Detecting Pneumonia in X-rays

Github: https://github.com/yoosufb/CECS456_FinalProject.git

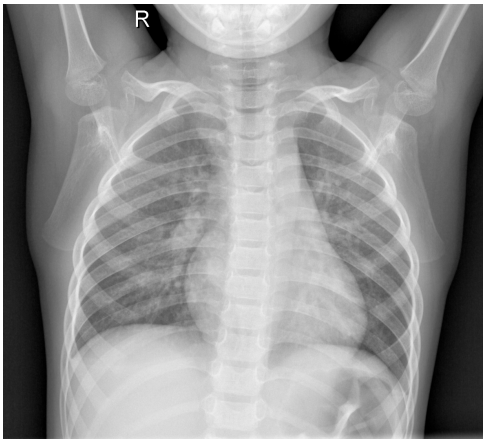
Introduction

With this project, we aim to train a convolutional neural network model to detect pneumonia in x-ray images of a patient's lungs. We train on the Kaggle dataset "Chest X-Ray Images (Pneumonia)". The goal of our project is to determine, given an X-Ray, whether the X-Ray belongs to the 'Normal' or 'Pneumonia' class. Notice we have to determine if an image is a part of one class or another, there is no third class, which means we have a binary classification problem.

Dataset

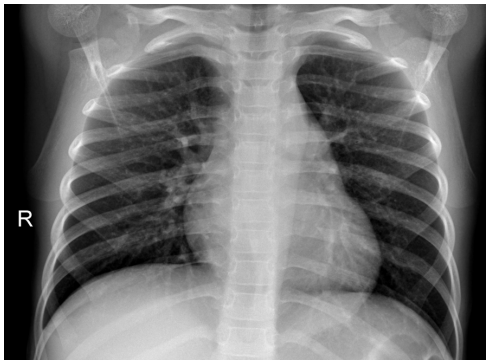
The dataset consists of three folders containing training, validation, and test images. Each folder comprises two classes of images, "normal" and "pneumonia". The images are of varying dimensions and specifications. Some are grayscale but represented in RGB format, others are truly RGB. Our aim is to provide a model that normalizes the details of any x-ray image and provides a reliable method of detecting pneumonia. Additionally, a crucial step in this process of balancing the dataset was the random removal of 2,534 images from the PNEUMONIA class in the training dataset. This balanced the classes in our dataset. Skewing a model's training in favor of the PNEUMONIA class may result in higher rates of false positive classifications.

This dataset's difficulty came from the extreme granularity of the differences between a "normal" image and a "pneumonia" image. In other words, other Kaggle datasets, such as "Animals-10" or "Natural Images" consist of images with vast, visible, and easy-to-identify differences between each class. For example, a picture of an airplane will be decidedly distinct from a picture of a dog, especially so for a machine learning model designed to recognize edges and patterns. In our dataset, the differences between the two classes are less distinct, with only minor differences in the brightness of the interior of the lung indicating pneumonia. For example, refer to Figure 1 below and notice how minor the differences are between each class..

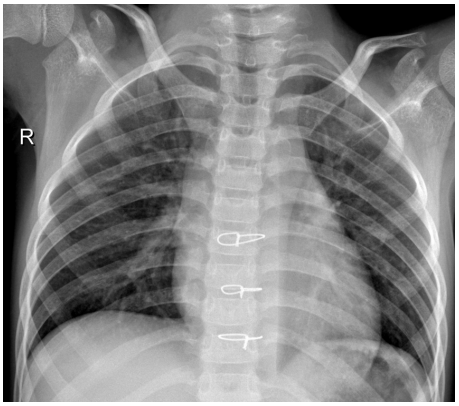


Normal

Pneumonia



Normal



Pneumonia

Figure 1

VGG16 - Tony Samaniego

The initial copy of the VGG16 model yielded poor results after 10 epochs with a training accuracy of 0.5258, a validation accuracy of 0.5, and a test accuracy of 0.6250. As this was just the starting point we began experimenting with changing the number of filters per convolution layer and node count per fully connected layer. After experimenting with several different model configurations, the following configuration resulted in the best performance accuracy.

Layer (type)	Output Shape	Param #
rescaling_1 (Rescaling)	(None, 224, 224, 3)	0
conv2d_5 (Conv2D)	(None, 224, 224, 64)	1792
max_pooling2d_5 (MaxPooling 2D)	(None, 112, 112, 64)	0
conv2d_6 (Conv2D)	(None, 112, 112, 128)	73856
max_pooling2d_6 (MaxPooling 2D)	(None, 56, 56, 128)	0
conv2d_7 (Conv2D)	(None, 56, 56, 256)	295168
max_pooling2d_7 (MaxPooling 2D)	(None, 28, 28, 256)	0
conv2d_8 (Conv2D)	(None, 28, 28, 512)	1180160
max_pooling2d_8 (MaxPooling 2D)	(None, 14, 14, 512)	0
conv2d_9 (Conv2D)	(None, 14, 14, 512)	2359808
max_pooling2d_9 (MaxPooling 2D)	(None, 7, 7, 512)	0
flatten_1 (Flatten)	(None, 25088)	0
dense_2 (Dense)	(None, 128)	3211392
dropout_1 (Dropout)	(None, 128)	0
dense_3 (Dense)	(None, 2)	258

Total params: 7,122,434
Trainable params: 7,122,434
Non-trainable params: 0

For training the model, we decided to do so in 3 stages to resolve the limited number of dataset samples due to the removal of “PNEUMONIA” images to balance the training set. Data augmentation was applied to the training dataset in the following configurations.

- 1st Stage: No data augmentation.
- 2nd Stage: Random flip (horizontal & vertical) and random rotation.

- 3rd Stage: Random flip (horizontal & vertical), random rotation, random brightness, and random contrast.

The training, validation, and testing images were all preprocessed to have an image size of 224x224x3 and rescaled by 1/255 for all 3 states. Each training session consisted of 40 epochs for a total of 120 training epochs.

This strategy resulted in some surprising results. During the 3rd stage, the validation accuracy and validation loss began to trend in a negative direction. The validation accuracy had a downward slope of 0.0042 per epoch with the validation loss increasing with a slope of 0.156 per epoch as shown in Figures 2 and 3. The 2nd stage resulted in the highest validation accuracy and the lowest validation loss. Despite the decrease in validation performance, each training session resulted in an increased test accuracy and a decreased test loss. Table 1 shows the testing results after each training session.

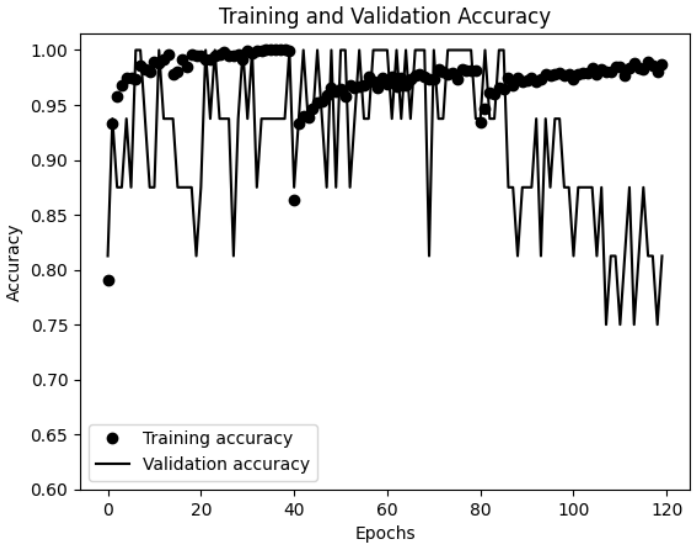


Figure 2

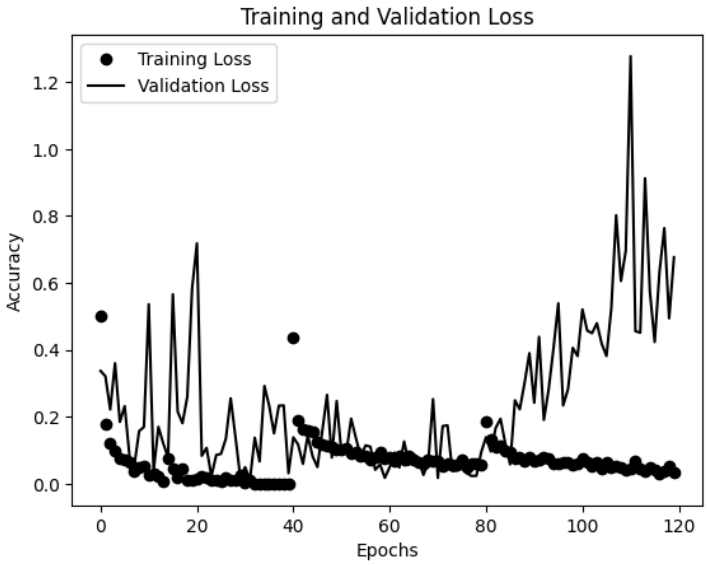


Figure 3

Training 1	Training 1	Training 1
• Accuracy: 80.1% • Loss: 4.505 • Precision: 76.0% • Recall : 99.7% • F1: 86.3%	• Accuracy: 83.7% • Loss: 0.856 • Precision: 80.3% • Recall : 97.9% • F1: 88.2%	• Accuracy: 89.6% • Loss: 0.473 • Precision: 88.6% • Recall : 95.6% • F1: 92.0%

Table 1

Table 2 shows the confusion matrix results for the testing dataset predictions after the 3rd training session.

		Predicted	
		1	0
Actual	1	373	17
	0	48	186

Table 2

Alexnet - Malhar Pandya

I used Alexnet 2012 as the basis for my machine learning model. Alexnet was introduced as a Convolutional Neural Network that was able to classify an images into 1000 different classes. Since Alexnet was one of the more basic CNN models I decided it would serve as the perfect basis to understand exactly how CNN's work when it comes to classifying images. The depth of Alexnet compared to other models also gives me a chance to understand how much of an effect methods such as preprocessing and regularization has on the effectiveness of a model. My Alexnet model is based on the image below(**Figure 4**), the change I made is changing the softmax from classifying images into 1000 classes to instead classifying images between 2 classes, as we only have 'normal' and 'pneumonia' class.

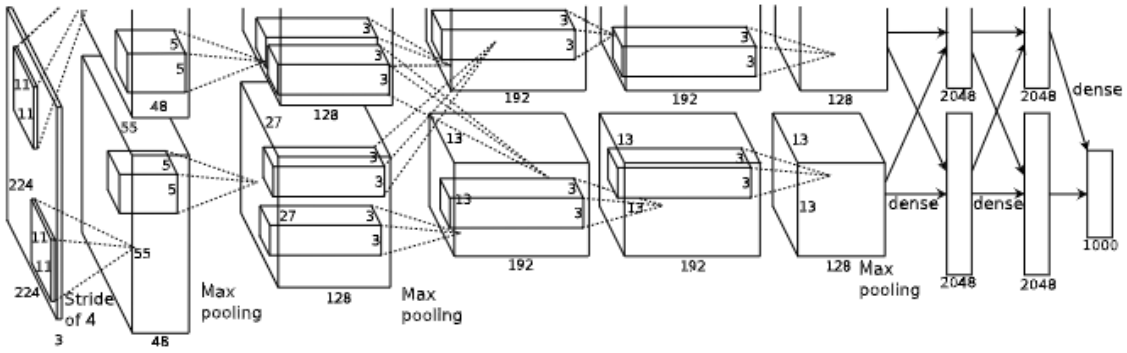


Figure 4

First Run:

The first time I ran my model, it was without any preprocessing/ data augmentation. I ran my models for 20 epochs as I noticed that after 20 epochs the training and validation accuracy did not change a lot. A problem I encountered with my first run was that the model had a very low test accuracy(around 70%) despite having a decent training accuracy(In the 90%'s). I understood this as my weights being too specific to the training set and decided to introduce pre-processing to take care of the overfitting. The model gave me a test accuracy of roughly 73% in my first run(**Table 4**).

Model:

Layer (type)	Output Shape	Param #
conv2d_5 (Conv2D)	(None, 62, 62, 96)	34944
max_pooling2d_3 (MaxPooling 2D)	(None, 30, 30, 96)	0
batch_normalization_2 (BatchNormalization)	(None, 30, 30, 96)	384
conv2d_6 (Conv2D)	(None, 30, 30, 256)	614656
max_pooling2d_4 (MaxPooling 2D)	(None, 14, 14, 256)	0
batch_normalization_3 (BatchNormalization)	(None, 14, 14, 256)	1024
conv2d_7 (Conv2D)	(None, 14, 14, 384)	885120
conv2d_8 (Conv2D)	(None, 14, 14, 384)	1327488
conv2d_9 (Conv2D)	(None, 14, 14, 256)	884992
max_pooling2d_5 (MaxPooling2D)	(None, 6, 6, 256)	0
flatten_1 (Flatten)	(None, 9216)	0
dense_3 (Dense)	(None, 4096)	37752832
dropout_1 (Dropout)	(None, 4096)	0
dense_4 (Dense)	(None, 4096)	16781312
dense_5 (Dense)	(None, 2)	8194

Total params: 58,290,946
Trainable params: 58,290,242
Non-trainable params: 704

		Predicted	
		1	0
Actual	1	377	13
	0	153	81

Table 3

Training 1
• Accuracy: 73% • Precision: 71% • Recall: 96.6% • F1: 81.8%

Table 4

In my first run, I tested the model, without any modifications or without any dropouts. I did this to see what results a bare bones Alexnet model, without any regularization would have. What I found is that the basic Alexnet model does not perform very well. This is understandable as the original model was created to handle a different set of data. Although an accuracy of 73% is not bad, I believe I could do a better job with introducing pre-processing and regularization to take care of my overfitting problem.

Second Run:

I realized I had a case of overfitting, so I decided to add preprocessing into my Methodology. My second time around I decided to rescale the images between values of [0,1]. I did feature scaling in order to make sure some aspects of the image did not dominate the entire model, using feature scaling I was able to make sure the data was normalized.

Another strategy I implemented to combat overfitting is using dropouts. I initially wanted a way to introduce regularization into my Alexnet model so I can counter the over-fitting problem. The Alexnet paper itself makes use of two dropouts in both of its Fully Connected Layers. I made a slight change to my dropouts and changed the probability from a value of 0.5 to 0.3. I did this to make sure I didn't make the regularization strong where it could lead to underfitting instead. I still ran my model for 20 epochs.

Model:

Layer (type)	Output Shape	Param #
conv2d_20 (Conv2D)	(None, 62, 62, 96)	34944
sequential (Sequential)	(None, 256, 256, 96)	0
max_pooling2d_12 (MaxPooling2D)	(None, 127, 127, 96)	0
batch_normalization_8 (BatchNormalization)	(None, 127, 127, 96)	384
conv2d_21 (Conv2D)	(None, 127, 127, 256)	614656
max_pooling2d_13 (MaxPooling2D)	(None, 63, 63, 256)	0
batch_normalization_9 (BatchNormalization)	(None, 63, 63, 256)	1024
conv2d_22 (Conv2D)	(None, 63, 63, 384)	885120
conv2d_23 (Conv2D)	(None, 63, 63, 384)	1327488
conv2d_24 (Conv2D)	(None, 63, 63, 256)	884992
max_pooling2d_14 (MaxPooling2D)	(None, 31, 31, 256)	0
flatten_4 (Flatten)	(None, 246016)	0
dense_12 (Dense)	(None, 4096)	1007685632
dropout_4 (Dropout)	(None, 4096)	0
dense_13 (Dense)	(None, 4096)	16781312
dropout_5 (Dropout)	(None, 4096)	0
dense_14 (Dense)	(None, 2)	8194

Total params: 1,028,223,746
Trainable params: 1,028,223,042
Non-trainable params: 704

		Predicted	
		1	0
Actual	1	379	11
	0	112	122

Table 5

Training 2

- Accuracy: 80%
- Precision: 77%
- Recall: 97.1%
- F1: 85.8%

Table 6

I noticed that normalization and dropout did help the model overfit less, however there was still a considerable gap in the testing accuracy and the training accuracy(**Table 6**). I decided to add more preprocessing and change my regularization. I also decided to save my model from the second run, and use the weights I got from the second model so that I was no longer randomizing the weights when training my model.

As mentioned earlier, I introduced a rescaling of the data to values between[0,1] along with two dropouts after my two fully connected layers. This made sure that my data was standardized and I had a strategy to combat overfitting. The results got better, I saw an increase in accuracy from 74% to 80%. There are also some key takeaways, the Precision and Recall went slightly higher, this suggests that the model improved overall in that it identified more cases that were true positives and true negatives.

Third Run:

For my third run, I decided to reload my model with the weights I got from my second run. My goal here was to use weights that were more specific to the problem and weren't randomly selected. I also decided to change the way I do data augmentation. I decided to continue rescaling the images, however I also added a horizontal/vertical flip along with a random rotation with a factor of 0.2. This added more data to my dataset which can help me with my overfitting problem.

I also changed the way I approached regularization. The original Alexnet model used two dropouts in both of their Fully-Connected layers. I however used only one dropout layer with a probability of 0.5. I did this because when I continued using both of the dropouts, I encountered underfitting, in that both my training accuracy and test accuracy were hovering around 50%. I still ran my model for 20 epochs.

Final Model:

Layer (type)	Output Shape	Param #
conv2d_60 (Conv2D)	(None, 62, 62, 96)	34944
sequential_1 (Sequential)	(None, None, None, 96)	0
max_pooling2d_36 (MaxPooling2D)	(None, 30, 30, 96)	0
batch_normalization_24 (BatchNormalization)	(None, 30, 30, 96)	384
conv2d_61 (Conv2D)	(None, 30, 30, 256)	614656
max_pooling2d_37 (MaxPooling2D)	(None, 14, 14, 256)	0
batch_normalization_25 (BatchNormalization)	(None, 14, 14, 256)	1024
conv2d_62 (Conv2D)	(None, 14, 14, 384)	885120
conv2d_63 (Conv2D)	(None, 14, 14, 384)	1327488
conv2d_64 (Conv2D)	(None, 14, 14, 256)	884992
max_pooling2d_38 (MaxPooling2D)	(None, 6, 6, 256)	0
flatten_12 (Flatten)	(None, 9216)	0
dense_28 (Dense)	(None, 4096)	37752832
dropout_9 (Dropout)	(None, 4096)	0
dense_29 (Dense)	(None, 4096)	16781312
dense_30 (Dense)	(None, 2)	8194

Total params: 58,290,946
Trainable params: 58,290,242
Non-trainable params: 704

Training 3
• Accuracy:86.86 % • Precision: 85.6% • Recall 94.8 % • F1:89.9 %

Table 7

		Predicted	
		1	0
Actual	1	370	20
	0	62	172

Table 8

:

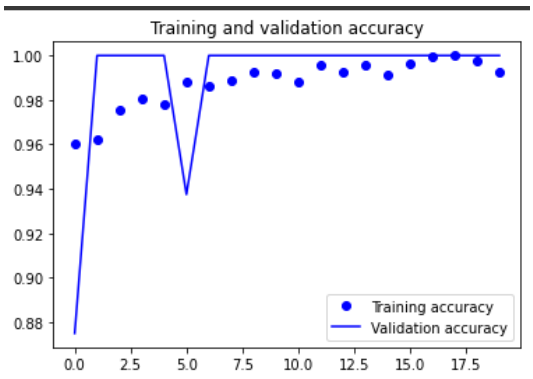


Figure 5

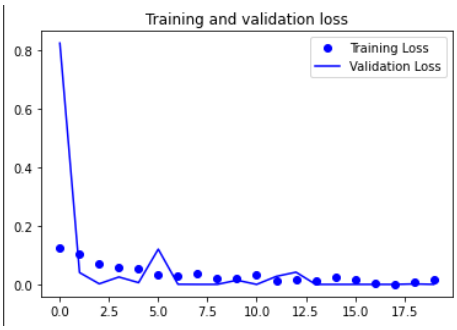


Figure 6

As I expected the accuracy of the model went up substantially, from 80% to 86%. What I noticed is that the recall went down from 97% to 95%, but the precision went up from 77%to 85%. This makes sense as we know these attributes generally have an inverse relationship.

. ZFnet - Yoosuf Batliwala

ZFNet introduced several key concepts. One was the use of stacking multiple convolutional layers together, which allowed for more complex feature learning. Another was the use of local response normalization which helped to reduce overfitting and improve the generalization of the model. In this experiment, we use the unmodified ZFnet architecture described in *Visualizing and Understanding Convolutional Networks* by Zeiler et al. It is shown below in Figure 7.

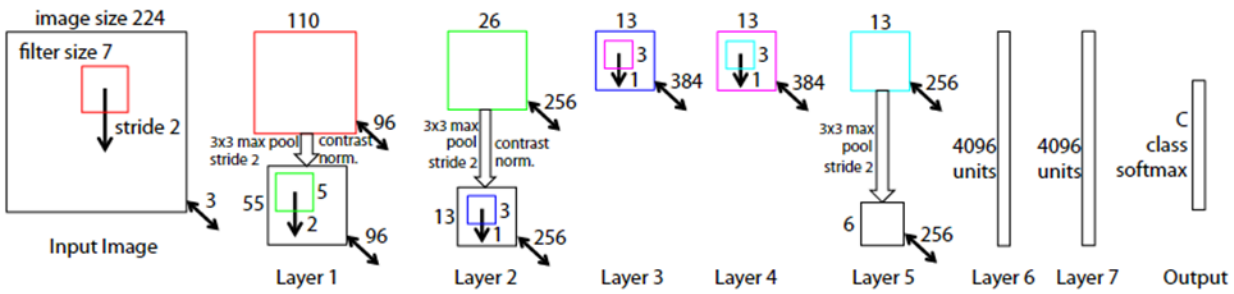
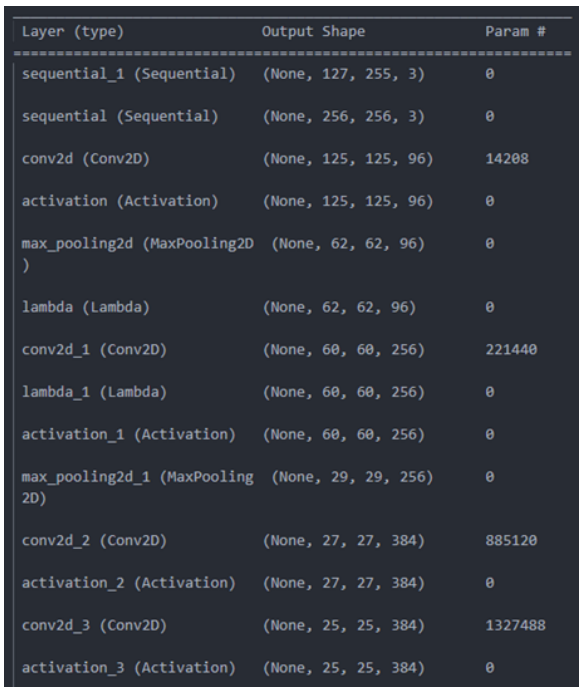
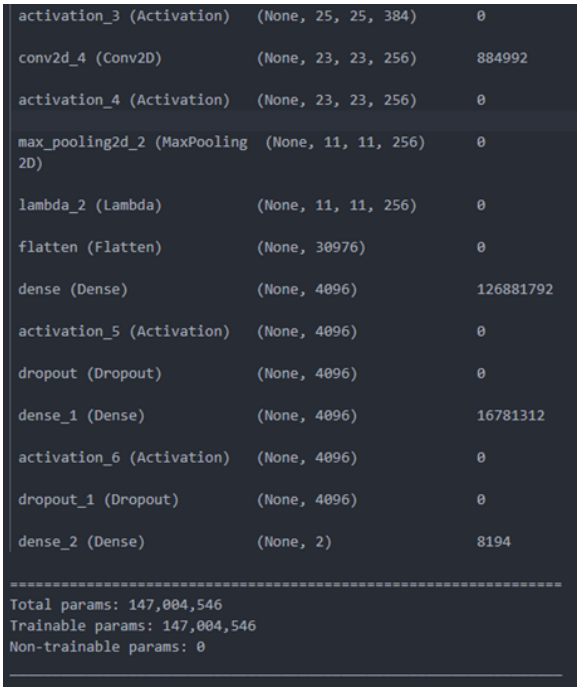


Figure 7



Layer (type)	Output Shape	Param #
sequential_1 (Sequential)	(None, 127, 255, 3)	0
sequential (Sequential)	(None, 256, 256, 3)	0
conv2d (Conv2D)	(None, 125, 125, 96)	14208
activation (Activation)	(None, 125, 125, 96)	0
max_pooling2d (MaxPooling2D)	(None, 62, 62, 96)	0
lambda (Lambda)	(None, 62, 62, 96)	0
conv2d_1 (Conv2D)	(None, 60, 60, 256)	221440
lambda_1 (Lambda)	(None, 60, 60, 256)	0
activation_1 (Activation)	(None, 60, 60, 256)	0
max_pooling2d_1 (MaxPooling2D)	(None, 29, 29, 256)	0
conv2d_2 (Conv2D)	(None, 27, 27, 384)	885120
activation_2 (Activation)	(None, 27, 27, 384)	0
conv2d_3 (Conv2D)	(None, 25, 25, 384)	1327488
activation_3 (Activation)	(None, 25, 25, 384)	0



activation_3 (Activation)	(None, 25, 25, 384)	0
conv2d_4 (Conv2D)	(None, 23, 23, 256)	884992
activation_4 (Activation)	(None, 23, 23, 256)	0
max_pooling2d_2 (MaxPooling2D)	(None, 11, 11, 256)	0
lambda_2 (Lambda)	(None, 11, 11, 256)	0
flatten (Flatten)	(None, 30976)	0
dense (Dense)	(None, 4096)	126881792
activation_5 (Activation)	(None, 4096)	0
dropout (Dropout)	(None, 4096)	0
dense_1 (Dense)	(None, 4096)	16781312
activation_6 (Activation)	(None, 4096)	0
dropout_1 (Dropout)	(None, 4096)	0
dense_2 (Dense)	(None, 2)	8194

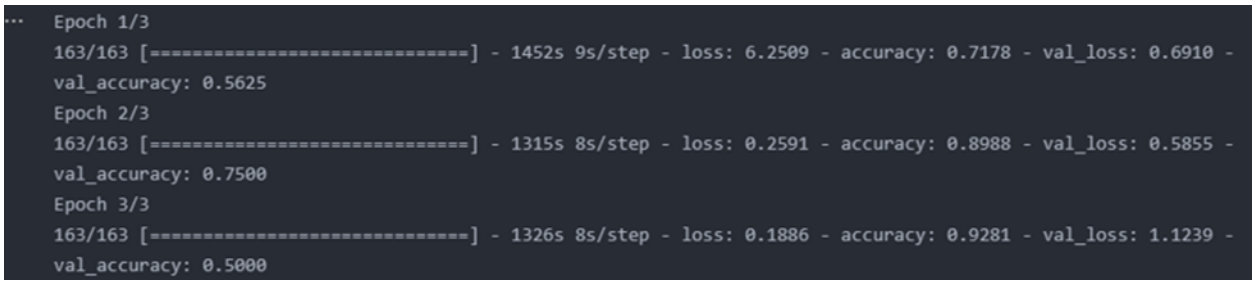
Total params: 147,004,546
Trainable params: 147,004,546
Non-trainable params: 0

Figure 8

Figure 9

First Run

During the first test run of this model, no data augmentation or preprocessing was applied. Instead, the images were simply fed into the model with a resizing layer that resized them to 224x224 as they were all disparate resolutions and dimensions before being imported. This run resulted in an accuracy of 76%, with a 0.90 loss. These settings were only trained for three epochs due to diminishing returns when training any further. Additionally, one of the limitations we ran into was that TensorFlow is not compatible with the AMD GPU on my machine, which meant that the model was training using limited CPU power. As such, it took around 22 minutes per epoch to train.



```
Epoch 1/3
163/163 [==============================] - 1452s 9s/step - loss: 6.2509 - accuracy: 0.7178 - val_loss: 0.6910 -
val_accuracy: 0.5625
Epoch 2/3
163/163 [==============================] - 1315s 8s/step - loss: 0.2591 - accuracy: 0.8988 - val_loss: 0.5855 -
val_accuracy: 0.7500
Epoch 3/3
163/163 [==============================] - 1326s 8s/step - loss: 0.1886 - accuracy: 0.9281 - val_loss: 1.1239 -
val_accuracy: 0.5000
```

Figure 10

```
score = model.evaluate(test_ds_norm)
✓ 33.1s
20/20 [==============================] - 33s 2s/step - loss: 0.9028 - accuracy: 0.7612
```

Figure 11

Final Run

After reviewing the arguably poor results from the first run, we decided that more methods of data augmentation and preprocessing were necessary in order to remedy the obvious underfitting that was occurring. Here, we deviate slightly from Zeiler et al.'s preprocessing methods: we resize the image to 256x256 instead of 224x224. This allows us to have more detail, giving each convolutional layer more data to learn on. Additionally, we rescale the image from 1.0 to 127.5 because this allows the image data to be in the range [-1, 1] instead of the smaller [0, 1]. For the issue of data augmentation, we introduce random flips, random rotations, random contrast, and randomly crop half of the image. Both the preprocessing and augmentation are done with Keras layers as can be seen below.

```
resize_rescale = tf.keras.Sequential([
    tf.keras.layers.experimental.preprocessing.Resizing(256, 256),
    tf.keras.layers.experimental.preprocessing.Rescaling(1.0/127.5)
])
data_aug = tf.keras.Sequential([
      tf.keras.layers.RandomFlip("horizontal_and_vertical"),
      tf.keras.layers.RandomRotation(0.2),
      tf.keras.layers.RandomContrast(0.5),
      tf.keras.layers.RandomCrop(127,255)
])
```

Figure 12

Then, these layers are added to the ZFnet model we constructed and trained over 100 epochs. We found that the learning rate had a surprisingly sizeable impact on the results. Larger learning rates resulted in more erratic behavior from the loss value, as can be seen in the results of the first run. The learning rate we chose was 0.0001 as it resulted in a gradual decrease of the loss value over time, which is the deciding factor for an optimal learning rate. The model was trained on Google Colab Pro, running for approximately 35 minutes before training was completed. The model was then evaluated on the test dataset. The training accuracy and loss were 96.22% and 0.1023, respectively. The test accuracy and loss were 90.71% and 0.3442, respectively.

When a model performs well on the training dataset but has a poor generalization to other datasets, it implies that the model is overfitting, and steps must be taken to mitigate that. However, the relatively small difference between the training and test accuracy indicates that

there is no overfitting occurring in this model. When a model performs poorly on both the training dataset and other datasets, it implies that the model is underfitting. However, a training and test accuracy of 96.22% and 90.71%, respectively, is quite respectable and indicates no underfitting.

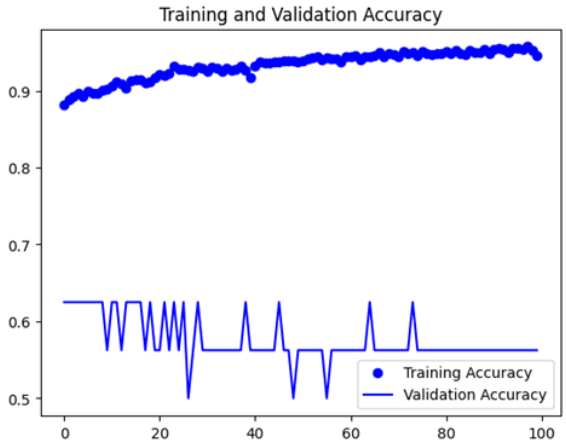


Figure 13

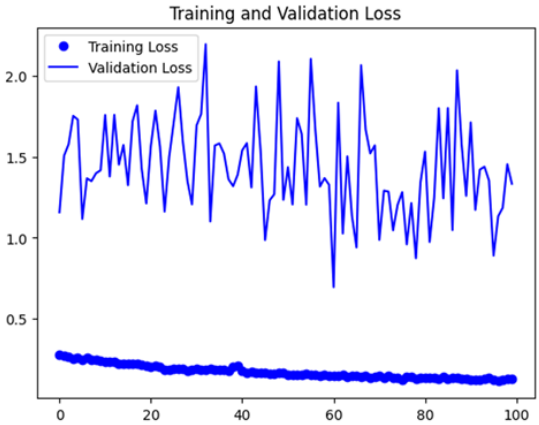


Figure 14

Confusion Matrix:

	1	0
1	387	3
0	55	179

Table 9

MiscData:

- Accuracy on test data: 90.71%
- Accuracy on training data: 96.22%
- Precision: 88%
- Recall: 99%
- F1: 93%

One issue this model had was the seemingly stubborn validation accuracy that oscillated between approximately 50% and 60% throughout the training period. Even after augmenting and balancing the training dataset, the validation accuracy remained low. We conclude that this is likely due to the incredibly small relative size of the validation dataset as compared to the training and test datasets (8 pictures per class, as opposed to 1,341 and ~300 pictures per class, respectively). This is further cemented by the lack of overfitting and underfitting when the training and test datasets are compared. However, this does not explain why the VGG clone created by Tony did not suffer from the same low validation accuracy. Even though the data augmentation was slightly different between this model and the VGG clone, that could not explain the massive difference in validation accuracy. We present this aberration as a question for further research with this model.

Analysis

In this experiment, we created three different models each with different pre-processing and augmentation methodologies. It is difficult to compare the results, keeping in mind all the differences. However, after seeing the results of each of the models, a few things can be concluded with confidence: our preprocessing (resizing, rescaling, augmentation) improved all

of our results tremendously. Additionally, ZFnet achieved the highest overall accuracy out of the three, but all three were fairly close.

Our results may be compared to the ILSVRC results from each of the years that our models competed, however, a few abnormalities emerge. The first is that the VGG model was created after the Alexnet and ZFnet models, and designed as an improvement to them. It is obvious then, that the VGG model should perform better than both Alexnet and ZFnet, however, this is not the case. Instead, our VGG clone achieves an accuracy of 89.6%, with Alexnet and ZFnet achieving 86.54% and 90.71%, respectively. We posit that the reason our VGG clone results in lower accuracy than the ZFnet model is that the VGG clone was trained on unaugmented and unprocessed datasets for the first 40 epochs, dragging down its accuracy somewhat. Additionally, as described by Tony in the VGG section, the design of our VGG clone is slightly dissimilar to the original VGG model proposed by Simonyan et al., potentially pointing to the reason behind the lower accuracy.

The results of our Alexnet and ZFnet models followed the results of the ILSVRC fairly closely. For the 2013 competition, ZFnet was explicitly designed as an improvement to the previous year's winner: Alexnet. It accomplished this with various differences such as more convolutional layers, smaller filter sizes, and lack of dropout regularization. These differences resulted in ZFnet winning the 2013 competition, beating last year's winner Alexnet. In our experiment, we see a similar situation where our Alexnet model achieves 86.54% accuracy and our ZFnet model achieves a much higher 90.71% accuracy. Of course, these comparisons come with the caveat that the augmentation and pre-processing methods were different in each model, making comparisons between them tenuous at best. However, for the purposes of ranking them in the context of our experiment, the comparisons are helpful.

Conclusion

After testing out different models using different data-preprocessing techniques, we made a couple of deductions. The model a person picks for any given problem matters-that is to say that a model like Alexnet was outperformed by VGGnet, however this is to be expected, as VGGnet is supposed to be an improvement compared to Alexnet.

Another thing we deduced is that the methodology of how one does data preprocessing is also very crucial. You can have a model like Alexnet and still make it competitive with VGGnet by introducing the correct data-preprocessing methods.

Contributions

Tony	VGGnet, Dataset
Malhar	Alexnet, Introduction, Conclusion
Yoosuf	Zfnet, Dataset, Analysis

References

Simonyan, Karen and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition." CoRR abs/1409.1556 (2014): n. pag.

Krizhevsky, Alex & Sutskever, Ilya & Hinton, Geoffrey. (2012). ImageNet Classification with Deep Convolutional Neural Networks. Neural Information Processing Systems. 25. 10.1145/3065386.

Zeiler, Matthew D. and Rob Fergus. “Visualizing and Understanding Convolutional Networks.” European Conference on Computer Vision (2013).