"It's a bird, it's a plane, it's definitely not a Frigatebird"

Using Convolutional Neural Networks to classify birds

Shubham Kulkarni
CSE Department
UC San Diego
La Jolla, CA 92092
skulkarn@ucsd.edu

Michael Wu
CSE Department
UC San Diego
La Jolla, CA 92092
m5wu@ucsd.edu

Abstract

We present a more advanced and accurate method of classifying images using a convolutional neural network. The aim of our project was to develop a model that would classify images successfully. We specifically looked to differentiate between 20 different categories of birds.

The dataset used in our experiment was the Caltech-UCSD Birds 200 dataset. Specifically, we used a small subset because of computational limitations. We used 600 images to train and 600 images to test.

The experiment was separated into 3 main stages:

1. *Baseline Model*- In this stage, we implemented a simple convolutional neural network to create a baseline for ourselves to test against. The simplistic network attained a test accuracy of $\mathbf{26.0}\%$.
2. *Custom Model*- In this stage, we built on our baseline model by adding depth to our initial network to improve our accuracy. Specifically, we went from 4 convolutional layers to 6 convolutional layers. This gave us a strong boost in accuracy. The custom network attained a test accuracy of $\mathbf{50.3}\%$.
3. *Transfer Learning* - In this stage, we used pre-trained state-of-the-art models like ResNet-18 and VGG16 to achieve significantly higher accuracy.
   VGG16 with batch normalization attained a test accuracy of $\mathbf{83.6}\%$ and ResNet-18 attained a test accuracy of $\mathbf{83.2}\%$.

Weight and Feature Maps - Finally, we generated weight maps and feature maps for our custom model and ResNet-18 and VGG16 so we could examine what these networks were learning and how they were making predictions.

1 Introduction

We used a series of advanced machine learning techniques to build and train our model.

1.1 Dataset

We were working with an incredibly difficult dataset. The current state-of-the-art model attains only 57.83% accuracy on the test set.

The images themselves are non-uniform in size. Most of the images are approximately 500 pixels wide by 300 pixels tall, with some images being over 400 pixels tall. Additionally, they are not consistent in terms of input channel count - most images are RGB, with a few grayscale images.

We dealt with these issues by reshaping them to all be 224×224 square images as ImageNet models expect, and changing them to all be 3 input channel (grayscale images would copy the channel into 3). We also normalized our images with the ImageNet mean and standard deviations (mean of $[0.485, 0.456, 0.406]$ and standard deviation of $[0.229, 0.224, 0.225]$ across the three channels.

1.2 Transfer Learning

For our transfer learning, we used state-of-the-art models from the ImageNet competition. In terms of models, we tested both VGG16 and Resnet18.

We used VGG16 with batch normalization, which uses multiple small convolutions sequentially to create large receptive fields with relatively low computational cost, to attain a test accuracy of 83.6%.

We also used ResNet-18, which uses residual connections to allow for very deep networks to train more effectively by creating skip connections that jump over layers, to attain a test accuracy of 83.2%.

VGG16 was the stronger model of the two, which lines up with our expectations and the results on the ImageNet dataset: VGG16 had 2% better top-5 error and 4% better top-1 error compared to ResNet-18 on the ImageNet test set.

These models had incredibly high accuracy, but we soon discovered that these results are not to be trusted. According to the Caltech-UCSD Birds 200 dataset documentation, the test set has overlap with the training set given in the ImageNet competition.

2 Related Work

To better understand the pre-trained models we were using, we found some background information on them. We learned about ResNet-18 from Kaggle [1] and VGG16 from Neurohive [2].

We explored the documentation and some previous papers for the dataset to help us better understand what we were working with. Most of the papers we found used attribute labels in some way to improve their model accuracy.

One paper we found interesting was Visual Recognition with Humans in the Loop [3]. The authors deal with how we can merge computer vision with human interaction to increase our accuracy on predictions. The authors found that using human interaction gave a significant boost to accuracy, while computer vision was able to help bring down the amount of human response that was required to correctly identify a bird. The team used a probabilistic model to incorporate the user answers (allowing for some erroneous answers) combined with both an attribute based classifier as well as a support vector machine to recognize images using computer vision. We found that they had quite interesting results, and potentially our model would benefit from using attribute labels like theirs did.

Another paper we found interesting was Nonparametric Part Transfer for Fine-grained Recognition [4]. The authors of this paper used the idea of localized attribute labels to get state-of-the-art results on the dataset. The basic idea they used was that, given a test image, if you were to find training images with attribute bounding boxes given that were similar to the test images, you could use the same attribute bounding boxes on the test image to do better classification. The logic behind is that since the images are similar, the salient features for object recognition would be in the same regions. We found this to be a very interesting concept. Their implementation dealt with a lot of computer vision concepts that we didn't fully understand, but it was still an interesting read.

PyTorch documentation proved extremely useful for us to understand where in the training code we could make changes to most effectively improve our model. This included the docs for BatchNorm2d / Conv2d / MaxPool2d / AdaptiveAvgPool2d, the docs for PyTorch's weight initializers,

the docs for torchvision's pretrained models and how to use them, and of course the docs for the various optimizers and schedulers.

We used official documentation from Matplotlib's pyplot to help us generate the plots for loss and accuracy results.

Some slides that were quite helpful were Lecture 5, slides 49, 51, 54-55, where Professor Cottrell went over some useful tips on the result of spatial pooling, the way batch norm works and why it's effective, and general procedures for building and optimizing CNNs.

Additionally, the Stanford CNN course website had a lot of good information on how CNNs work, as well as useful case studies on award winning CNNs and their strategies.

3 Models

3.1 Baseline Model

The baseline model is constructed as follows:

1. Convolutional layer (64 output channels) with batch norm and ReLU activation function
2. Convolutional layer (128 output channels) with batch norm and ReLU activation function
3. Convolutional layer (128 output channels) with batch norm and ReLU activation function
4. 3×3 maxpool layer
5. Convolutional layer (256 output channels) with batch norm, ReLU activation function, and stride 2
6. 1×1 adaptive average pool through all the channels
7. a fully connected layer (1024 nodes) with dropout engaged at 0.5 and ReLU activation
8. a fully connected layer (20 nodes) that maps to outputs

All convolutions are 3×3 with a stride of 1 unless indicated.

The baseline model uses an interesting architecture. It avoids becoming excessively deep and sticks to only 4 convolutional layers. To counteract the low performance often given by simple models, it uses a lot of batch normalization. Specifically, it uses a batch normalization after every convolutional layer and before the non-linear ReLU activation.

3.2 Custom Model

To make improvements on the baseline model, we drew inspiration from the way VGG-16 repeated a pattern of multiple convolutional layers followed by a single max pooling layer. Due to resource limitations, we were not able to make as many of these patterns as VGG-16, but we were able to make enough to drastically improve the model's accuracy from the baseline.

1. Convolutional layer (64 output channels) with ReLU activation function
2. Convolutional layer (128 output channels) with ReLU activation function
3. 3×3 maxpool layer
4. Convolutional layer (128 output channels) with ReLU activation function
5. Convolutional layer (256 output channels) with ReLU activation function
6. Batch normalization
7. 3×3 maxpool layer
8. Convolutional layer (256 output channels) with ReLU activation function
9. Convolutional layer (512 output channels) with ReLU activation function and stride 2
10. 1×1 adaptive average pool through all the channels
11. a fully connected layer (1024 nodes) with dropout engaged at 0.5 and ReLU activation

12. a fully connected layer (20 nodes) that maps to outputs

All convolutions are 3×3 with a stride of 1 unless indicated.

Our model is more complex and has more convolutions than the baseline model, which boosts our accuracy significantly. We made a number of design decisions that helped us achieve our excellent results on the test set.

We wanted to have a more even amount of convolutions between max pools to stabilize learning, so we split it into 3 sets of two convolutions each. Within each set, the second convolutional layer has twice as many output channels as the first layer.

Additionally, we used more max pooling layers to make our model more invariant to translation within the images. All of our max pooling was with 3×3 and a stride of 3, so it shrunk our image by a factor of 3 each time. This, combined with the pattern our convolutional layers follow, results in a reduction of input size by approximately $5\times$ as an image moves from one pooling stage to the next.

Excessive batch normalization actually hurt our model's ability to generalize to the validation and test sets, so we decided to reduce the batch norm to only a single instance. This worked very well for us, likely because we were using quite small batches (batch size of 10), which may cause batch norm to be less consistent between minibatches.

Table 1: Custom Model Layers

Layer	**Input (H,W,C)**	$\rightarrow$	**Output (H,W,C)**	**Kernel**	**Stride**	**Padding**	**Activation**
Convolution 1	(224,224,3)	$\rightarrow$	(222, 222, 64)	3x3	1	0	ReLU
Convolution 2	(222, 222, 64)	$\rightarrow$	(220,220,128)	3x3	1	0	ReLU
Max pooling 1	(220,220,128)	$\rightarrow$	(73,73,128)	3x3	3	0	ReLU
Convolution 3	(73,73,128)	$\rightarrow$	(71,71,128)	3x3	1	0	ReLU
Convolution 4	(71,71,128)	$\rightarrow$	(69,69,256)	3x3	1	0	ReLU
Batch Norm 1	(69,69,256)	$\rightarrow$	(69,69,256)				
Max pooling 2	(69,69,256)	$\rightarrow$	(23,23,256)	3x3	3	0	ReLU
Convolution 5	(23,23,256)	$\rightarrow$	(21,21,256)	3x3	1	0	ReLU
Convolution 6	(21,21,256)	$\rightarrow$	(10,10,512)	3x3	2	0	ReLU
Average pooling 1	(10,10,512)	$\rightarrow$	(1,1,512)	10x10	1	0	ReLU
Fully Connected 1	(1,1,512)	$\rightarrow$	(1,1,1024)				ReLU
Fully Connected 2	(1,1,1024)	$\rightarrow$	(1,1,20)				ReLU

3.3 VGG16

VGG16 accepts a 224x224 RGB image for input. It uses a series of convolutional layer groups, which consist of 2 or 3 convolutional layers (with ReLU activation) followed by a max pool layer, to create large receptive fields with relatively low computational cost. The number of output channels leaving each of these convolutional layer groups increases as you go deeper into the model, but the height and weight of each layer decreases.

Following these groups are three fully connected layers (with ReLU activation), which result in 1000 output channels to be used by the final softmax layer to make the final classification for the input message.

3.4 ResNet-18

ResNet-18 uses a different approach to train the model: Deep residual learning.

One of the challenges of using too many layers is that the model may become too difficult to train. ResNet-18 works around this issue by creating "leapfrog" connections that jump over layers, which allows for ResNet-18 to train with 20 convolutional layers (to VGG-16's 13) while maintaining a high accuracy on the ImageNet dataset.

4 Experiments

4.1 Baseline Model

As expected, the baseline model performed modestly on the dataset. It struggled to generalize well from the training set to the validation set, as especially seen with the unstable changes in the model's validation accuracy over time. It finished with a test accuracy of 26.0%.

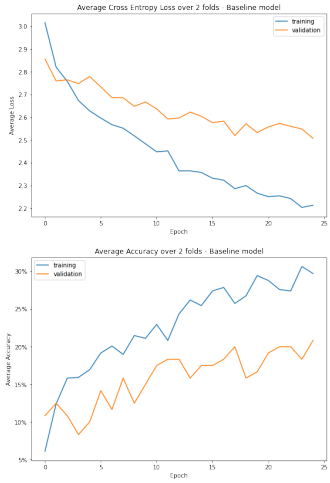

Figure 1: Cross Entropy loss and Accuracy for tuned model. Hyperparameters: 25 epochs, learning rate = 1e-4, batch size = 10, no early stopping, Adam optimizer with default β, no regularization

4.2 Custom Model

Our custom model got a final accuracy of 50.3% percent on the test set. From the plot, we can see that there is a fair amount of overfitting. However, even though validation loss was not decreasing as fast as training loss, we found that using early stopping gave bad results. This is probably because our validation set was tiny (60 images), and not very representative of the data as a whole.

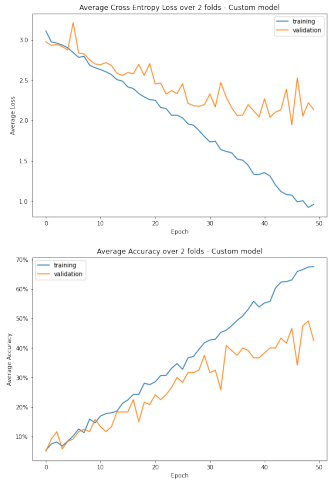

Figure 2: Cross Entropy loss and Accuracy for tuned model. Hyperparameters: 50 epochs, learning rate = 5e-4, L2 regularization = 5e-4, batch size = 10, no early stopping, Adam optimizer with default β, Uniform Xavier initialization

Table 2: Custom model experiments and results

Model Name	**Train Accuracy**	**Test Accuracy**
Original Settings	64.2%	37.8%
High LR	81.1%	30.2%
High LR and Reg	70.3%	44.7%
Low LR	55.4%	24.2%
Low LR and Reg	53.6%	29.6%
High Epochs	95.4%	45.3%
smol batch	100%	16.2%
BIG Batch	76.1%	39.4%
Goldilocks	77.2%	50.3%

Hyperparameters shared by all models (unless otherwise specified): models were trained for 50 epochs with batch sizes of 10.

Original Settings was a model that stuck with the original parameters from the baseline model: Adam optimizer with learning rate of 1e-4 and no L2 regularization. This model did okay, but wasn't highly effective. We found that this was due to a low learning rate that was not causing it to converge fast enough to finish during the allotted epoch count.

High LR used the following hyperparameters: Adam optimizer with learning rate of 1e-3 and no L2 regularization. This model did worse than the original settings model. This time, our learning rate was fine but our model was quickly overfitting to training data - something that was even more obvious by looking at the loss plots for this model.

High LR and Reg used the following hyperparameters: Adam optimizer with learning rate of 1e-3 and L2 regularization of 1e-3. This model did better than the original settings model, but still missed the required accuracy by quite a bit. We needed to figure out a better learning rate and weight decay combo that would fit quickly, but also not overfit the training data.

Low LR used the following hyperparameters: Adam optimizer with learning rate of 1e-5 and no L2 regularization. This model was a significant overcorrection in hindsight, but we felt that it would be good to figure out if lowering the learning rate could help reduce the amount of overfitting.

Low LR and Reg used the following hyperparameters: Adam optimizer with learning rate of 1e-5 and L2 regularization of 5e-6. We wanted to check if adding some amount of regularization would help our low learning rate models like it did our high learning rate models - this would help establish whether overfitting was occurring. We found that indeed low learning rate models were also overfitting quite a bit, as shown by the jump in test accuracy when adding regularization to the *Low LR* model.

High Epochs used the following hyperparameters: Adam optimizer with learning rate of 5e-4 and L2 regularization of 5e-4 trained for 100 epochs. This model did quite well actually, but it took far too long to train and looking at the loss plots, we saw a significant overfitting (validation loss was going way up while training loss was dropping). We knew that we had to revise our model to obtain these results without such significant overfitting.

smol batch actually had a significant departure in model design, but we felt it was important to mention since it was a big part of how we created our final model design. This model had a batch norm layer after every single convolutional layer for a total of 6 batch norms. It was trained with the Adam optimizer with learning rate 3e-4 and weight decay 3e-4. The data was presented in batch size of 1 (single element each time). This model was highly overfitted, which it turned out was a result of the batch norm layers interacting strangely based on training mode vs validation mode. This result led us to reduce the amount of batch normalization we were doing, and basically sparked our final model design idea.

BIG batch had the same significantly different model design as *smol batch* (6 batch norms - 1 after each convolution). It was trained with the Adam optimizer with learning rate 3e-4 and weight decay

3e-4. The data was presented in batch size of 10. This model was actually much more effective than *smol batch*, which led us to switch to a batch size of 10.

Goldilocks is our crown jewel - our claim to fame if you will. This model was trained with the Adam optimizer with a learning rate of 5e-4 and L2 regularization of 5e-4. This model was highly effective with a test accuracy of 50.3%, even beating the required test accuracy of 47%.

4.3 ResNet-18

ResNet-18 attained a test accuracy of 83.1% and a test loss of 0.62.

We can see that ResNet-18 didn't have a significant difference between finetuning and just changing the weight on the final layer. The only benefit was that finetuning gave us a more even loss curve, and quicker convergence. This is definitely a good thing, but finetuning the model took a while longer to train.

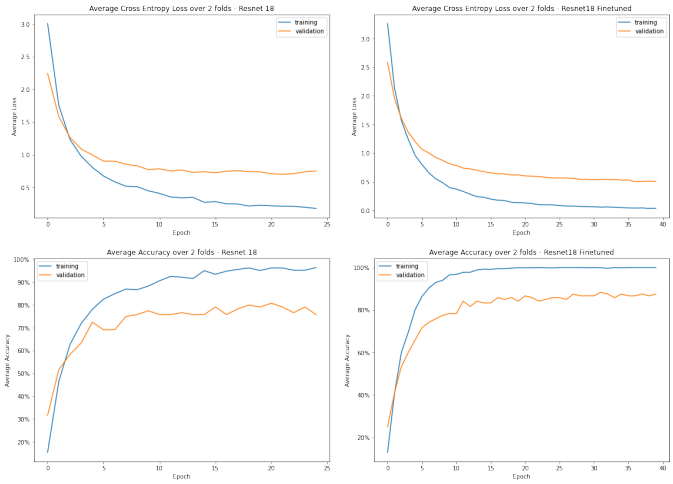

Figure 3: Cross Entropy loss and Accuracy for tuned model. Hyperparameters: 40 epochs, learning rate = 6e-6, L2 regularization = 1e-7, batch size = 10, no early stopping, Adam optimizer with default β, Uniform Xavier initialization. Left is without finetuning, right is with finetuning.

Table 3: Resnet 18 experiments and results

Model Name	**Train Accuracy**	**Test Accuracy**
Original Settings	100.0%	72.4%
Low LR	55.4%	24.2%
Regularization	97.7%	76.5%
Low Epochs	87.4%	45.3%
Goldilocks-locks	100.0%	83.1%

Hyperparameters shared by all models (unless otherwise specified): models were trained with batch sizes of 10.

Original Settings was a model that stuck with the original parameters from the baseline model: Adam optimizer with learning rate of 1e-4 and no L2 regularization trained for 40 epochs. This model did okay, but we felt we could do better. We weren't sure if the learning rate or L2 regularization would be the most effective to tweak.

Low LR used the following hyperparameters: Adam optimizer with learning rate of 1e-7 and no L2 regularization, trained for 40 epochs. This model did much worse than *Original settings*, but its loss graph was more evenly decreasing. We found that this learning rate was too low however.

Regularization used the following hyperparameters: Adam optimizer with learning rate of 1e-4 and L2 regularization of 1e-5, trained for 40 epochs. We found that this did model did better than the *Original Settings* model, but we still had some room to improve. We decided to combine the parameters here with the parameters from *Low LR* with some tweaking to come up with *Goldilocks-locks*

Low Epoch used the following hyperparameters: Adam optimizer with learning rate of 6e-6 and L2 regularization of 1e-7 trained for 15 epochs. It was essentially the *Goldilocks-locks* model with less training to see if we could train for less epochs and get equally good results. It turns out we could not, and the model performed quite badly indeed. The loss graphs were very close together though, which implied a lack of overfitting.

Goldilocks-locks is our piece de resistance. This model was trained with the Adam optimizer with a learning rate of 6e-6 and L2 regularization of 1e-7 for 40 epochs. This model was highly effective with a test accuracy of 83.1%, even beating the expected test accuracy of 80%. The name comes from the fact that ResNet is a residual network - the "locks" in "Goldilocks" is residual in the name.

4.4 VGG16

VGG16 attained a test accuracy of 83.6% and a test loss of 0.58.

We can see that VGG16 had a more pronounced difference caused by finetuning. This is probably because the model architechture is linear, and the sheer number of weights makes it easier to improve accuracy by finetuning for our problem. Both models arrive at roughly the same solution, but the finetuned model converges much faster an has a much smoother loss curve. Like ResNet-18, the time cost for finetuning was not worth it - especially since VGG had so many more weights to adjust that the time difference was even more significant.

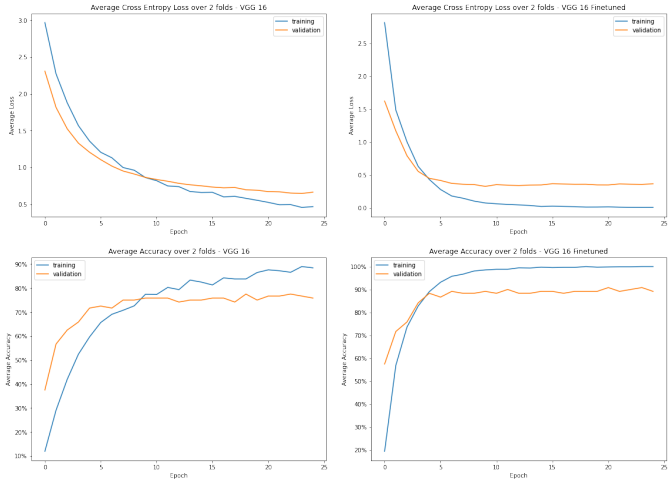

Figure 4: Cross Entropy loss and Accuracy for tuned model. Hyperparameters: 50 epochs, learning rate = 5e-4, L2 regularization = 5e-4, batch size = 10, no early stopping, Adam optimizer with default β, Uniform Xavier initialization

Hyperparameters shared by all models (unless otherwise specified): models were trained with batch sizes of 10 for 25 epochs each.

Original Settings was a model that stuck with the original parameters from the baseline model: Adam optimizer with learning rate of 1e-4 and no L2 regularization. This model did fairly well, but had a significant divergence in terms of training and test loss. We felt that with some hyperpa-

Table 4: VGG16 experiments and results

Model Name	Train Accuracy	Test Accuracy
Original Settings	100.0%	72.4%
Low LR	91.4%	70.2%
High LR	95.4%	45.5%
Regularization	97.4%	78.5%
VGGoldilocks	99.5%	83.6%

rameter tuning, we could obtain better results. Specifically, a slightly lower learning rate or adding regularization might help.

Low LR used the following hyperparameters: Adam optimizer with learning rate of 1e-6 and no L2 regularization. We tried this model to see if a lower learning rate might do better. It did worse than the *Original Settings* model, but the training and test accuracies didn't diverge nearly as much.

High LR used the following hyperparameters: Adam optimizer with learning rate of 1e-3 and no L2 regularization. We tried this model to see if we could potentially fit our data faster. It was not very successful, and led to a lot of overfitting unfortunately.

Regularization used the following hyperparameters: Adam optimizer with learning rate of 1e-4 and L2 regularization of 1e-4. We tried this model to see if overfitting could be combated using regularization. This model did pretty well actually, and we decided to put our results all together and come up with the best possible model in *VGGoldilocks*.

VGGoldilocks is our magnum opus. This model was trained with the Adam optimizer with a learning rate of 1e-5 and L2 regularization of 1e-5. This model was highly effective with a test accuracy of 83.7%, even beating the required test accuracy of 80%.

5 Feature Map / Weight Map Analyses

This section will cover our findings based on generated feature maps and weight maps, model by model.

5.1 Custom Model

While the information available in our weight map is somewhat limited due to the smaller 3x3 size of our maxpool layers, there are still plenty of findings we can make.

For example, our weight map includes streaks or "L" shapes of high weight values. This could be the model looking for the wingspan of a bird, which could be straight across the image or oriented at a right angle, depending on the orientation of the subject bird in the image.

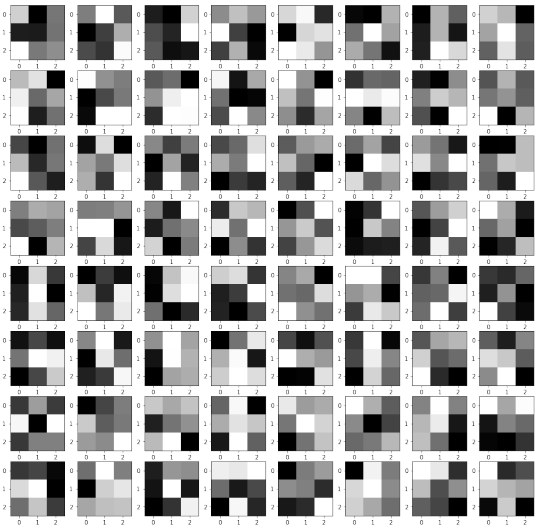

Figure 5: Weight map for custom model

After one convolutional layer, the different levels of brightness and contrast with each subimage in the feature map tells us how the model is identifying the different basic components of the sample image. For example, the first subimage (from the left) in the fourth row (from the top) has a dark background and light bird body, indicating that this feature is trying to identify the background.

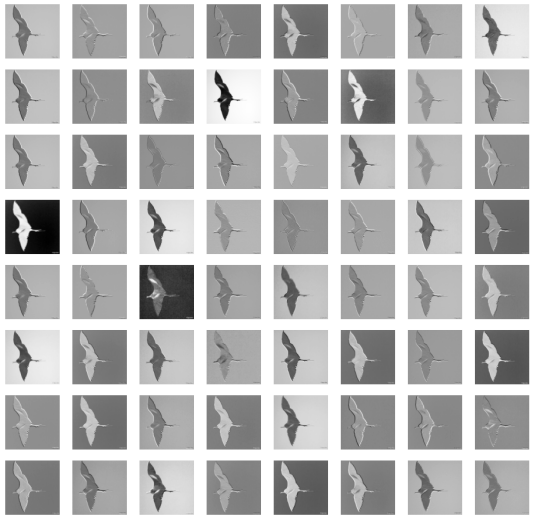

Figure 6: Feature map for custom model after 1 convolutional layer

After three convolutional layers, we can start to see more big-picture details of the input image. For example, the second subimage of the seventh row has a white outline along the bird's left-facing edge, indicating that the model may have identified a boundary separating the background from the bird.

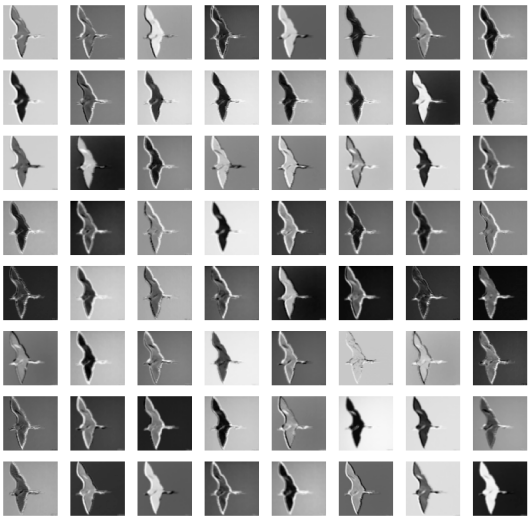

Figure 7: Feature map for custom model after 3 convolutional layers (approximately half)

The more blurred feature map after all six convolution layers seems to indicate that the model has now identified even bigger picture details about the image. For example, the last subimage of the fifth row has a broad weight distribution around the bird's apparent center of mass, indicating that the model may be using this feature to identify the center area of the subject bird.

Figure 8: Feature map for custom model after all 6 convolutional layers

5.2 ResNet-18

The larger 7x7 subimages in the weight map of ResNet-18 allows us to get more information about the model's behavior with each subimage.

For example, this weight map has smoother transitions between the higher and lower-weighted sections compared to that of our custom model. Consider the fifth subimage of the fourth row. This subimage looks like a multivariate Gaussian curve; the model could be indicating that the border is "more likely" to be a background and the center "less likely" to be so.

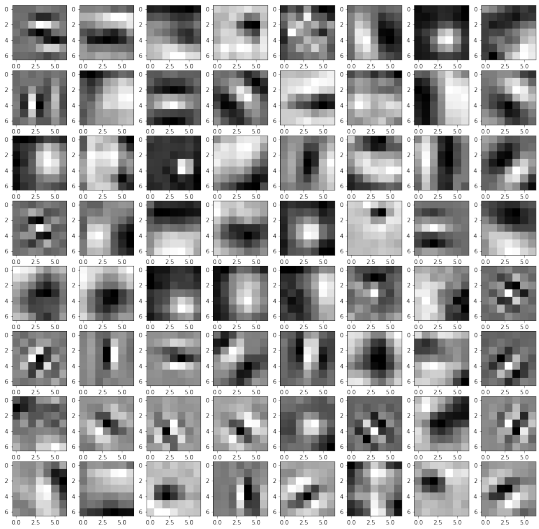

Figure 9: Weight map for ResNet-18

Compared to that of our custom model, ResNet-18's feature map has many more distinctive features after 1 convolutional layer. Whereas the variance in the "brightness" of each subimage is quite small for our custom model, ResNet-18 has more darkened and brightened subimages. This could be due to ResNet-18 not specializing in birds and thus having to use stronger pre-learned heuristics to generate its feature map.

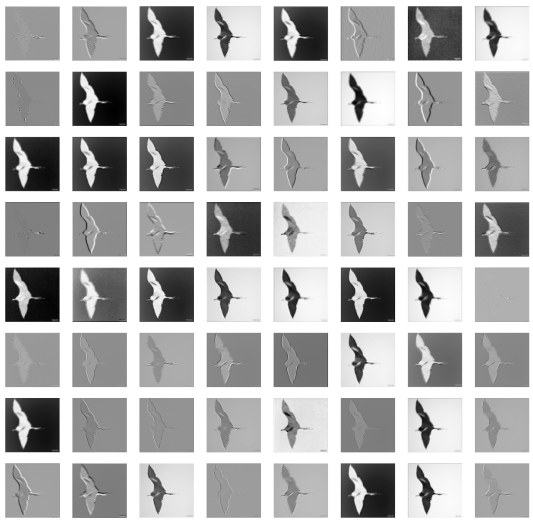

Figure 10: Feature map for ResNet-18 after 1 convolutional layer

The rougher nature of the subimages in the next feature map suggest that the model is searching for coarser information about the image; for example, nearly all the subimages have some indicator of the outline of the bird's wings with respect to the bird's body and the background.

Figure 11: Feature map for ResNet-18 after 10 convolutional layers (approximately half)

After all 20 convolutional layers, the subimages in ResNet's feature map no longer even look like birds. The model seems to be looking for a "ballpark" estimation of where the bird's center of mass is in the image.

Figure 12: Feature map for ResNet-18 after all 20 convolutional layers

5.3 VGG 16

Like our custom model, VGG16 uses 3x3 max pooling layers, so the weight map will also be 3x3. Unlike the custom model, however, we rarely see dark streaks (indicating a bird-like shape) like we did with the custom model. This could be because VGG16 is trained to identify a much wider variety of objects in images, so what the model emphasizes in the first convolutional layer may be different as well.

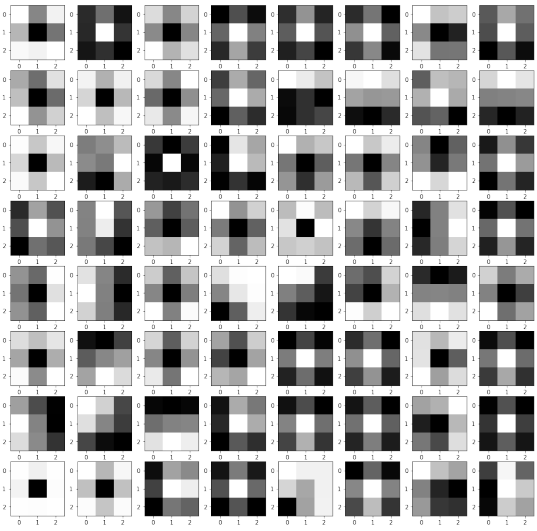

Figure 13: First layer convolution weights for a finetuned VGG16 mode. (convolutions corresponding to first input channel)

Compared to that of our custom model, VGG16's feature map also has many more distinctive features after 1 convolutional layer. Unlike ResNet-18, however, VGG16 has fewer subimages where the bird's body is indistinguishable from the background.

Figure 14: Feature map after passing test image through a single convolutional layer.

After half the convolutional layers, the feature map begins to resemble that of ResNet's feature map, in the sense that almost all the subimages have some level of emphasis on the outline separating the bird and the background.

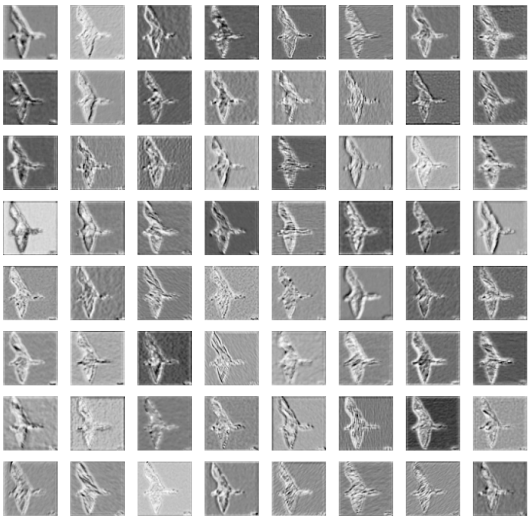

Figure 15: Feature map after passing test image through 6 convolutional layers (approximately half)

After passing through all the convolutional layers, almost every subimage in the resulting feature map has a few small black spots. These black spots do not give a lot of information about what the model is looking for; this may be because the model is not specifically looking at images of birds, so the model could be looking at specific regions of the image that can distinguish broader categories of objects.

Figure 16: Feature map after passing test image through all 13 convolutional layers

6 Discussion

6.1 Models

While not very comprehensive, the strengths and weaknesses of the baseline model gave us hints as to what improvements we could make towards building our custom model. It also allowed us to observe how the weaknesses of the baseline model were addressed in the pre-trained models VGG16 and ResNet-18.

The custom model was interesting to work on because we had many options on how to move forward with improving the baseline model design - our only limit was computational capability.

The biggest challenge with finding the right architecture for our model was trying to balance learning speed and accuracy. With more layers, we could have likely reduced training loss more, but at the risk of overfitting; with too few layers, we do not get enough of an accuracy improvement from the baseline model. Also, because we made our batch sizes small to help the model decrease its learning time, the batches were not large enough to say with confidence that one mini-batch was "similar enough" to another. So, excessive batch normalization made our model inconsistent when learning from batch to batch. We addressed this by only having one stage of batch norm instead of three.

VGG16 was a very deep model, but it was relatively simplistic in terms of architecture. It works in a very linear fashion. Combining this with its depth meant that the model was extremely large - the model was approximately 550 MB on disk. It was slightly slower to propagate through the network than ResNet, likely because it involved much more matrix computation. The large size did come with a positive though - it was slightly more accurate on test data than the ResNet model. Overall, we felt that the slight increase in test accuracy was dwarfed by the excessive size and training time for the model.

ResNet 18 is also a deep model, but it has a pretty complicated architecture. It uses skip connections to improve performance. ResNet-18 specifically uses 4 layers consisting of two 3x3 convolutions. The last 3 layers have a 1x1 convolution with stride of 2 to downsample the image. It's a much smaller model - 50 mb on disk (1/10th of the VGG16 network). This leads to a much faster training time. We preferred this model actually, even though it had lower performance.

Something to note is that we cannot trust the results of either ImageNet model on this data. The ImageNet training set actually overlaps with Caltech-UCSD Birds 200's test set, which means our test set is not previously-unseen data. If we were to create a true test set with only new data, these networks would get lower performance.

Something that might have possibly helped us with our custom model is to apply PCA on the image data first. Birds are very similar to each other in appearance - 2 wings, a beak, lots of pictures in flight. Since our network is tasked with differentiating birds, we want to give our model the most salient information to differentiate between birds. PCA has a knack for identifying the data that gives maximum variance - hopefully when applied to our dataset it would point us in the direction of the pixels that would be most helpful in differentiating between birds.

PCA on its own would probably have lackluster results because not all images are framed the same. We would likely need to apply some form of pooling with different dilations to achieve a decent amount of translational and scale invariance. Since birds are prone to flying in many positions, we should build in some amount of rotational invariance prior to running the PCA.

6.2 Hyperparameter Tuning

We didn't run a grid search this time because there were simply too many variables to account for. Instead, we used our intuition and a pseudo random search (based on numbers that sounded good) to figure out what parameters would be most effective for optimization.

Learning rate was pretty important for Adam, as well as the type of weight initialization (Xavier Uniform gave the best results but we also tried Normal and Xavier Normal). We could have also experimented with Kaiming initialization if we had more time - this initialization strategy has had more success with ReLU-based deep networks.

Finding the right amount of L2 regularization was also quite difficult. It would have been interesting to see if different regularization types, such as L_1 or L_∞, would have led to better success

6.3 Design choices

One important design choice for us was to reduce the amount of batch normalization. We found that due to our memory constraints, we couldn't have large batch sizes. This caused our batch norm to perform badly, and had inconsistent results based on training mode vs evaluation mode. We spent

quite a lot of time trying to get batch norm working, but eventually gave up and reduced it to a single instance within the model. This was a good choice since it allowed us to focus on other design choices, like drawing from VGG16's idea of having layer groups followed by pooling.

7 Team contributions

Michael - I ran experiments to fine-tune the pre-trained models to optimize their test accuracy with the CalTech-UCSD dataset. I was responsible for keeping the style of the code consistent across the whole project to improve readability. I wrote up about half the report and made sure syntax and organization stayed consistent across Shubham's and my writings.

Shubham - I designed the custom model, and ran experiments to tune the hyperparameters for it. I also wrote utility functions to make training and evaluating models easier. I helped Michael write the other half of report and verified the mathematical figures.

8 References

1. ResNet-18 Description: https://www.kaggle.com/pytorch/resnet18
2. VGG16 Description: https://neurohive.io/en/popular-networks/vgg16/
3. Visual Recognition with Humans in the Loop http://vision.ucsd.edu/sites/default/files/Visipedia20q.pdf
4. Nonparametric Part Transfer for Fine-grained Recognition https://pub.inf-cv.uni-jena.de/pdf/Goering14:NPT