accuracy of 43.2% were: epoch size = 25, batch size = 4, learning rate = 1e-4. Learning rate scheduler and weight decay were not implemented as part of our baseline model.

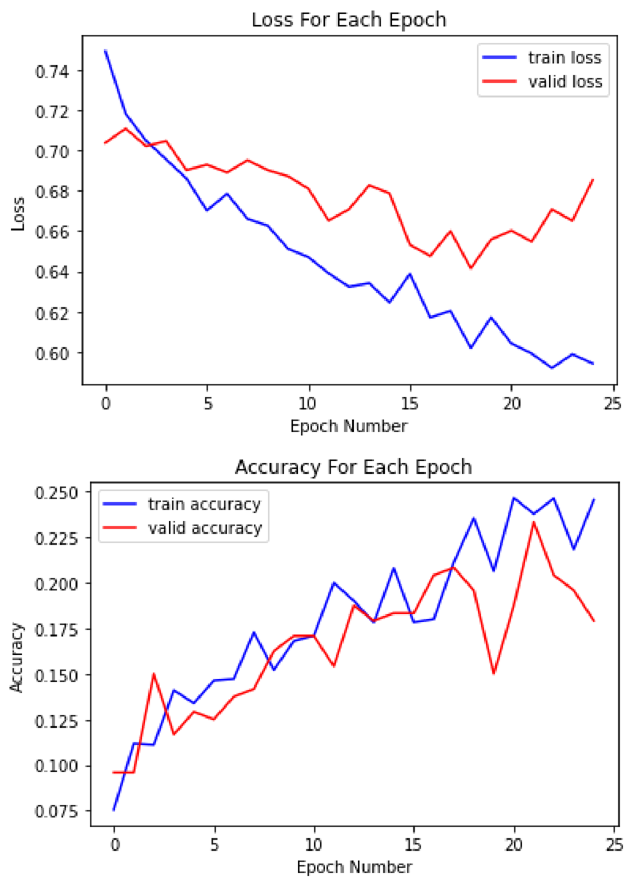

Figure 1: Baseline loss and accuracy, Learning rate = 1e-4, Final accuracy = 26.2%, Epoch size = 25

After fitting our data on the base model, we attempted to improve the base model by adding our own layers and tweaking the hyperparameters. We concluded on this model, which gave us a test accuracy of 43.2%.

$$conv1- > conv2- > maxpool1- > conv3- > conv4- > conv5- >$$
$$maxpool2- > conv6- > avgpool- > fc1- > fc2- > fc3$$

Layer	Input	Output	Parameters
Convolutional Layer 1	[3, 224, 224]	[64, 222, 222]	Kernel-size = 3x3
Batch Normalization 1	[64, 222, 222]	[64, 222, 222]	None
Activation ReLU 2	[64, 222, 222]	[64, 222, 222]	None
Convolutional Layer 2	[64, 222, 222]	[128, 220, 220]	Kernel-size = 3x3
Batch Normalization 2	[128, 220, 220]	[128, 220, 220]	None
Activation ReLU 2	[128, 220, 220]	[128, 220, 220]	None
Max pool 1	[64, 220, 220]	[64, 73, 73]	Kernel-size = 3x3, Stride = 3
Convolutional Layer 3	[64, 73, 73]	[128, 71, 71]	Kernel-size = 3x3
Batch Normalization 3	[128, 71, 71]	[128, 71, 71]	None
Activation ReLU 3	[128, 71, 71]	[128, 71, 71]	None
Convolutional Layer 4	[128, 71, 71]	[128, 69, 69]	Kernel-size = 3x3
Batch Normalization 4	[128, 69, 69]	[128, 69, 69]	None
Activation ReLU 4	[128, 69, 69]	[128, 69, 69]	None
Convolutional Layer 5	[128, 69, 69]	[256, 67, 67]	Kernel-size = 3x3
Batch Normalization 5	[256, 67, 67]	[256, 67, 67]	None
Dropout 1	[256, 67, 67]	[256, 67, 67]	25%
Activation ReLU 5	[256, 67, 67]	[256, 67, 67]	None
Max pool 2	[256, 67, 67]	[256, 22, 22]	Kernel-size = 3x3, Stride = 3
Convolutional Layer 6	[256, 22, 22]	[512, 10, 10]	Kernel-size = 3x3, Stride = 2
Batch Normalization 6	[512, 10, 10]	[512, 10, 10]	None
Activation ReLU 6	[512, 10, 10]	[512, 10, 10]	None
Average Pool	[512, 10, 10]	[512, 1, 1]	None
FC Linear Layer 1	512	256	None
Dropout 2	256	256	25%
Activation ReLU 7	256	256	None
FC Linear Layer 2	256	128	None
Activation ReLU 8	128	128	None
FC Linear Layer 3	128	20	None (Max taken as prediction)

The first input/output number represents the number of channels. For example, in convolutional layer 1, the input is [3, 224, 224] and output is [64, 222, 222]. The input number of channels is 3 and the output number of channels is 64. All convolutional layers had a stride of length 1 and no zero-padding unless otherwise specified. The training accuracy and validation accuracy peaked at 51.6% and 42.5% respectively. The hyperparameters we used to obtain our test accuracy of 43.2% were: epoch size = 50, batch size = 4, learning rate = 2e-4, weight decay = 1e-6, learning rate scheduler step = 3, learning rate scheduler gamma = 0.75.

VGG16 is a deep learning model that consists of 16 combined layers of convolutional layers and fully connected layers. Due to having 3 fully connected layers, VGG16 consists of many weights that contribute to the output. VGG was able to outperform many of the previous models solely because if its depth that allowed the model to detect many more features that made up the categories. This amount of layers was only possible because of advancements in hardware such as GPU that accelerated training capabilities.

Resnet neural network model uses residual blocks to propagate inputs faster throughout the layers. Resnet also uses 1 to 1 convolutional layer that allows the model to prevent having so much parameters. In contrast to VGG's total number of weights, Resnet has only a fraction. The skip connections also help solve the problem of vanishing gradients that exists in deep networks such as VGG by helping gradients flow throughout the deep network. This explains why Resnet is more efficient and faster than VGG.

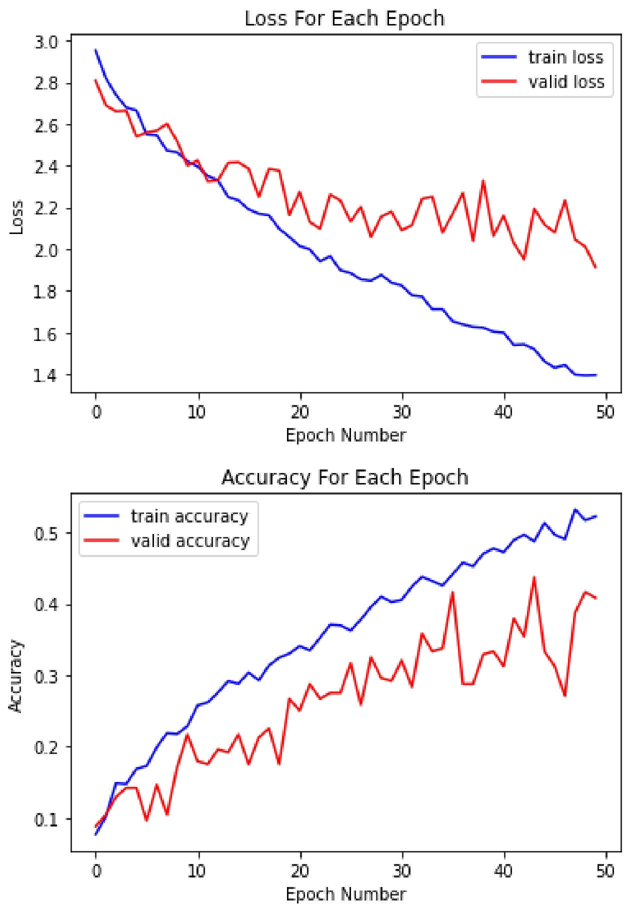

Figure 2: Custom model loss and accuracy, Learning rate = 2e-4, Final accuracy = 43.2%, Epoch size = 50

The loss and accuracy plot show valid generally following the trend of the training set, more closely in the first 25 epochs that in the latter half. We concluded that the model began slightly overfitting in the latter half, but this amount was deemed acceptable since loss and accuracy were still decreasing and increasing respectively. Increasing epoch size demonstrated much heavier overfitting on the loss graph and minimal improvements in accuracy in the validation set (although loss and accuracy on the training set still decreased and increased respectively). Since overfitting was a factor we tried to limit, we set the epoch size to 50. There were also many fluctuations in both the validation loss and accuracy. We attributed this to the low dataset size. Since we had only 60 images, 3 of each class, there may be lots of variance in the loss and accuracy across each training.

A batch size of 4 was used after testing different batch sizes on multiple models. A small increase or decrease in batch size, such as to 3 or 5, caused significant decreases in accuracy. A larger batch size, such as 10 or 20, saw similar results. A batch size of 1 was not used, since batch normalization would be unable to obtain accurate validation results in evaluation mode. A large learning rate saw less overfitting, but would also cause higher fluctuations in the validation loss. We used learning rate scheduler to decrease the learning rate as epochs increase. This is used to quickly converge, but not get stuck in a local minima early. The large initial learning rate would skip over smaller local minima, while the lower later learning rates would be used to converge to a local minima after training. Weight decay is used to prevent weights from becoming too large, effectively penalizing large weights and complexity. We tested with more layers (more than adding 2) and more channels/neurons in both the convolutional layers and linear layers (testing values of 1024, 2048, 4096, 8192), but found the best performance with a simpler model. To decrease overfitting, we added another dropout layer in the convolutional layer. We found that two relatively low dropout layers (0.25) performed better and lowered overfitting more than a single larger dropout (0.5) in the linear layer. The appropriate activation function in the last layer is ReLU, to get a probability of each class label prediction. This was concluded after testing other activation functions such as sigmoid and tanh. ReLU's primary benefit is that it does not have the problem of vanishing gradient, since its derivative is not less than 1 and greater than 0. To get the prediction, we took the maximum of the output of the final layer. Overall hyperparameters used were: epoch size = 50, learning rate = 2e-4, weight decay value = 1e-6, batch size = 4, learning rate scheduler step = 3, and learning rate scheduler gamma = 0.75.

The model architecture was a result of attempting to improve the model's performance on the test set while limiting overfitting noticed on the validation set during training. Increasing complexity by adding convolutional layers and increasing channels allows the model to gain a deeper understanding of features. However, too complex of a model causes overfitting. We tried adding more layers, and only slightly increasing number of neurons in the last layer. Since we added 2 convolutional layers in total, we added another max pool layer. Max pool takes the maximum from each kernel, and thereby selecting the most important feature in that region. With additional and more complex layers, we noticed more overfitting, so we attempted to correct this by adding dropout and changing our hyperparameters such as epoch size, learning rate, weight decay, and learning rate scheduler to minimize overfitting. After getting a consistent model that could perform better than the base with minimal overfitting, we further looked to improve it by adding another linear layer, slightly changing layer sizes, and changing activation functions.

To train our model, we used cross entropy loss as our objective function. We used the Adam optimizer, passing in our learning rate and weight decay to the optimizer. We initialized our weights using Xavier initalization. Weights for both the convolutional and linear layers were initialized with Xavier, with the bias for the convolutional layer initialized to zeros. We used two-fold cross validation by splitting the data into two folds randomly. Each fold consisted of a training set of 540 images and a validation set of 60 images. The validation set was balanced across each class by randomly sampling 3 images from each class. Loss and accuracy were plotted using an average of these two folds across every epoch. Training dataset was shuffled for every epoch, to ensure randomness learning rather than memorization from the model. We noticed the test set accuracy was sometimes greater than the validation accuracy. We attributed this to the small data set size of the validation set compared to the large test set.

4 Experiments

4.1 Custom Model

Layer	Test Accuracy	Valid Accuracy	Train Accuracy
Lower initial layer channels	38.4%	36.5%	38.9%
More layers	39.4%	40.7%	49.7%
Higher later layer channels	41.3%	43.2%	64.1%
Large Linear	41%	41.7%	50.5%
Early dropout	34.1%	35.2%	42.3%
Late Convolutional dropout	37.9%	39.2%	46.1%
Activation function	28.8%	28.3%	32.3%

These values were the highest noted accuracies on the according set after tuning a variety of hyperparameters. These hyperparameters include: epoch size, batch size, learning rate, weight decay value, learning rate schedule step, and learning rate schedule gamma.

Lower initial layer channels consisted of creating a model with lower output channels initially. For example, instead of the first layer having 64 output channels, we changed the first layer to initially have 32 output channels. We did not notice any large charges in performance.

More layers included adding more convolutional layers. We added more intermediary convolutional layers in between each initial base layer, following the previous layer's example. For example, we added another layer in between the first and second layer to take in 64 input channels but also output 64 output channels. While this improved the model, we found that after we added more layers, overfitting was more common. We also differed where we added the new layers to the existing base model.

Higher later layer channels included creating a model with higher output channels near the end. We increased the number of channels that each layer output by a factor of two. This would result in the final convolutional layer outputting up to 1024, 2048, 4096. Our fully connected linear layer would also have increased neurons as a byproduct, increasing run time. It also increased overfitting in some instances.

Large linear consisted of creating a fully connected linear layer with many outputs. For example, we tested fully connected linear layers with 1024, 2048, 4096, 8192 input/outputs. Increasing the number of neurons in the fully connected linear layers improved accuracy marginally in some instances, but caused overfitting and the validation loss to fluctuate more heavily in other instances. Particularly, the 8192 layer we tested caused drastic fluctuations in the validation loss across 50 epochs. Increasing the number of neurons in these layers drastically increased run time.

Early dropout included adding dropout in some of the earlier layers. For example, we tried adding a low dropout rate of 0.1 and 0.2 in the first two layers. We noticed that this did not seem to improve the model much, and often led to a stagnate and fluctuating highly validation loss. This may be because there are many other layers in the network that it learns to not use all of the neurons in the first few layers.

Late convolutional dropout included adding dropout in some of the later convolutional layers. For example, we tried adding a low dropout rate of 0.2-0.3 in the last convolutional layer. This seemed to slightly lower overfitting. However, a high dropout rate (such as increasing to 0.4 or higher), coupled with the dropout in the fully connected linear layer, started to drop performance significantly. This may be because as we drop too many neurons, the model does not have enough data to accurately capture the classification.

Activation function included testing with different activation functions in each layer, such as ReLU, sigmoid, and tanh. ReLU overall had the best performance, with sigmoid marginally the worst compared to tanh. However, sigmoid was slightly more consistent in loss and accuracy.

Adding complexity to the model, such as increasing number of neurons or layers, could improve accuracy, but was often inconsistent. Because our validation set was small in size, it was often a harsh judge of the resulting test set accuracy. Our test set accuracy was often higher than the highest validation set accuracy, which we primarily attributed to differences in sample size. Adding dropout decreased overfitting trends, but too much or too often dropout started decreasing performance. We guessed that eliminating too many neurons from the network caused the network to be unable to properly predict and create a classification from the remaining data. Our model had to balance between complexity to accurately fit our data and allow for generalizations while avoiding overfitting.

4.2 VGG

Layer	Test Accuracy	Valid Accuracy	Train Accuracy
Replace last layer (freeze rest)	75.5%	74.1%	98.4%
Unfreeze last 2 convolutional layers	65.7%	68.2%	95.2%
Unfreeze last 3 convolutional layers	69.2%	71.2%	98.4%
Unfreeze last 3 conv layers and decrease learning rate	76.1%	73.5%	98.6%

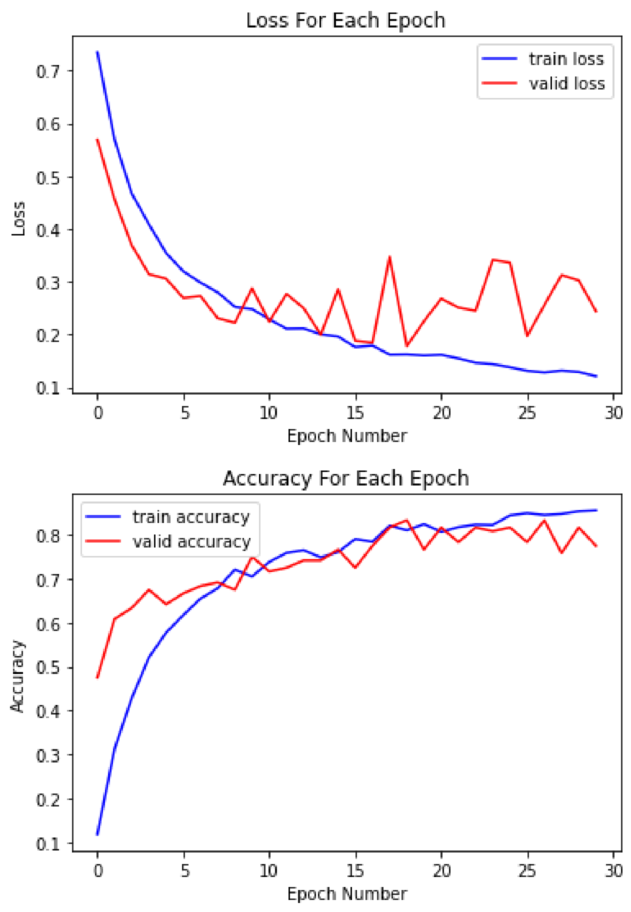

Figure 3: VGG 16, Learning rate = 1e-4, Final accuracy = 75.5%, Epoch size = 25

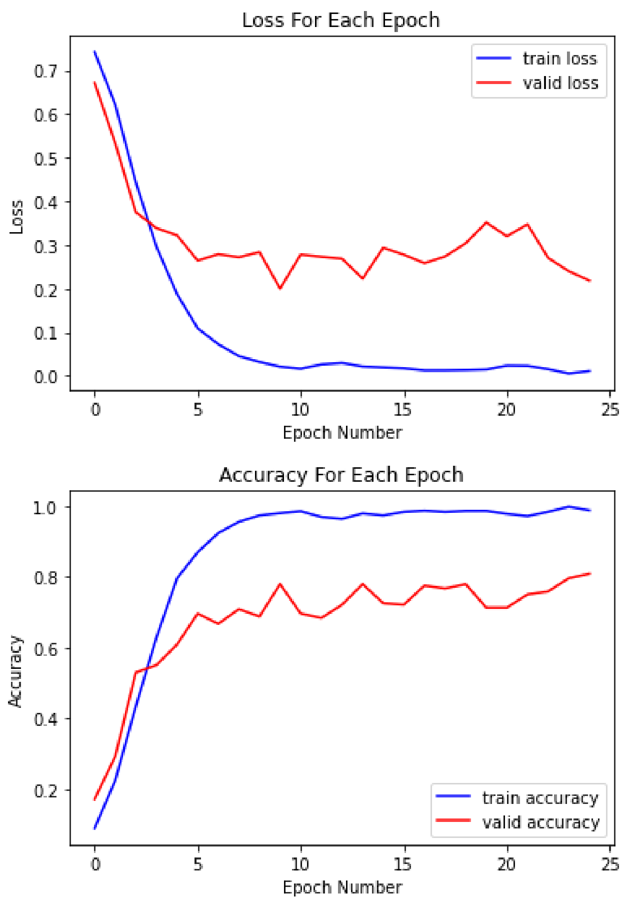

Figure 4: VGG 16 Fine Tuning, Learning rate = 1e-5, Final accuracy = 76.1%, Epoch size = 25

The idea of Transfer Learning allows us to use pre-trained models to learn and classify our set of data because the neural network has already trained over millions of data and found the weights that best points out the distinct features in each picture. Since we are also using images as inputs and attempting to classify different categories of things such as birds, models such as VGG16 and Resnet18 works well with our data set.

By using a pre-trained VGG16 and replacing the last layer with a custom one for our specific classifier which consists of 20 categories, we were able to get a decent accuracy score of 75% straight away. In this case, the model had all its weights frozen except just the last layer - in order to fit our custom number of categories.

I attempted to increase the accuracy by unfreezing other layers in the model. Since the beginning layers of CNNs detect common features such as edges and shapes, we don't want to retrain those weights since the model comes with the best weights for thousands of categories. Instead, we want to alter layers deeper in the network that detect more general and vague features that may be significant to our problem. Initially having all the weights frozen besides the last fully connected layer, I started to unfreeze layers starting from the last convolutional layer one by one while manipulating the learning rate. I was able to pass the initial 75% by a small margin of 1% when unfreezing the last 3 convolutional layer and decreasing the learning weight. My inference is that since the model consists of many convolutional layers that was effective when trained with millions of data, we won't be able hit high accuracy scores given the limitation of how much data we were training the model with. A smaller learning weight helps since we don't want to change too much of the features that the pretrained model has learned.

4.3 Resnet

Layer	Test Accuracy	Valid Accuracy	Train Accuracy
Replace last layer (freeze rest)	79.2%	81.1%	99.5%
Unfreeze last 2 convolutional layers	77.2%	72.5%	98.6%
Unfreeze last 3 convolutional layers	78.6%	82.3%	97.4%
Unfreeze last 3 conv layers and decrease learning rate	80.4%	85.3%	99.4%

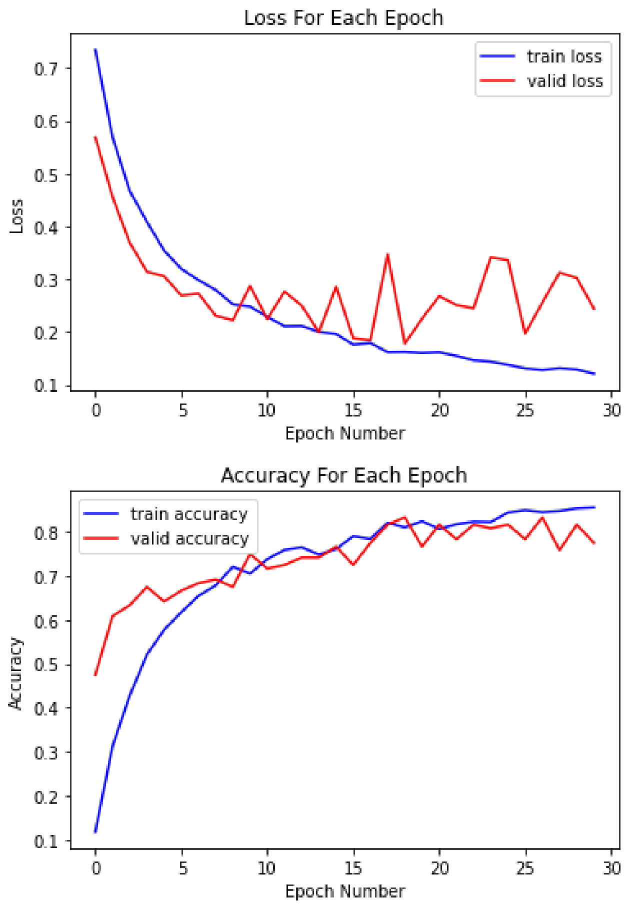

Figure 5: Resnet18 loss and accuracy, Learning rate = 1e-3, Final accuracy = 79.2%, Epoch size = 25

Similar to the process in VGG, we started off by replacing the last fully connected layer with our own classifier for 20 categories. We were able to gain a higher score (79.2%) than VGG (76.1%). We were able to achieve the best accuracy by freezing everything but the last 3 convolutional layers and our custom classifier. We infer that the reason why the improvement was small is because of the fact that we don't have nearly as much data required to fully optimize the deep neural network model.

5 Feature Map and Weight Analysis

5.1 Weight Map

These are visualizations for the first convolutional layer weight maps.

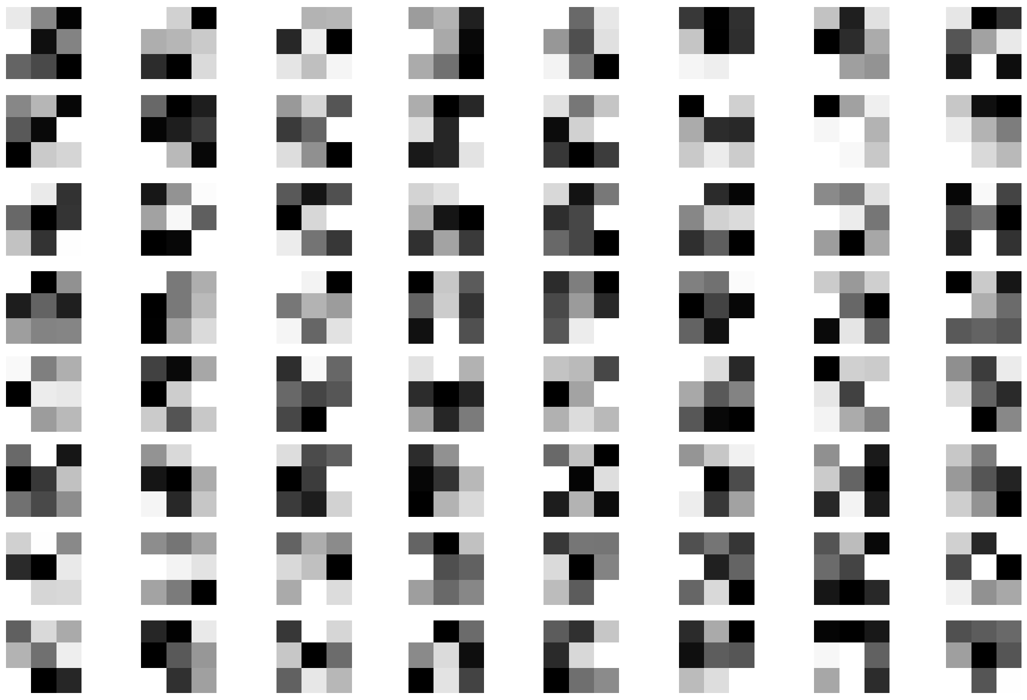

Figure 6: First Weight Map for Custom Model

Each of the weights focus on a different subset of the picture. For example, some of the weights are very light in the upper left corner, representing that that weight map is focused on detecting features from the upper left section of the image. At this stage, it is difficult to see a correlation between the weight map and the image.

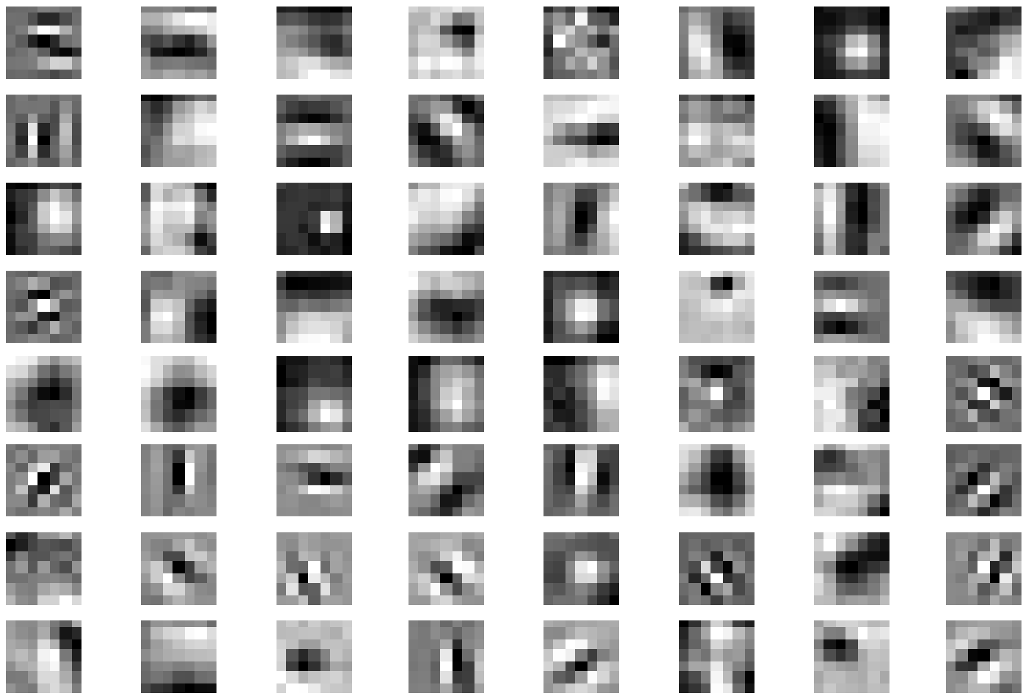

Figure 7: First Weight Map for Resnet 18 Model

The first weight map in Resnet appears to detect more general features of the image such as shapes on top of basic features such as horizontal and vertical edges.

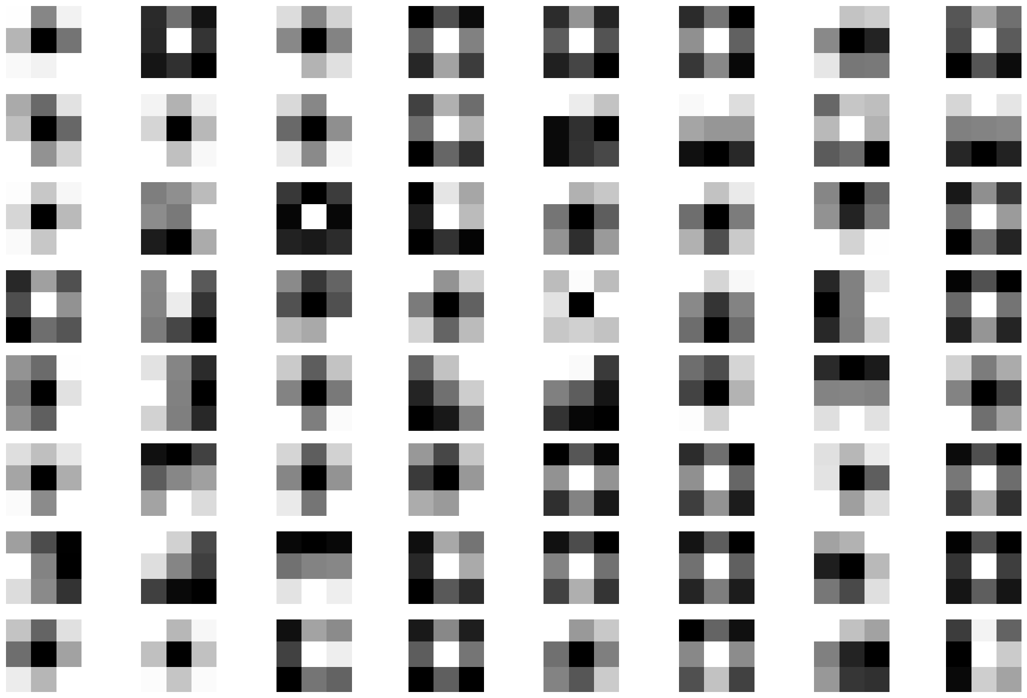

Figure 8: First Weight Map for VGG 16 Model

VGG's first weight map is similar to the custom model weight map but it can be inferred that these basic features best represent the dataset because of its significant increase in accuracy.

5.2 Custom Model Feature Maps

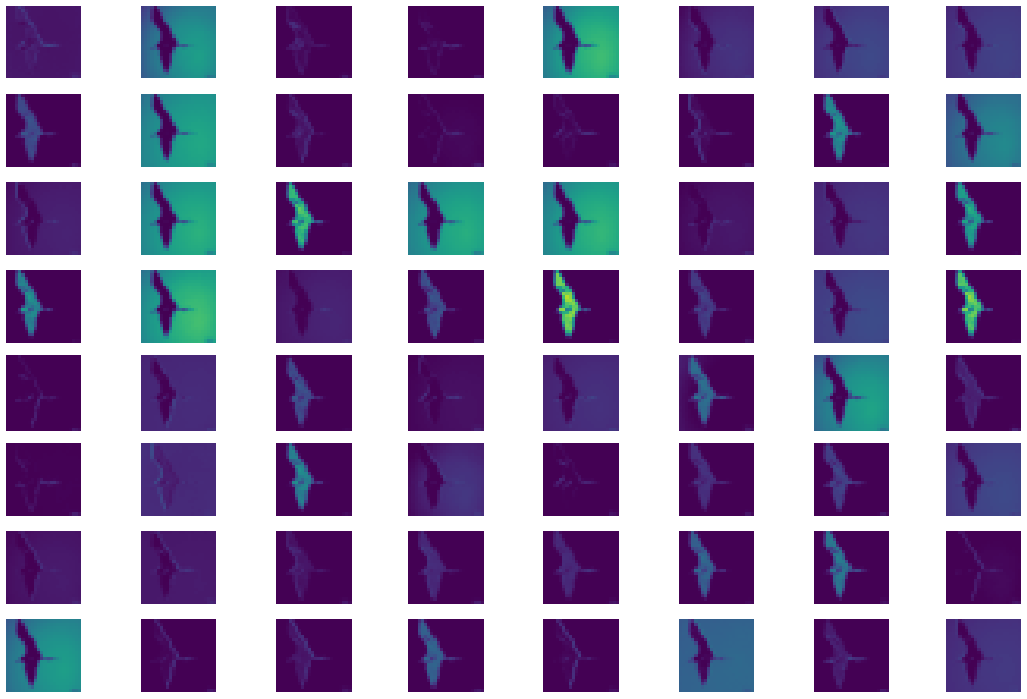
Figure 9: Feature maps for the first convolutional layer

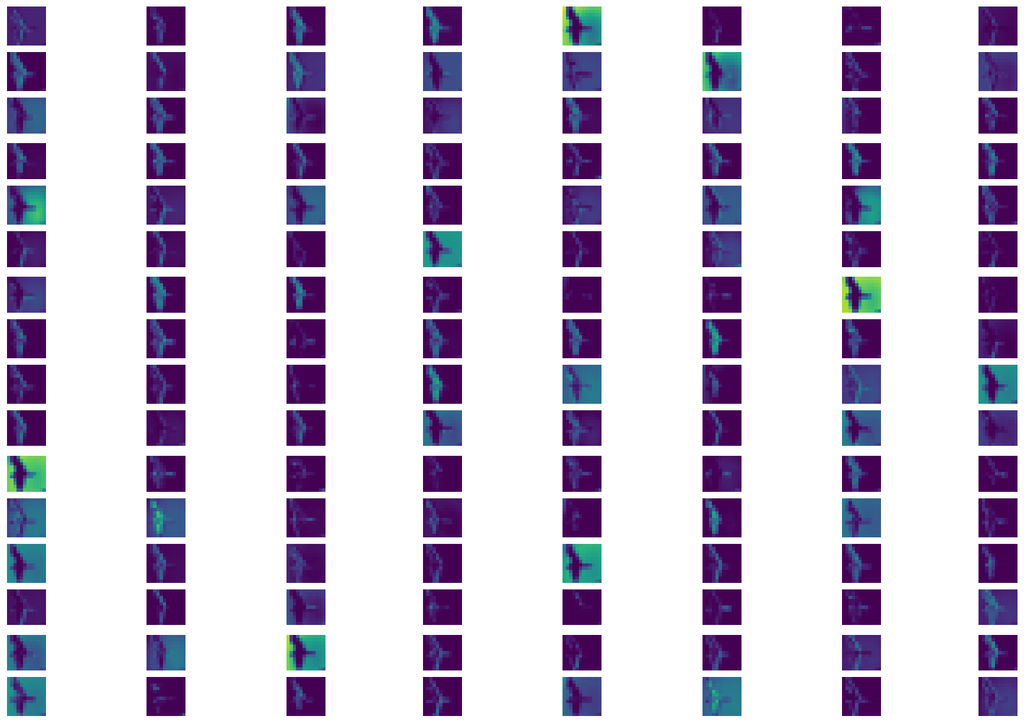
Figure 10: Feature maps for the middle (4th) convolutional layer

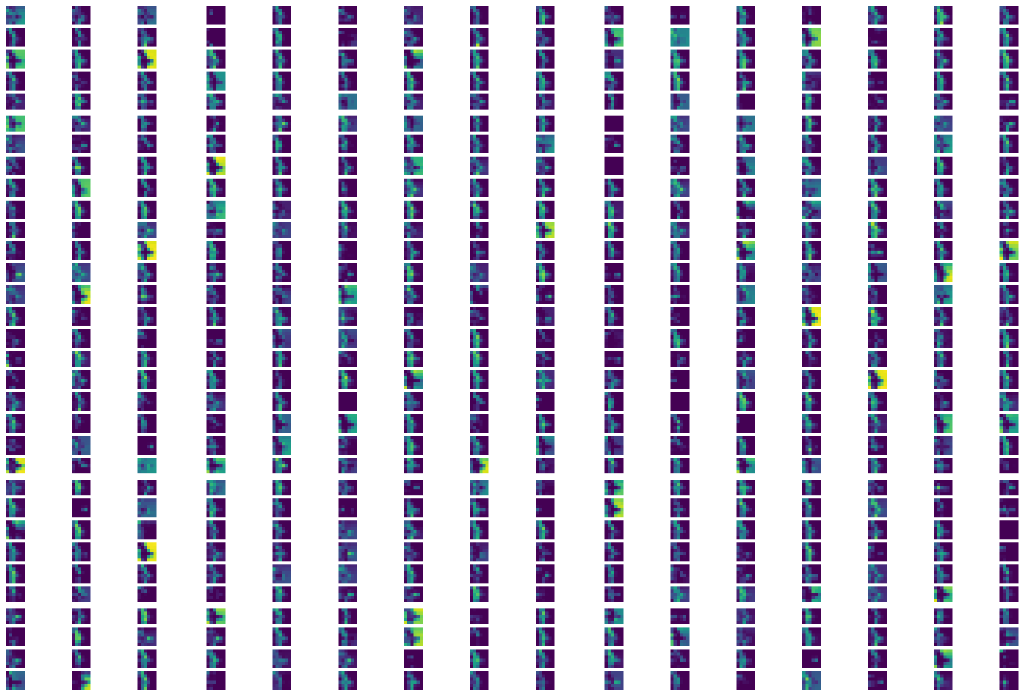

Figure 11: Feature maps for the last convolutional layer

As the layers increase, there is much more noise and contrast between the feature maps. For example, in the first layer feature map, there are some brighter and darker tones, bu in comparison to the last layer feature map, the last layer has many more filters that are very ”yellow”, signaling higher contrast. The last layer feature maps also have more noise and is more difficult to compare to the actual individual image relative to the first layer’s feature maps. However, many of the feature maps are very similar, seemingly identical in some, particularly in the last layer of the convolutional layer. Compared to the vgg16 and resnet 18 models, this may explain the lower accuracy in our custom model.

5.3 vgg16 Feature Maps

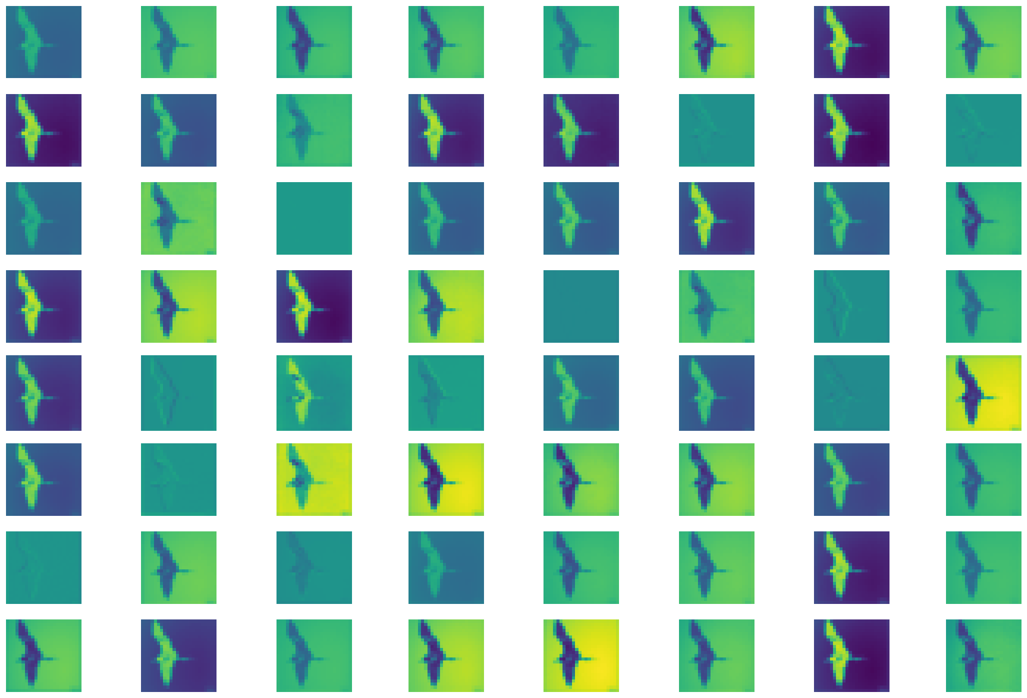

Figure 12: Feature maps for the first convolutional layer in VGG16

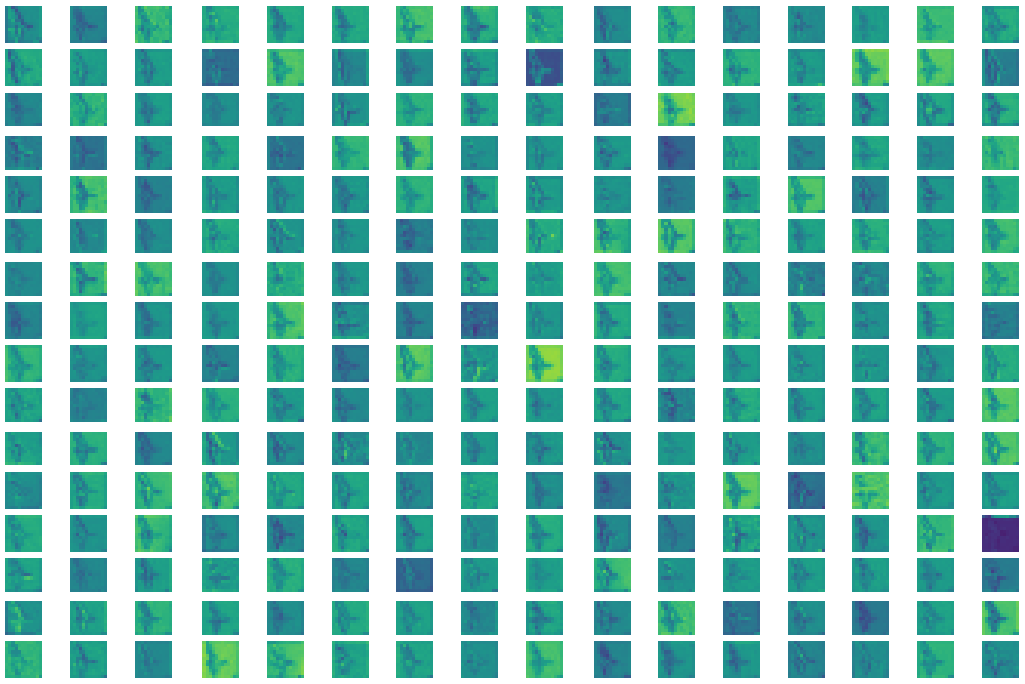

Figure 13: Feature maps for the middle convolutional layer in VGG16

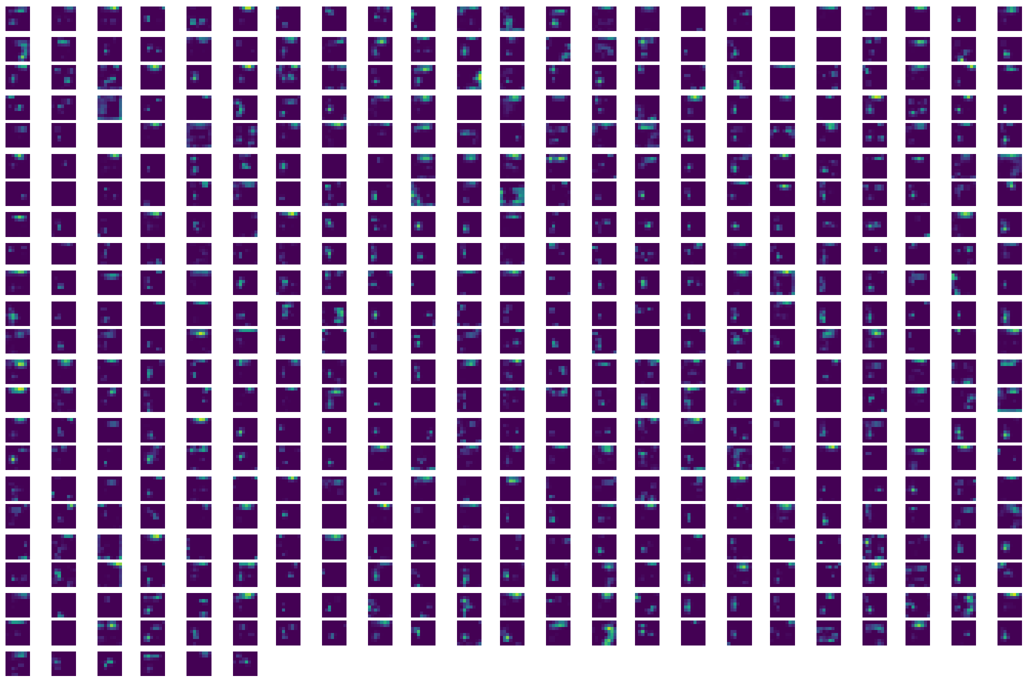

Figure 14: Feature maps for the last convolutional layer in VGG16

As the layer gets deeper, the feature maps show more abstract features that focus on different areas of the image.

5.4 resnet18 Feature Maps

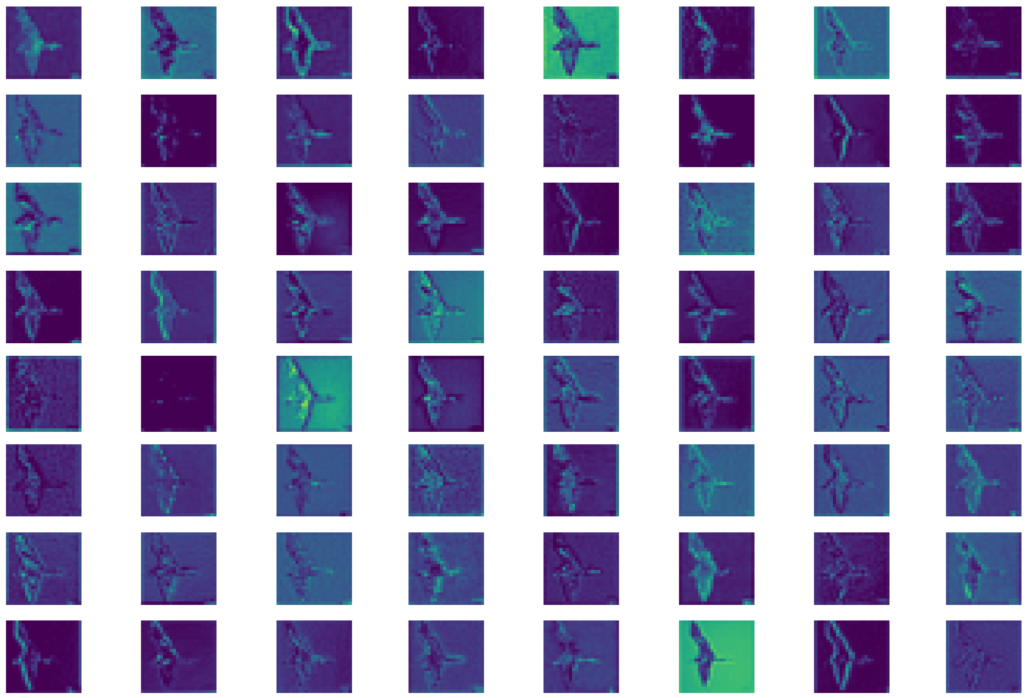

Figure 15: Feature maps for the first convolutional layer in Resnet18

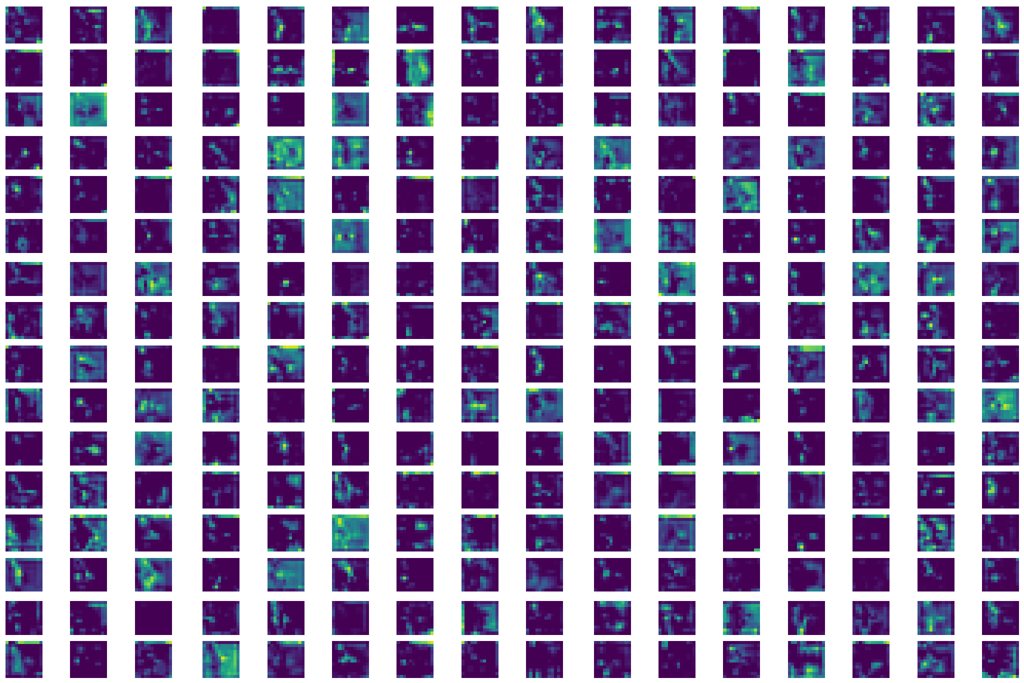

Figure 16: Feature maps for the middle convolutional layer in Resnet18

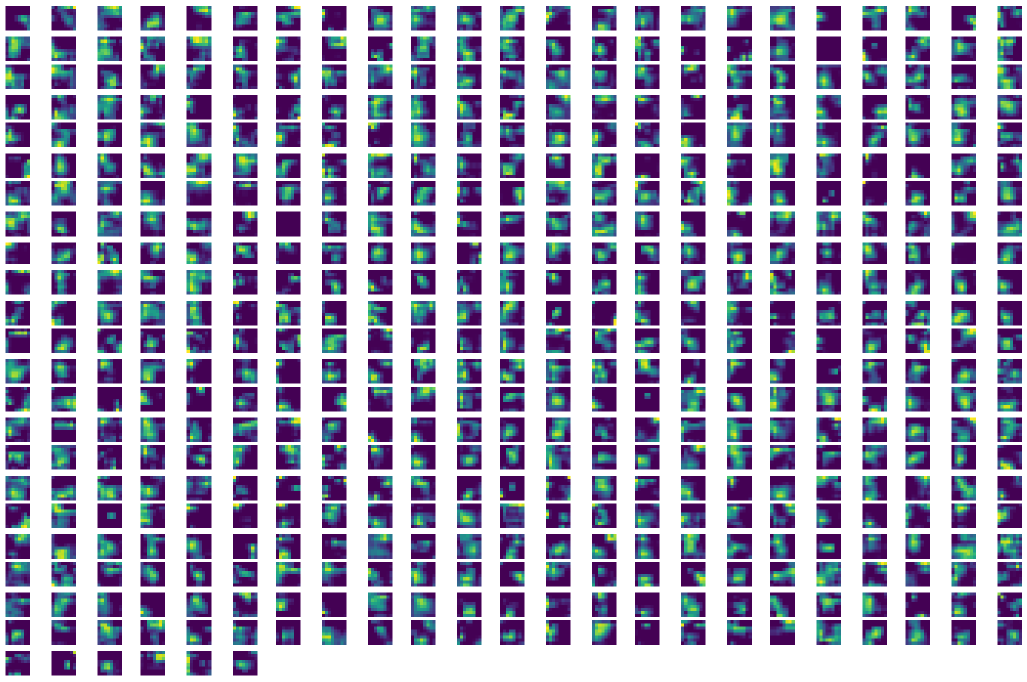

Figure 17: Feature maps for the last convolutional layer in Resnet18

Resnet was our best performing model and we are able to see how in the feature maps. Compared to the last feature maps of the custom model and VGG, resnet shows a lot more distinct and unique abstract features that make up the various categories. This is shown in the contrast between the light and dark colors.

6 Discussion

Our baseline model consisted of four convolutional layers. We found that this was not very accurate, and as we added two more convolutional layers in our custom model, performance on the test set improved dramatically. Our baseline model was too simple to accurately capture the complexity of the images. By adding more convolutional layers, we captured more of the features in the images that would help distinguish between the different classes. To improve our custom model, more convolutional layers could be added. However, we believe that while we shoul.d add more convolutional layers, the output channels of each should be diminished. We found that by testing with large number of output channels, the data was likely to overfit more heavily. However, there may be more ways to decrease overfitting, should we in the future decide to continue to have large number of output channels. These include more dropout and increasing regularization and weight decay. To more accurately test the performance of our model, a larger dataset would be very beneficial. Data augmentation could be useful, such as flipping or transforming the images prior to entering the dataset into our model. In comparison, the baseline and custom model were much less complex and deep than the pre-trained models of resnet and vgg. Although those models were pre-trained, this may still be a partial reason of the discrepancy in accuracy between the pre-trained models and our baseline and custom model.

Through transfer learning, we were able to realize the practicality of pre-trained models. Since our problem, bird classification, is very similar to the problems for VGG and resnet, we are able to keep the same features in each layer that allowed these models to gain such a high accuracy - especially in the layers towards the beginning. Early layers in CNNs usually detect the same basic features of images such as edges and shapes. We want those weights/filters to stay the same for our data instead of having to retrain all those weights again. However, we learned that it was advantageous to reset the weights in the last couple convolutional layers because those layers detected more abstract features that we wanted specific to our data. Keeping the learning rate low was also beneficial as we did not want to change the well-trained model too much. One drawback from using such deep neural networks was that we did not have nearly the same amount of data to train the model. Resnet and VGG both iterated over millions of data while our dataset was in the hundreds. In order to take full advantage of deep networks, we would need a much larger dataset.

Hyper-parameter tuning for our model consisted of primarily learning rate, weight decay, and learning rate scheduler. We found that epoch size of 50 was often a good epoch size to capture the data with our model, but avoid too much overfitting. Due to our data set size of 540 for training, the batch sizes we could use were limited, and small changes could change performance. We found that a batch size of 4 was the optimal amount in our testing. Learning rate, weight decay, and learning rate scheduler had to be tweaked for each model that we tested. In some instances, a larger learning rate was more beneficial to overcome smaller local minima and converge faster. Weight decay was used to penalize large weights, but would also depend on the learning rate. Learning rate scheduler was beneficial in the later epochs, to avoid too much fluctuations in the validation and allow for smaller convergence to a minima.

7 Author's Contributions

Mitchell: Worked on creating the baseline and custom models and weight/feature maps. Helped write the report including the model and experiments.

Moses: Worked on implementing and fine tuning VGG and Resnet. Helped write the report for VGG/Resnet models and weight/feature maps.

8 References

[1] Steve Branson and Grant Van Horn and Serge Belongie and Pietro Perona. Bird Species Categorization Using Pose Normalized Deep Convolutional Nets. 2014

[2] S. Nawaz, A. Calefati, M. Caraffini, N. Landro and I. Gallo, "Are These Birds Similar: Learning Branched Networks for Fine-grained Representations," 2019 International Conference on Image

and Vision Computing New Zealand (IVCNZ), Dunedin, New Zealand, 2019, pp. 1-5, doi: 10.1109/IVCNZ48456.2019.8960960.

[3] VGG16 - Convolutional Network for Classification and Detection. (2018, November 21). Retrieved November 15, 2020, from https://neurohive.io/en/popular-networks/vgg16/

[4] Cioloboc, F. (2019, January 04). Ideas on how to fine-tune a pre-trained model in PyTorch. Retrieved November 15, 2020, from https://medium.com/udacity-pytorch-challengers/ideas-on-how-to-fine-tune-a-pre-trained-model-in-pytorch-184c47185a20

[5] Finetuning Torchvision Models. (n.d.). Retrieved November 15, 2020, from https://pytorch.org/tutorials/beginner/finetuning$_{t}orchvision_{m}odels_{t}utorial.html$

[6] K. He, X. Zhang, S. Ren and J. Suan. Deep Residual Learning for Image Recognition. Dec 2015.