Diving *Deep* for Accuracy with Convolutional Neural Networks

Nicholas DeGroot
CSE 151B: Deep Learning
University of California San Diego
La Jolla, CA 92122
ndegroot@ucsd.edu

Jianchang Yu
CSE 151B: Deep Learning
University of California San Diego
La Jolla, CA 92122
j7yu@ucsd.edu

Rick Truong
CSE 151B: Deep Learning
University of California San Diego
La Jolla, CA 92122
rvtruong@ucsd.edu

Abstract

Using the open source machine learning framework Pytorch, we explored and built a few image classification Deep CNN models to classify food based on the Food 101 dataset. The image classification Deep CNN models we trained were a baseline convolution neural network (convnet), a custom convnet, a pre-trained Residual Network (ResNet-18), and Karen Simonyan's and Andrew Zisserman's pre-trained Visual Geometry Group convent (VGG-16). In our experiment, we decided to train our CNN models on only 20 categories of the dataset due to resource constraints. After training our models for 25 epochs, we found that our baseline model with default metaparameters resulted in a test accuracy of 37.9%, our initial custom model with default metaparameters yielded a test accuracy of 39.86%, the ResNet-18 model with no changes yielded a test accuracy of 68.32%, and the VGG-16 model with batch normalization and no changes yielded a test accuracy of 70.7%. We fine tuned the metaparameters of the models, such as experimenting with extra layers on the custom model, tuning the learning rate, including dropout, implementing augmentation, toggling selective weight freezing, etc., to improve performance. Finally, we visualized feature activation maps for each of our CNN models across different convolutional filters and found that the activation maps for our custom model tended to be more obfuscated, the activation maps for our VGG-16 model yielded clearer visualizations than the former, and the activation maps for our ResNet-18 model provided the most clear visualizations and the highest similarity to the human V1 visual cortex system.

1 Introduction

In our project, we built a baseline convolution neural network (convnet), a custom convnet, a pre-trained Residual Network (ResNet-18), and Karen Simonyan's and Andrew Zisserman's pre-trained Visual Geometry Group (VGG-16) convent wwre trained on the Food 101 dataset [1]. We built these CNNs in order to understand the basics of convolutional networks, how to architect and fine tune CNNs in order to improve performance for classification type problems, and to visualize the inner workings of a Deep CNN's hidden layers. Due to resource constraints, we decided to train our CNN models on only 20 categories of the dataset in our experiment. In the next paragraph, we will investigate a basic overview of the architecture and training of each model.

The architecture of the baseline model consisted of a total of four convolutional layers, two spatial pooling layers, and two fully-connected layers at the end of the model. This baseline model was initially designed using Cross Entropy loss, Xavier initialization, and the Adam optimizer during training. For our customized convnet, we construed our model to have a total of six convolutional layers, three spatial pooling layers, and two fully-connected layers at the end of the model. Our custom model was also designed using Cross Entropy loss, Xavier initialization, and the Adam optimizer during training. For our VGG-16 model, we used PyTorch's implementation of their vgg_16_bn with batch normalization. One change we made to the vgg_16_bn model was changing the last fully connected layer to output to 20 categories. For our Resnet-18 model, we used PyTorch's pre-trained Residual Network-18 model. Similar to PyTorch's vgg_16_bn model, we also configured PyTorch's ResNet18 last fully connected layer to output to 20 categories in our project.

Finally, we visualized feature activation maps for each of our CNN models across different convolutional filters. The activation maps for our custom model turned our obfuscated which corresponds to a lower test accuracy. Our activation maps for our VGG-16 model yielded clearer visualizations than our customized CNN. We observed that our VGG-16 model's higher test accuracy corresponded to this observation. Lastly, the activation maps for our ResNet-18 model provided the most clear visualizations and the highest similarity to the human V1 visual cortex system.

2 Related Work

The dataset we utilized came from a publication which attempted to classify discriminative components of some data through a Random Forest Ensemble [1]. Though we did not use Random Forests for image classification in our project, we found this image dataset helpful in our training of CNN models.

Throughout training of our various CNN models, we referred to Andrej Karpathy's "A Recipe for Training Neural Networks" [5] and Nathan Inkawhich's "Finetuning Torchvision Models" [4] to construct and fine tune our models.

Moreover, many of the mathematics required for our model's training and inference were discussed in a number of UCSD courses. We would like to express our gratitude for the "Introduction to Convolutional Networks" slides [3] and "Convolutional Networks, Part II" slides [2] presented during Gary Cotrell's Winter 2022 CSE 151B course . Both slide decks detail the fundamentals of convolutional networks and architecture design techniques that our model heavily relies on.

3 Models

3.1 Baseline model

Our baseline model (architecture laid out in Table 1) architecture was given to our team by Professor Cotrell. This baseline CNN model architecture consisted of a total of four convolutional layers, two spatial pooling layers, and two fully-connected layers at the end of the model. and we built off of our baseline model. For the convolutional layers, three of the four convolutional layers were placed before our first spatial pooling filter, which max pooled together convolutional features of the previous filter. The 4th and final convolutional filter lies after which follows with the 2nd and final spatial pooling layer, which uses adaptive averaging on the convolutional filters of the previous layer. Lastly, the baseline consisted of two fully connected linear layers at the end which resulted in classification for 20 categories of an image. Although one can tune hyperparameters of our baseline model manually, our baseline model was initially designed using Cross Entropy loss, Xavier initialization, and the Adam optimizer during training.

3.2 Custom model

For our custom model (architecture laid out in Table 3), we built off of our baseline model. While the requirement was only 3 changes that made an impact, we tried a variety of things we learned from class. The first thing that needed to be added was at least two extra convolutional layers. Instead of placing them haphazardly, we decided to place one after the conv3 and maxpool layers of the baseline. As it turned out from our experiment, this was the sweet spot to place 2 extra

Layer	In	Out	Kernel	Pad	Stride	Activation	Normalization	Dropout
conv1	3	64	3	0	1	ReLU	batch	No
conv2	64	128	3	0	1	ReLU	batch	No
conv3	128	128	3	0	1	ReLU	batch	No
maxpool1	128	128	3	0	1	N/A	N/A	No
conv4	128	128	3	0	2	ReLU	batch	No
adaptiveAvgPool	128	1	N/A	N/A	N/A	N/A	N/A	No
fc1 Linear	128	128	N/A	N/A	N/A	ReLU	N/A	Yes
fc2 Linear	128	20	N/A	N/A	N/A	ReLU	N/A	No

Table 1: Architecture for baseline model

layers, as other locations for new layers and adding another layer both increased runtime and had little effect on the improvement. From class, we also learned that pooling layers could help in making the model more invariant to transforms, make receptive fields larger, and potentially reduce overfitting. In addition, we learned in class that max pooling was empirically better than sum and average pooling, so we tried adding an extra maxpooling layer between conv2 and conv3 layers of baseline. This also ended up having a good effect on the accuracy of the model. In lecture, it was mentioned that changing the activation to swish (or SiLU in PyTorch) could lead to better results. However, through our experiments, there was negligible difference. Once the impactful architecture changes were figured out, we tried hyperparameter tuning. We ended up finding that a learning rate of 0.0005 was a good learning rate. This means that for our final model, we added 2 convolutional layers, 1 maxpooling layer, and changed the learning rate to 0.0005. All the other settings remained the same as the baseline model. This includes the criterion, which remained Cross Entropy loss, and the optimizer, which remained Adam.

Layer	In	Out	Kernel	Pad	Stride	Activation	Normalization	Dropout
conv1	3	64	3	0	1	ReLU	batch	No
conv2	64	128	3	0	1	ReLU	batch	No
maxpool1	128	128	3	0	1	N/A	N/A	No
conv3	128	128	3	0	1	ReLU	batch	No
conv4	128	128	3	0	2	ReLU	batch	No
maxpool2	128	128	3	0	1	N/A	N/A	No
conv5	128	128	3	0	1	ReLU	batch	No
conv6	128	128	3	0	2	ReLU	batch	No
adaptiveAvgPool	128	1	N/A	N/A	N/A	N/A	N/A	No
fc1 Linear	128	128	N/A	N/A	N/A	ReLU	N/A	Yes
fc2 Linear	128	20	N/A	N/A	N/A	ReLU	N/A	No

Table 2: Architecture for custom model

3.3 VGG16 with batch normalization

For our VGG16 with batch normalization model, we used PyTorch's pretrained vgg16_bn model, and the one change we made to the architecture was to replace the last fully connected layer with one that had an output size of 20, as that was the number of labels in our food101 dataset. When training and tuning this model, we found that when all the weights were frozen, the default settings given lead to an accuracy of about 68%. However, the finetuned results showed that the default settings left a lot to be desired as the model performed poorly with the default learning rate of 0.001. So, our strategy for transfer learning for vgg16 was to try four strategies of partial finetuning, selective weight freezing, data augmentation, and hyperparameter tuning. Partial finetuning so we can see if changing the weights can help the model get more accurate. Selective weight freezing to see if just updating certain weights will prove better than all frozen or all unfrozen. Data augmentation because we saw overfitting very soon into the training and augmenting data could give us more data as well as possibly help with overfitting. And finally hyperparameter tuning because we saw that VGG was very sensitive to certain learning rates during tuning. We tried all of these individually, then combinations of each to see which ones worked together well. In the end, the best VGG16 model

we came up with used selective weight freezing of the first convolutional layer, data augmentation using a wide selection of transforms, and a modified learning rate of 0.0001.

3.4 ResNet18

For our ResNet18 model, we used PyTorch's pretrained resnet18 model, and the one change to the architecture we made was to replace the last fully connected layer with once that had an output size of 20, as that was the number of labels in our dataset and the model wouldn't work with our dataset otherwise. When training and tuning this model, we implored a very similar strategy to what we did for the vgg16 model tuning. Once again, we tried 4 different things, partial finetuning, selective weight freezing, data augmentation, and hyperparameter tuning for all the same reasons as mentioned before. While the default parameters were already performing far better than the baseline model and our custom model, we once again saw overfitting at certain learning rates with fine tuning. However, we felt that fine tuning was necessary as no finetuning was capping out at around 70% test accuracy. To abate this overfitting, we added data augmentation with a number of transforms. In the end, our best ResNet18 model used partial finetuning by unfreezing all weights, data augmentation using a wide selection of transforms, and a modified learning rate of 0.0001.

4 Experiments

For our baseline model, we ended up with these train set and validation set curves (see Figure 1) along with a test accuracy of 37.9% and a test loss of 2.059. Notably in the loss and validation curves, we see that improvements start leveling out early, around 10-15 epochs in.

Figure 1: Loss and Accuracy plots for the baseline model

4.1 Custom

To improve on this in our custom model, we tried a multitude of things (see Table 3) from increasing the number of convolutional layers, changing activation functions, adding max pooling layers, different combinations of those, and finally hyperparameter tuning.

Our first model trained was what we called "custom1", which had an additional two convolutional layers added between conv3 and maxpool layer of the baseline model. Both of them had the same in

Name	Changes	Train	Validation	Test
baseline	None	acc=32.5% loss=2.219	acc=35.4% loss=2.131	acc=37.8% loss=2.059
custom1	Added two conv layers	acc=34.62% loss=2.155	acc=37.9% loss=2.054	acc=39.86% loss=1.982
custom2	Changing to swish	acc=32.87% loss=2.222	acc=35.5% loss=2.135	acc=37.02% loss=2.064
custom3	Added maxpool between conv2+conv3	acc=40.68% loss=1.937	acc=43.07% loss=1.836	acc=46.92% loss=1.756
custom4	Changed learning rate to 0.0006	acc=35.74% loss=2.139	acc=39.0% loss=2.038	acc=40.92% loss=1.974
custom5	Added 2 conv and maxpool layers	acc=47.4% loss=1.7820	acc=47.7% loss=1.727	acc=50.88% loss=1.616
custom6	Added 2 conv layers, maxpool, and swish	acc=45.6% loss=1.797	acc=45.6% loss=45.6%	acc=49.16% loss=1.673
custom7	Added 2 conv layers, max pool, changed lr to 0.0005	acc=49.25% loss=1.655	acc=50.0% loss=1.628	acc=52.44% loss=1.550

Table 3: Changes attempted for custom model tuning

channels and out channels of 3, kernel size of 3, and a default stride of 1. This change led to a final test accuracy and loss of 39.86% and 1.982, which is an improvement over the baseline test accuracy by 2%. This improvement is because now the model is deeper and there are more parameters to be learned.

Our second model trained was what we called "custom2", which changed all the activation functions of the convolutional layers and fully connected layers to be "swish" (insert graph here), or SiLU in PyTorch. In lecture, it was mentioned that replacing ReLU with swish yielded a 0.9% improvement for Mobile NASNetA and 0.6% improvement for Inception-ResNet-v2 on the ImageNet database. However, in our findings and our architecture, it seems that the change to swish wasn't enough of an impact and actually had a slight negative effect on our accuracy, resulting in a test accuracy of 37.02% compared to baseline's 37.8%. We reason that this could be due to either our data or that our model's architecture was different from that of ImageNet and the two mentioned state of the art models, resulting in a negligible difference in performance when changing to ReLU activations to swish. Our third model trained was what we called "custom3," where we added a max pooling layer between conv2 and conv3 layers of baseline. This max pooling layer had a kernel size of 3 as the model had been using kernel size 3, so we continued with this. We added this max pooling layer because it was mentioned in lecture that spatial pooling could aid in getting the model to be invariant to small transforms, creates larger receptive fields as we were representing more of the input through pooling multiple inputs into 1 value, and reduces dimensionality. When training, we noticed that the training time (at least on the DataHub GPUs) reduced by half compared to the baseline. This is because of the dimensionality reduction effect of pooling that reduced the number of parameters compared to the baseline without this extra pooling layer. This led us to a test accuracy and loss of 46.92% and 1.756, which was a huge improvement over the baseline's accuracy and loss of 37.8% and 2.059.

Our fourth model trained was what we called "custom4", where we adjusted the learning rate from the default 0.001 to 0.0005. This resulted in our model performing with a test accuracy and loss of 40.92% and 1.974, which was for sure an improvement over the baseline's performance. This improvement seems to be due to the fact that the default learning rate was too large, and reducing the learning rate allowed the model to be more precise in making weight changes.

Our fifth model, "custom5", we combined the changes made in "custom1" and "custom3", adding two convolutional layers before maxpool of baseline and another maxpooling layer after conv2 of baseline. This resulted in a test accuracy and loss of 50.88% and 1.616.

Our sixth model, “custom6”, we combined the changes made in “custom5” and “custom2”, changing the activation functions to SiLU in addition to the 3 layers added in “custom5”. This was because we saw a negligible difference between the performance of just changing to swish alone, so we wanted to see if there were any improvements when the model was deeper. This model resulted in a test accuracy and loss of 49.16% and 1.673, which is actually worse performance than that of “custom5”, so we believe that it could be due to our model’s architecture being different from the models that swish helped improve performance in.

Putting all of the changes that made an impact together, we arrived at our best model, which we’ll call “custom7”, which added two convolutional layers, one max pooling layer, and a learning rate change from 0.001 to 0.0005. This resulted in our best performance of 52.44% and 1.550 for test accuracy and loss. We see in the plots (see Figure 2) for this that improvements start to become minimal once we reach around epoch 20 and the loss and accuracy stay around the same for the next couple of epochs.

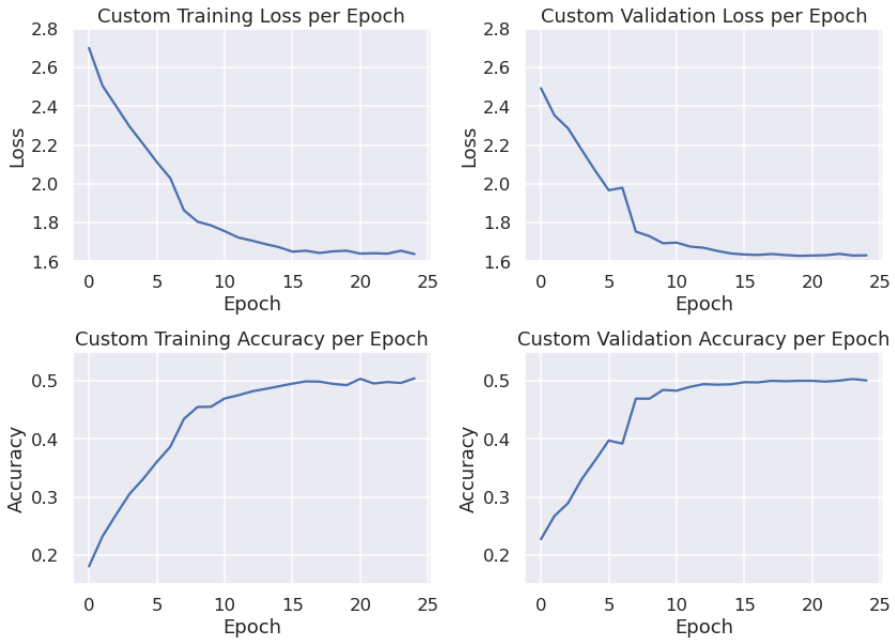

Figure 2: Loss and Accuracy plots for the best custom model

4.2 ResNet 18

To perform transfer learning on the resnet18 model from PyTorch, we tried a few different things (see Table 4), from changing the learning rate, to selective weight freezing, to partial finetuning, and data augmentation. We called the model without fine tuning (all weights frozen besides the last fully connected layer) and default parameters as “stock” and this is the basis for which we want to improve the model upon.

For the first change, we called our model “resnet1”, which unfroze all the weights and allowed the model to update the weights and perform gradient descent (also called fine-tuning). With this change, we saw a slightly improved test loss and accuracy of 71.22% and 1.085, but we also saw the model start overfitting mightily very soon about 10 epochs in. We also saw that with fine-tuning on, the model actually took slightly longer to train than “stock”, from 12 to 16 minutes on the DataHub GPU cluster.

For the second model, called “resnet2”, in addition to fine-tuning, we also changed the learning rate to 0.0001 from 0.001. This saw a dramatic improvement from “stock”, with our test accuracy and loss being 81.6% and 0.653. It seemed that the default learning rate was too large and making it smaller allowed the model to be more accurate. However, we once again saw overfitting around 10

Name	Changes	Train	Validation	Test
stock	None (weights frozen)	acc=71.09% loss=0.954	acc=67.97% loss=1.083	acc=68.32% loss=1.003
resnet1	partial finetune	acc=99.98% loss=0.007	acc=68.23% loss=1.257	acc=71.22% loss=1.085
resnet2	lr=0.0001, partial finetune	acc=99.99% loss=0.004	acc=77.87% loss=0.799	acc=81.6% loss=0.653
resnet3	lr=0.0001 (weights frozen)	acc=61.54% loss=1.415	acc=62.5% loss-1.387	acc=63.34% loss=1.326
resnet4	freeze conv1	acc=98.57% loss=0.082	acc=68.3% loss=1.192	acc=71.98% loss=1.074
resnet5	freeze conv1, augmentation, lr=0.0001	acc=82.68% loss=0.577	acc=79.17% loss=0.674	acc=80.34% loss=0.634
resnet6	partial finetune, augmentation, lr=0.0001	acc=82.32% loss=0.576	acc=79.17% loss=0.665	acc=80.7% loss=0.627

Table 4: Changes attempted for resnet18 model tuning

epochs in, so something else had to be done. We also saw once again that the model actually took slightly longer to train than "stock", from 12 to 16 minutes on the DataHub GPU cluster.

For our third model, called "resnet3", we were curious if a change in learning rate to the frozen model would help. So, we changed the learning rate from 0.001 to 0.0001 again, but leaving the weights frozen. We actually saw a dip in test performance compared to "stock", with accuracy and loss being 63.34% and 1.326. So this means that maybe the 0.001 might be too large when fine-tuning, but might be better for a frozen model that just needed the last layer to be trained. However, this was still worse than the result for fine-tuning.

For our fourth model, called "resnet4", we tried freezing weights of just the first convolutional layer of resnet18. This resulted in a slight improvement over partial finetuning, with test accuracy being 71.98% and loss being 1.074. Once again, we saw overfitting about 10 epochs in, so something had to be done about that.

For our fifth model, called "resnet5", we combined the best changes we've had so far, selectively freezing convolutional layer 1 and changing to a learning rate of 0.0001, but also adding data augmentation into the mix. We thought that data augmentation could help solve our overfitting issues as it would enlarge our dataset and provide different examples to help the model be more generalizable. For augmentation, we used the list of augmentations provided in the start code "log" folder (see Table 5). With this extra change, we actually saw a slight increase of training time from 12 minutes to 18 minutes on DataHub compared to "stock". As for our results, we saw an improvement, with test accuracy at 80.34% and test loss of 0.634. While the accuracy is lower compared to "resnet2", the loss is lower as well, along with overfitting being way less of an issue, with improvements over epochs leveling out around epoch 15 instead.

For our sixth model, called "resnet6", it was the same configuration as "resnet5", but with all the weights unfrozen instead. This resulted in a test accuracy of 80.7% and a test loss of 0.627. This is actually a slight improvement over "resnet5", which we think is due to the fact that with all these augmentations, it could actually be helpful to allow the model to perform gradient descent on that first layer.

We ended up choosing "resnet6" as our best model, even though it had the second highest testing accuracy because it had both the lowest test loss as well as not overfitting unlike "resnet2". Once again, the best performance of this model had a test accuracy of 80.7% and test loss of 0.627.

Looking at the plots for our best model "resnet6" (see Figure 3), we see that there is a dramatic amount of learning within the first 10-15 epochs and learning levels out around epoch 15, with the validation and training curves being very similar.

Transform	Detail
RandomResizedCrop(224)	Crop a random portion of image and resize it to a given size (in our case, 224)
RandomHorizontalFlip()	Horizontally flip the given image
RandomVerticalFlip()	Vertically flip the given image
RandomRotation(45)	Rotate the image by an angle
RandomAffine(45)	Random affine transformation of the image keeping center invariant
ColorJitter()	Randomly change the brightness, contrast, saturation and hue of an image

Table 5: Data augmentation used in ResNet and VGG (given in starter code logs)

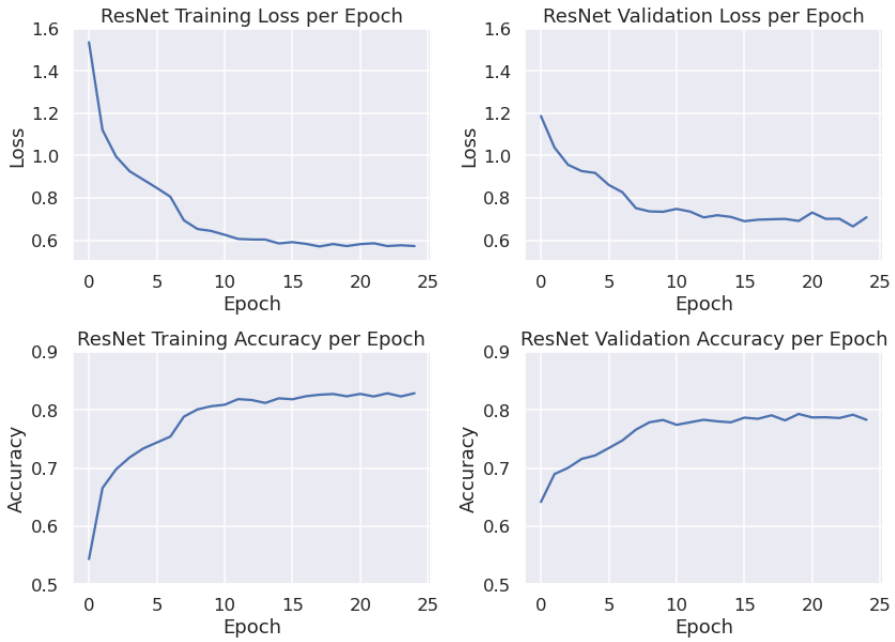

Figure 3: Loss and Accuracy plots for the best ResNet model

4.3 VGG 16 with Batch Normalization

To perform transfer learning on the vgg16_bn model from PyTorch, we tried a few different things (see Table 6), from changing the learning rate, to selective weight freezing, to partial finetuning, and data augmentation. We called the model without fine tuning (all weights frozen besides the last fully connected layer) and default parameters as "stock" and this is the basis for which we want to improve the model upon.

For our first model, "vgg1", we kept everything the same, but unfroze all the weights to allow fine-tuning. This resulted in actually a worse performance compared to "stock", with test accuracy being 49.9% and test loss being 1.740. We suspected that maybe this was another case of learning rate being too large. In addition, we saw the training time double on DataHub when we unfroze the weights, probably because we now have to do gradient descent on significantly more weights than when all the weights (besides the last fully connected layer) was frozen.

Thus, for our second model, "vgg2", we added another change on top of "vgg1", by changing the learning rate from 0.001 to 0.0001. This resulted in a dramatically improved performance, with test accuracy of 82.54% and test loss of 0.902. However, we saw a huge amount of overfitting around

Name	Changes	Train	Validation	Test
stock	None (weights frozen)	acc=67.87% loss=1.024	acc=68.9% loss=1.037	acc=70.7% loss=0.938
vgg1	partial finetune	acc=57.57% loss=1.321	acc=48.27% loss=1.672	loss=49.9% acc=1.740
vgg2	partial finetune, lr=0.0001	acc=98.88% loss=0.040	acc=77.3% loss=0.934	acc=82.54% loss=0.902
vgg3	freeze conv1	acc=53.24% loss=1.483	acc=44.0% loss=1.831	acc=46.76% loss=1.739
vgg4	partial finetune, augmentation	acc=33.93% loss=2.125	acc=35.03% loss=2.052	acc=35.5% loss=2.026
vgg5	partial finetune, augmentation, lr=0.0001	acc=84.11% loss=0.524	acc=81.57% loss=0.606	acc=82.9% loss=0.596
vgg6	freeze conv1, augmentation, lr=0.0001	acc=83.98% loss=0.512	acc=81.2% loss=0.605	acc=82.96% loss=0.590

Table 6: Changes attempted for vgg_16 model tuning

epoch 5 this time, with the training accuracy hitting 100%, meaning that something had to be done about overfitting.

For our third model, "vgg3", we tried selectively freezing the first convolutional layer of the model again, leaving everything else default, like we did with "resnet4". This resulted in a test accuracy of 46.76% with a loss of 1.739. This was worse than the performance of "vgg1", but it could be because of the learning rate, so we made a note to revisit this once changes are combined.

For our fourth model, "vgg4", we combined two changes, fine-tuning and data augmentation, leaving everything else default. Once again, we did data augmentation because we thought it could help solve our overfitting issues as it would enlarge our dataset and provide different examples to help the model be more generalizable. For augmentation, we used the list of augmentations provided in the start code "log" folder (see Table 5). This saw a huge dip in performance compared to "stock", getting a test accuracy of 35.5% and a test loss of 2.026. We believe this is due to the default learning rate being too large, causing a vanishing gradient, as with the fine-tuned model without augmentation.

For our fifth model, "vgg5", we changed the learning rate to 0.0001 on top of the changes made in "vgg4". This time, we saw a huge improvement over "stock", with test accuracy at 82.9% and loss of 0.596. The overfitting was no longer as dramatic and training accuracy ended also at around 84

For our final model and best performing model, "vgg6", we revisited freezing the first convolutional layer again along with data augmentation and changing the learning rate to 0.0001. This resulted in a slightly better performance over "vgg5" with test accuracy being 82.9% and loss being 0.590. We believe that the slight improvement could be 1 of 2 reasons, that the features in the first layer of VGG were already sufficient, or that it was a performance difference from run to run.

Looking at the plots for this best performing model (see Figure 4), we see that the changes are very dramatic in the first 10 epochs of training and the improvements slow down a lot more from epochs 10-15, and finally level out around epochs 15-20.

5 Feature map and weights analysis

5.1 Weight Maps

The weights for each model's first layer are presented in Figure 5. Each plot represents a different convolutional filter that was applied to each input image (across the RGB channels). To create each plot, the filter's weights were first shifted to be all positive, then normalized to a [0,1] scale against the max. This allows us to view which colors/patterns maximally activate the filter.

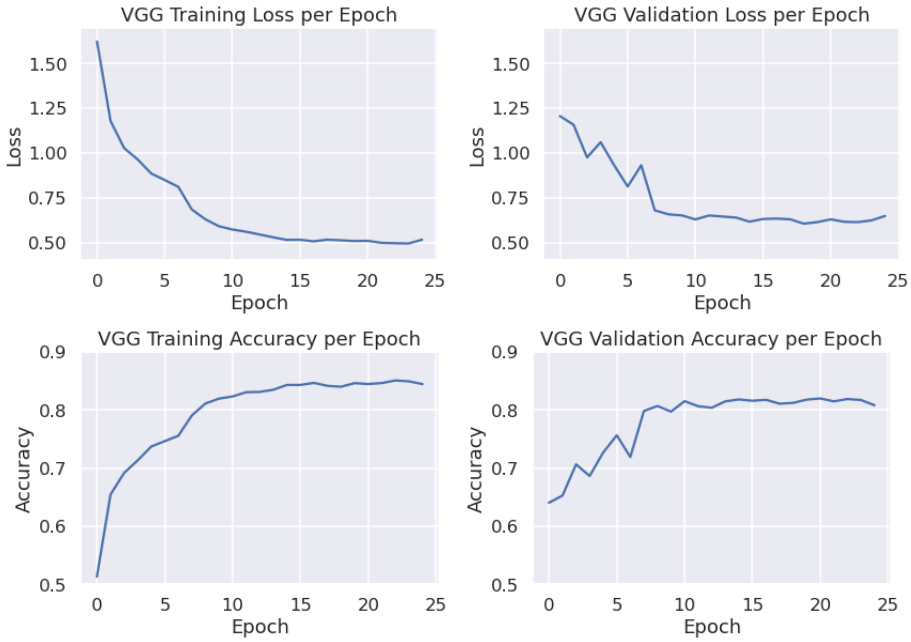

Figure 4: Loss and Accuracy plots for the best VGG model

For the custom model, we see a pretty disorganized array of filters. No filter seems to be searching for anything that comparable to what the human V1 visual cortex does, instead relying on seemingly random assortments of weights that have been learned in service of the particular task.

The story changes for VGG. Despite having the same 3x3 kernel, the VGG model learned to search for more organized patterns in the region. Many kernels seem to be looking for one color (i.e. blue spots in the bottom left) or bright spots (i.e. bright green spots at (3,3)). This is much more analogous to what we'd expect from V1 pathways in the brain.

ResNet bumps it up a notch by having its first kernel be larger at 7x7. Here, we can much more clearly see what the model is looking for. Multiple filters have been trained to look for lines of various orientations (e.g. horizontal, vertical, 45deg, -45deg) in various colors (the bottom filters activate particularly well to lines of blue and orange). This results in a model that has the highest similarities to the V1 system.

5.2 Feature Maps

Feature maps for all models were generated using the following procedure:

1. Run the first testing image through the model
2. Attach a hook to the corresponding layer, giving us the output/activations
3. Normalize the activation patterns by subtracting off the min, then dividing by the max.
4. Plot the output of each filter in a grid.

5.2.1 Custom Model Feature Map

The custom model's activation maps can be found in Figures 6, 7, and 8 for progressively deeper layers.

In the first layer (Fig 6), we see a variety of activation patterns. Some filters seem to have resulted in sparsely activated patterns, while others seem to have densely activated patterns. Many filters reacted to the top right corner particularly well (particularly those with otherwise sparse activations).

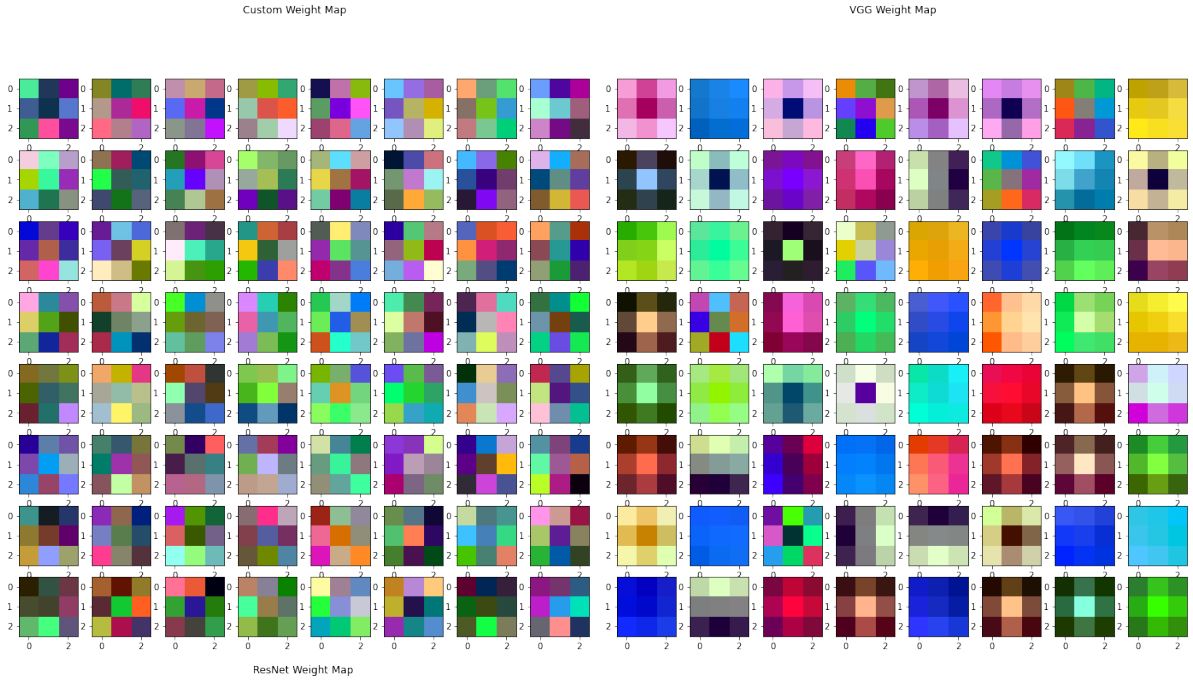

Figure 5: Weight Maps

In the middle layer (Fig 7), we start to see filters with more variance across their space. It's difficult to know what exactly they correspond to (since they correspond to features of features learned in service of the classification task), but features corresponding to the bottom of each filter seem to activate for the image.

In the final layer (Fig 8), each feature of feature has progressed to its high level form. Again, these are features learned in service of the classification task and can't be readily understood. This last layer has wildly varying activations that will end up getting fed into the last FC layer.

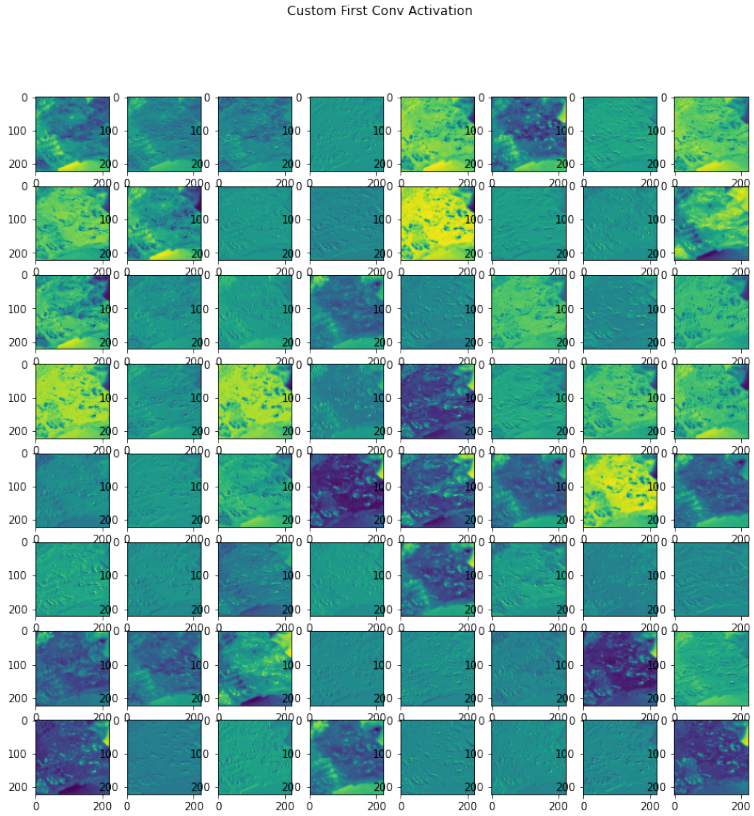

Figure 6: Custom First Activation Map

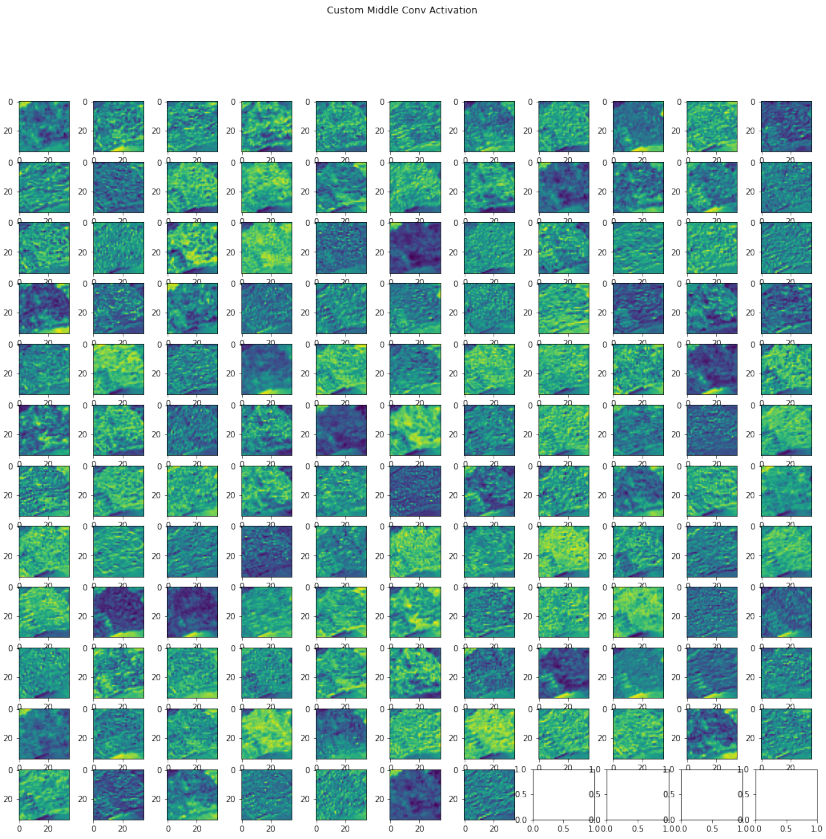

Figure 7: Custom Middle Activation Map

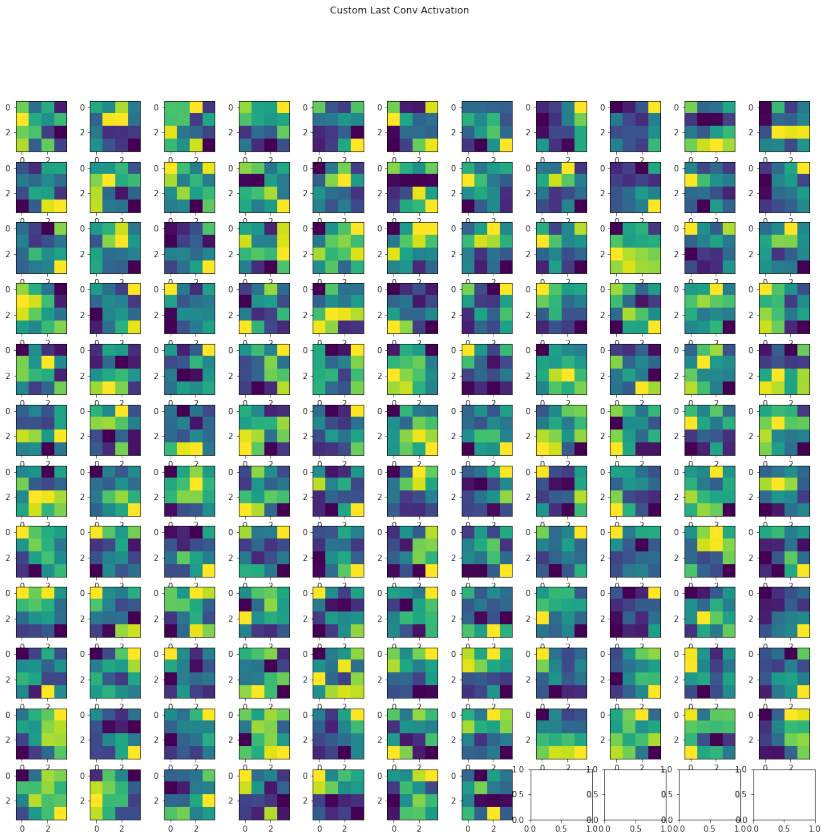

Figure 8: Custom Last Activation Map

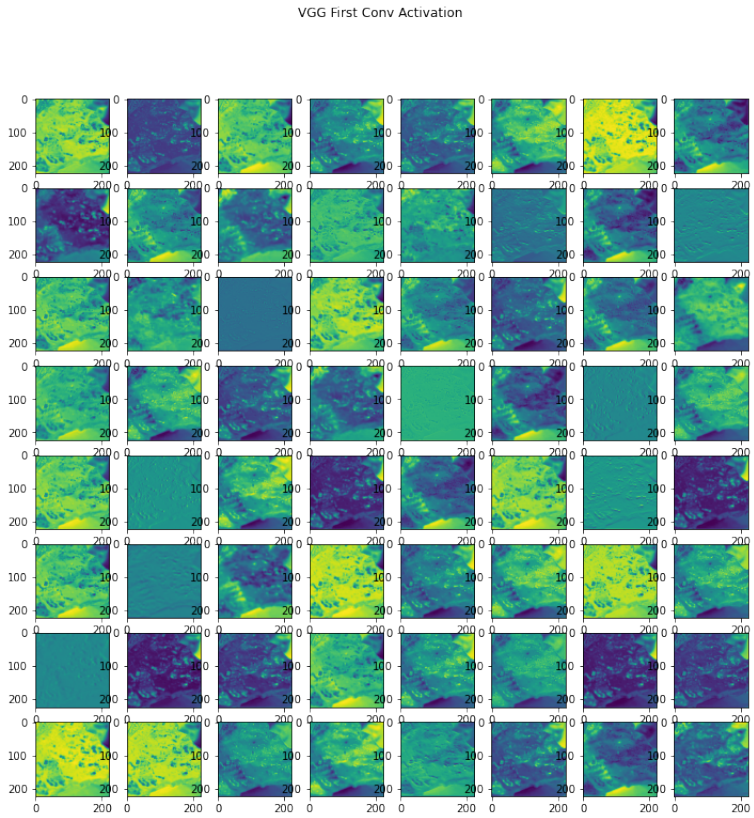

Figure 9: VGG First Activation Map

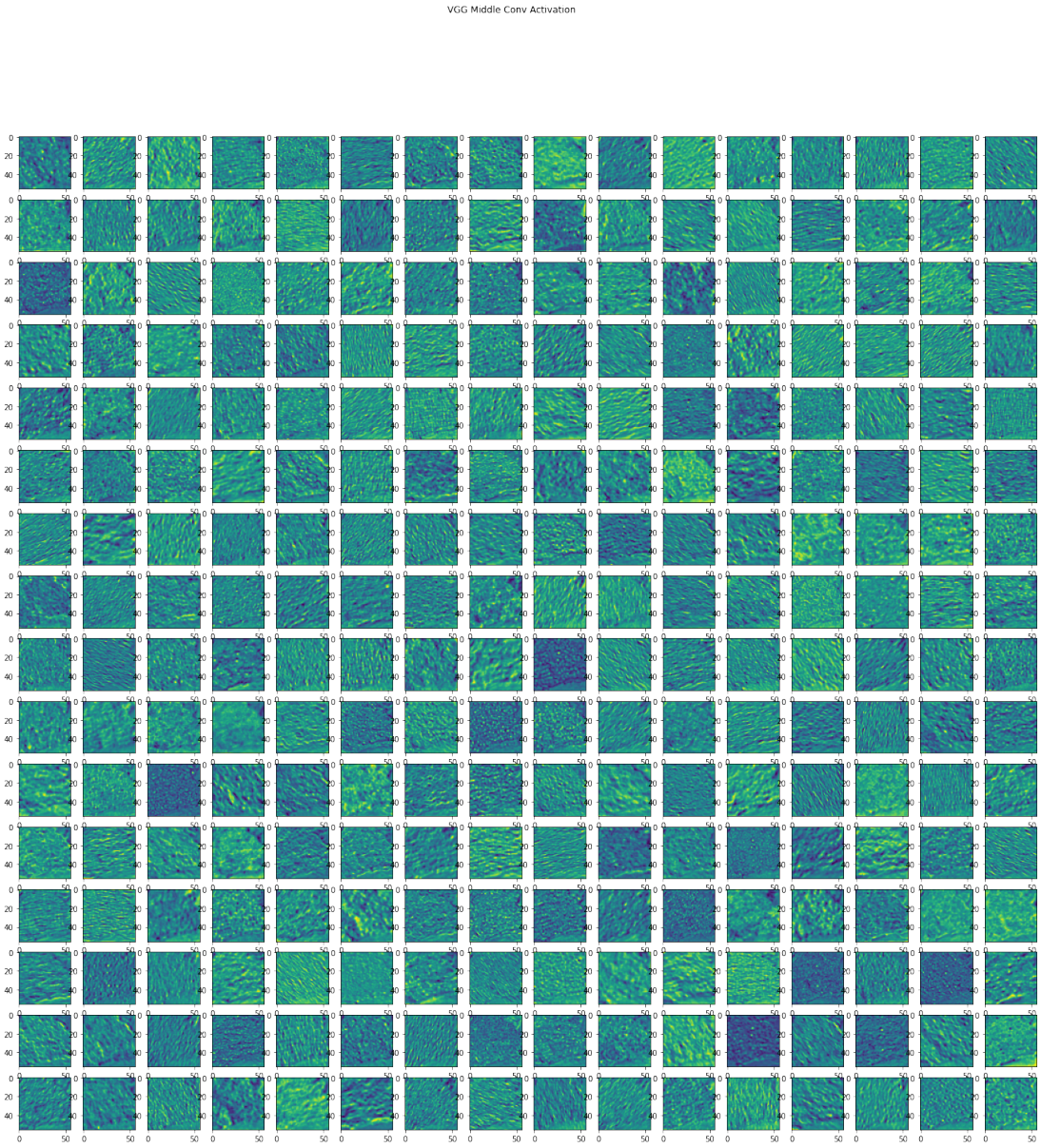

Figure 10: VGG Middle Activation Map

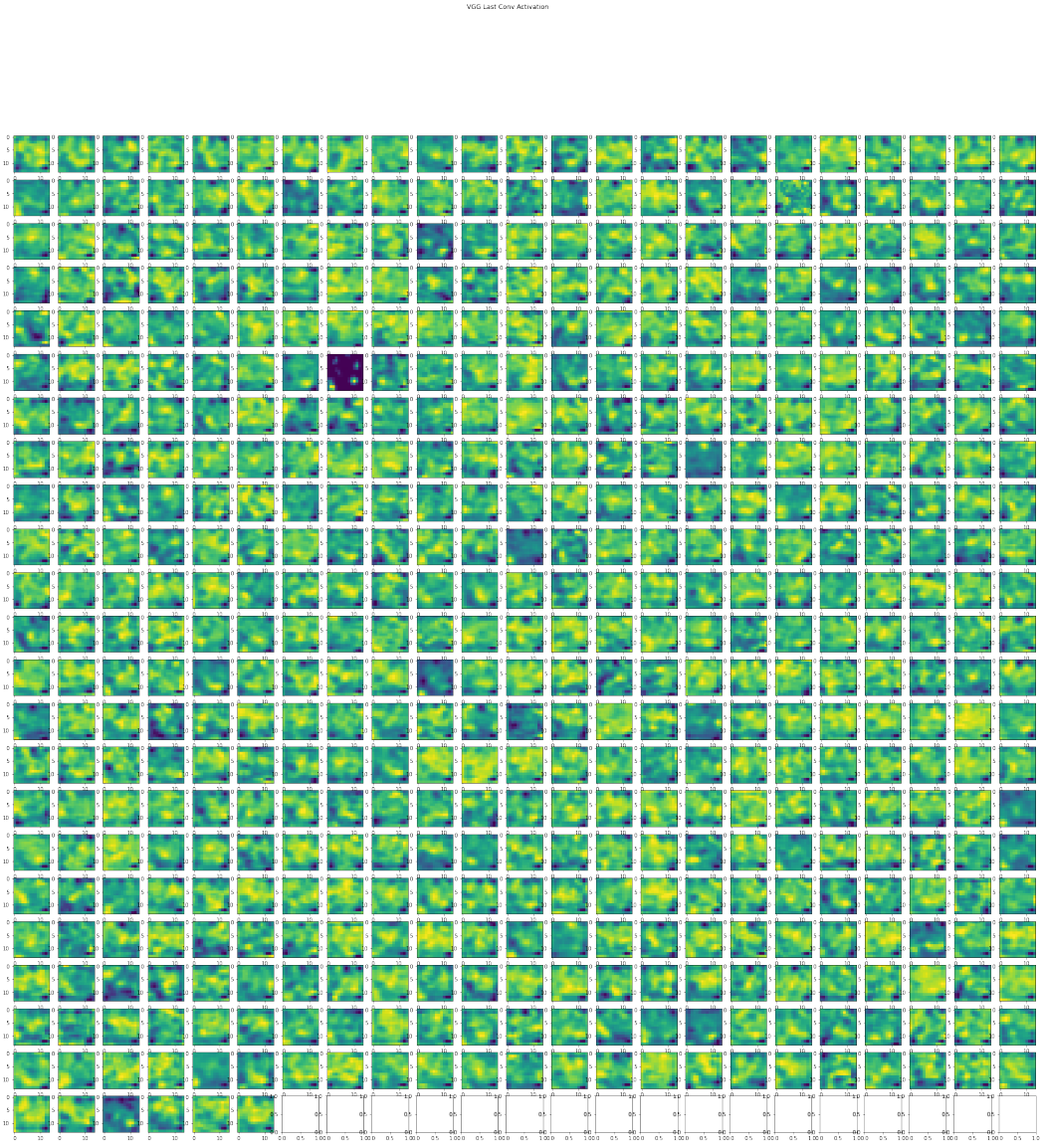

Figure 11: VGG Last Activation Map

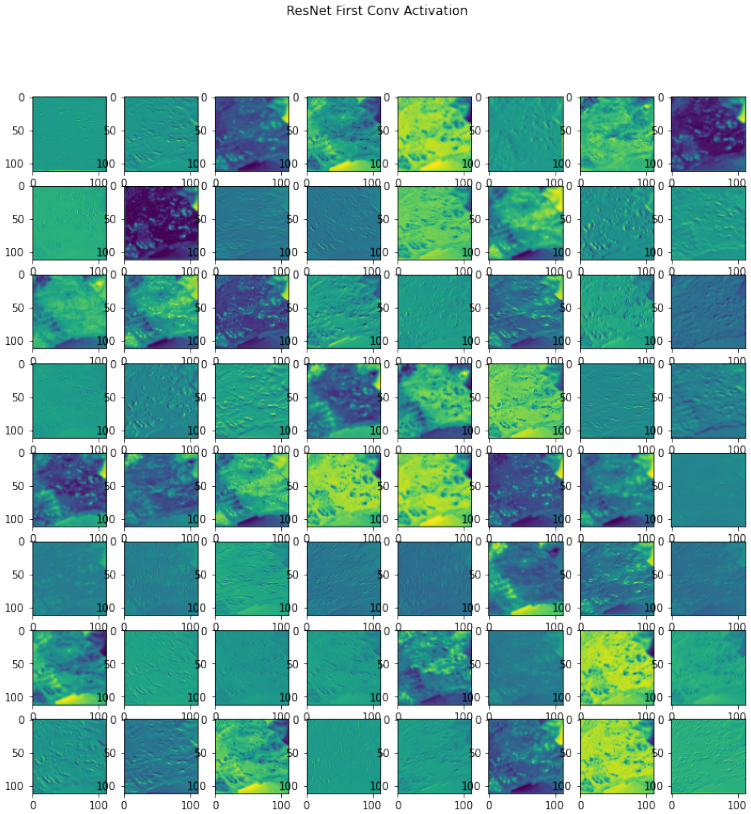

Figure 12: ResNet First Activation Map

5.2.2 VGG Feature Map

The VGG model's activation maps can be found in Figures 9, 10, and 11 for progressively deeper layers.

In the first layer (Fig 9), we see activation patterns similar to the custom layer but with more variance in the activations. We once again see the top right corner with higher activations, particularly among those with otherwise low activations.

In the middle layer (Fig 10), we start to see patterns in the activations. The activations almost look like rivers flowing in various directions with lines all pointing in the same direction. This leads us to believe the middle layer has been trained to look for lines/edges in particular directions and light up when it sees them.

In the final layer (Fig 11), we see lots of activation maps from the many final convolutions. Much like the custom last, it's hard to make out what these are reacting to due to their features of features status. Each map seems to have a few spotches of high activation, along with other splotches of low activation.

5.2.3 ResNet Feature Map

The ResNet model's activation maps can be found in Figures 12, 13, and 14 for progressively deeper layers.

In the first layer (Fig 12), we see much the same story as our custom model's first layer. Most filters have produced activation maps with low variance. The bottom part of the image seemed to respond particularly well to a good number of filters.

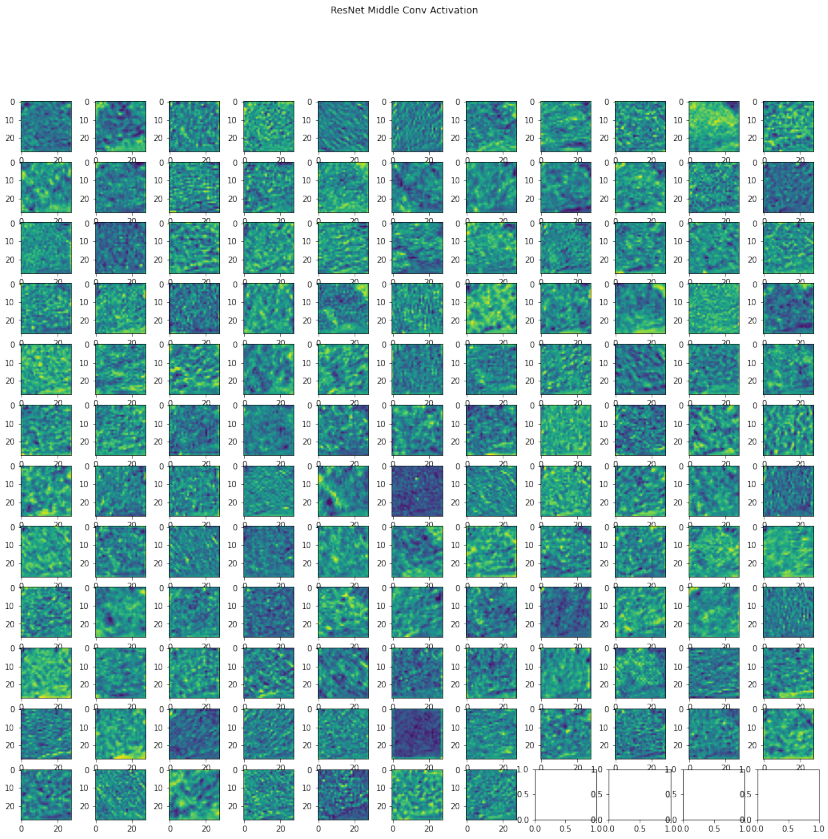

Figure 13: ResNet Middle Activation Map

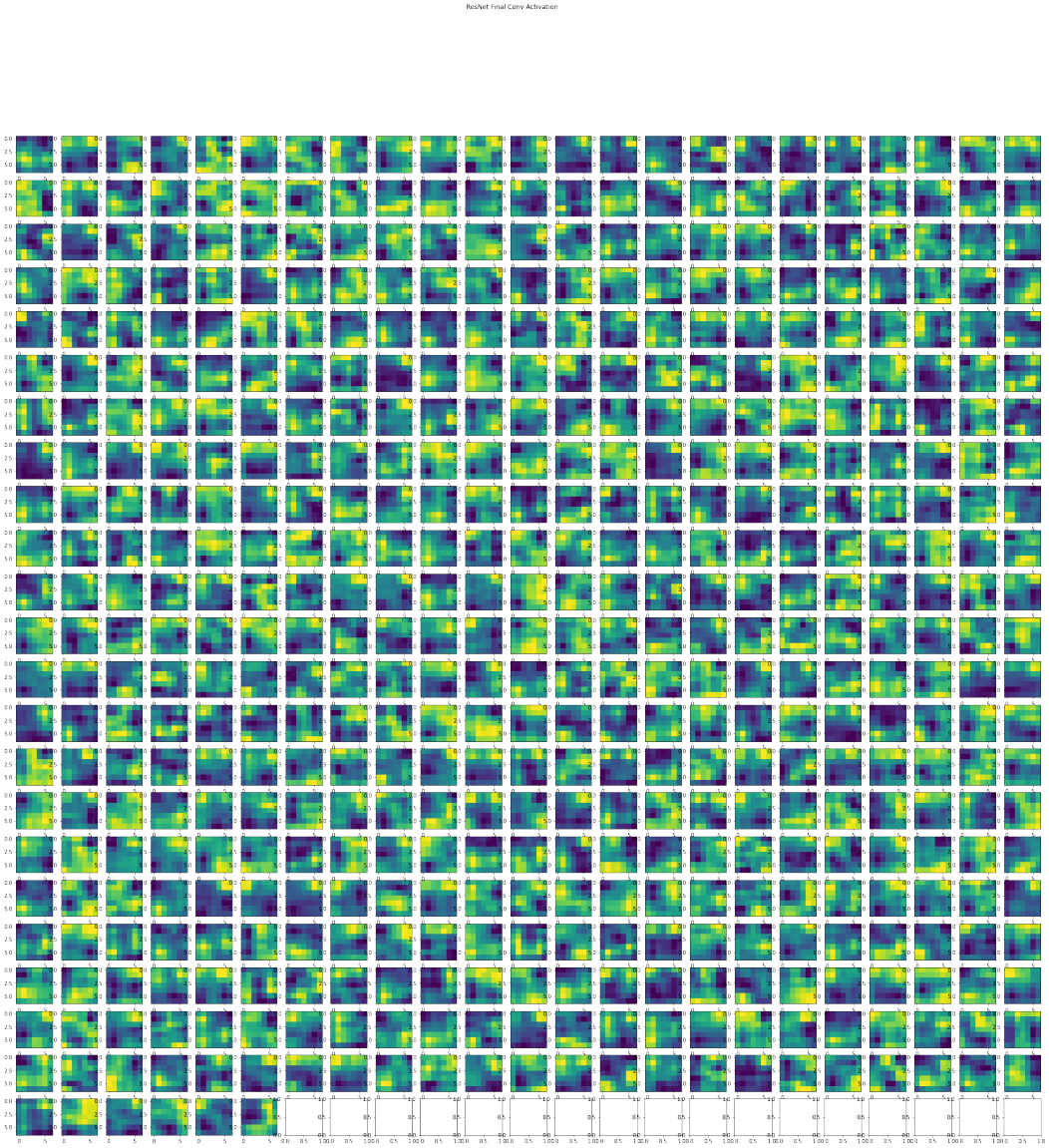

Figure 14: ResNet Last Activation Map

In the middle layer (Fig 13), we see feature maps similar to VGG's middle layer. Many filters have produced activation maps with streaks of high/low activation, perhaps looking for edges in the image.

In the final layer (Fig 14), we see a less extreme version of VGG's last layer. There is much more variance in the activations of each map, with each map activating to a different location.

6 Discussion

6.1 Comparison

To remind ourselves, the baseline model had a test loss of 2.059 and a test accuracy of 37.8%, the custom model had a test loss of 1.550 and a test accuracy of 52.44%, the best resnet model we trained had a test loss of 0.627 and a test accuracy of 80.7%, and finally the best VGG model we trained had a test loss of 0.590 and a test accuracy of 82.96%.

While training these models, we noted that on the DataHub cluster with pin_memory=True and num_workers=4 for our dataloaders, the baseline took the longest at about 57 minutes to train. The custom took less time to train, around 35 minutes to train on the cluster, we think this could be due to the extra max pooling layer that we added compared to the baseline, which reduced the dimensionality and the amount of parameters in the network. The resnet model was by far the fastest, taking around 17 minutes to train on the DataHub cluster. The VGG model actually took around the same time as the baseline model to train.

The model with the highest test accuracy that we trained turned out to be our vgg16_model with selective freezing of the first convolutional layer, data augmentation using the given augmentations (see table), and a learning rate of 0.0001. We think this was probably the best compared to the custom model because it was deeper and had more parameters. In addition, it was also pre-trained already, and it could have certain features that are better suited for detecting things better than how far our custom model got. Comparing it to the ResNet model, the reasons VGG performed better could have been due to the fact that VGG has more parameters ResNet, maybe the learning rate of ResNet was not completely optimal, or maybe freezing different layers of ResNet and VGG could have ended with different results.

Comparing our custom model's performance to the higher performing ResNet and VGG models, we think there are a few areas that we could focus on to improve the custom model in the future. One thing we believe we could do to improve the model is make the model even deeper, as ResNet and VGG's architectures have significantly more layers and thus more parameters to be learned. In addition to this, also try to tune more hyperparameters than learning rate. There is also the momentum gamma that we could tune, the number of epochs, the batch size, as well as the patience we want for early stop in order to try to train a better model.

6.2 Helpful changes for custom

For our custom model, the first change that really made an impact was adding two convolutional layers. Compared to the baseline model, the increase of 2 convolutional layers yielded a performance increase of 2% when it comes to test accuracy, from 37.8% to 39.86%. We think this is due to the fact that an increase in the number of layers meant that the model would have more parameters to learn and more parameters to learn would mean more features are able to be learned.

The second change that made an impact was changing the learning rate from 0.001 to 0.0005. Compared to the baseline model, the change to the smaller learning rate of 0.0005 came with an increase of test accuracy from 37.8% to 40.92%. We think this is due to the fact that the default learning rate was actually too large, meaning the steps taken at each update was a bit too large, missing minima that might lead to better performance. Changing it to a smaller learning rate allowed the model to take smaller steps and find the other minima that might've been missed.

The final individual change that made an impact was adding an extra max pooling layer. Compared to the baseline model, just adding an extra max pooling layer led to a significant increase in test accuracy, from 37.8% to 46.92%. We believe this is due to the fact that max pooling made the model simpler by reducing the number of dimensions and parameters, leading to possible better

generalization and allowed the receptor fields to see more of the input at time through this pooling. In addition, it also made the training time a lot faster due to that reduction of dimension and number of parameters.

Putting all three of these changes together yielded us our best custom model that had a test accuracy of 52.44% and test loss of 1.550.

6.3 Helpful changes for ResNet

When tuning our ResNet model, the first change that made an impact was partial fine-tuning. Compared to the stock model with just the last fully connected layer unfrozen, the fine-tuned model saw an increase from 68.32% test accuracy to 71.22% test accuracy. We believe this improvement was due to the fact that while the model was already pretrained to be very good at detecting features, allowing the model to further fit the model towards the food dataset allowed it to generalize better to Food 101.

The second change that made an impact was changing the learning rate to 0.0001. This change saw a drastic increase in performance, from 68.32% to 81.6% for test accuracy. We believe this improvement was due to the fact that the default learning rate was a little too high, possibly missing some local minima that could help the model be more accurate. However, this came with a tradeoff, which was overfitting the training set. The model hit 99% training accuracy very quickly, so we tried to turn to data augmentation to reduce this effect.

The last change that made an impact was data augmentation on top of a combined change of changing learning rate and partial fine-tuning. We modified the dataloader to transform the dataset using a long list of transforms (see Table 5). While this slightly lowered our test accuracy, our test loss decreased from 1.003 to 0.627 over the stock frozen model. In addition, the overfitting was reduced by a lot, with the model taking 15 epochs before learning started leveling out. We believe this is due to the data augmentations enlarging our dataset and providing different even more training examples to help the model be more generalizable.

6.4 Helpful changes for VGG

When tuning our model for VGG, the first change that made an impact was partial fine-tuning and lowering of learning rate to 0.0001. This saw a dramatic increase of test accuracy from 70.7% to 82.54%. However, just fine-tuning alone on the default parameters didn't yield very good results, and it seemed to be because the learning rate was too large, resulting in very poor performance. Allowing both fine-tuning and learning rate decrease however resulted in a huge accuracy increase and we think it is due to the fact that the learning rate allowed the finetuning to make small enough steps to find the minima. However, this change resulted in overfitting very early on in training and training accuracy reaching 99% very soon and learning not being productive after a few epochs. So, we turned to data augmentation once again to reduce this overfitting and help the model generalize.

Next, the second change we made was adding data augmentation using the given transforms (see Table 5) on top of the first set of changes. This saw a slight increase from the previous change, from 82.54% test accuracy to 82.9%. However, overfitting was reduced dramatically, with learning being productive up until at least epoch 15. We think this is because data augmentations enlarging our dataset and providing different even more training examples to help the model be more generalizable. And in our case, the augmentations really did help the model generalize better and is shown in the test set accuracy increase.

Finally, our final change that was impactful was selective weight freezing of the first convolutional layer on top of data augmentation and lowering learning rate to 0.0001. This resulted in a slight increase in performance from 82.9% to 82.96% in test accuracy. This increase could be due to differences from run to run, but we believe this improvement could be due to the fact that maybe the parameters in the first convolutional layers were already very generalizable and updating them could hurt performance on unseen test data.

7 Authors' contributions

7.1 Jianchang

I added on to Nick's original version of the code by writing in the custom model and making a few changes to engine.py to accurately display the loss according to the given log files. In addition, I attended office hours to verify and get help for our implementation. I tuned the models for custom, resnet, and VGG by experimenting with extra layers, learning rate tuning, dropout, augmentation, and selective weight freezing. As for the report, I wrote the custom model, reset, and VGG parts of the "Models," "Experiments," and the "Discussion" sections.

7.2 Nick

I coded the boilerplate for the model, including data transforms, dataloaders, model training, and loss/accuracy tracking. I also created the baseline model, as well as the VGG/ResNet implementations. Finally, I did the visualizations for the model weights and activation maps. For the report, I wrote the Feature Maps/Weights section.

7.3 Rick

I tested and pruned the code for bugs and performance evaluation. For the report, I was responsible for completing the abstract, introduction, related work, description of the baseline model, and references.

References

[1] Lukas Bossard, Matthieu Guillaumin, and Luc Van Gool. Food-101 – mining discriminative components with random forests. In *European Conference on Computer Vision*, 2014.

[2] Gary Cottrell. Convolutional networks, part ii. 2022.

[3] Gary Cottrell. Introduction to convolutional networks. 2022.

[4] Nathan Inkawhich. Finetuning torchvision models.

[5] Andrej Karpathy. A recipe for training neural networks, 2019.