Chest X-Ray Images (Pneumonia) Classification

Student ID: | CSCU9M6 | April 9 2024

Introduction

Bacterial pneumonia exhibits focal lobar consolidation, which typically appears as an opaque area in an X-Ray image. In this assignment, the images are classified into two categories by three different neural network architectures. A manual heuristic approach to hyperparameter tuning is employed, and a final architecture design is chosen for the optimal accuracy and loss when considering both validation and testing sets. Overfitting is also kept in mind through comparisons of test and validation metrics.

Preprocessing

Google Colab was used for the whole of the project, which implies that training took an absurdly long time, but more importantly that the dataset had to be stored on google drive and accessed through the drive Python library. To import the dataset, I had the choice between using ImageDataGenerator or image_dataset_from_directory from tf.keras.preprocessing. I chose ImageDataGenerator because the other option was not suitable for datasets that had already been separated into training and test sets. It also has the benefit of performing slight augmentations of the data for generalizability, which is its primary use case.

I chose not to use image augmentation for the first few iterations, but then decided to employ it, which greatly improved my results. This augmented the data for each epoch, diversifying the training data by the various possible modifications; horizontal flipping, sheering and zooming by up to 20%, as well as rotation and dimension shifting. The brightness range was also given some variability because the images are grayscale and brightness changes may be able to aid in detecting opacity, which is the most clear indication of pneumonia in the problem area. the fill_mode for the ImageDataGenerator was set to 'nearest', which uses the nearest neighbor algorithm to fill in blank areas caused by image alteration, which causes the images to be repaired such that they look relatively unedited.

The dataset came with training, test, and validation sets, each with their own two categories of "NORMAL" and "PNEUMONIA". The test and validation sets were left unaltered because they ought to be left in their states in which they were intentionally labeled. In all training sets, I rescaled the images to 1./255, which normalizes the data between 0-1, so that less processing is needed. The images were reshaped to a dimension of 200x200, except for the iterations where ImageNet weights were finetuned on the dataset, in which case a dimension of 224x224 was required.

```
data_dir = "/content/drive/MyDrive/Colab Notebooks/chest_xray"  # small
dataset
#data_dir = "/content/drive/MyDrive/Colab Notebooks/large_chest_xray"
```

```
data = ImageDataGenerator(rescale = 1./255, # simplifies data by normalizing
pixel ranges
                          horizontal_flip=True, # vertical flips make no
sense in this context
                          shear_range = 0.2,
                          zoom_range = 0.2, # Zooming in too much might not
allow the key area to appear for pneumonia
                          width_shift_range=0.2, # Due to the consistency of
the dataset, this should vary little
                          height_shift_range=0.2,
                          brightness_range=(0.8, 1.2), # 20% both ways
                          rotation_range=20, # rotation between -n and n
degrees
                          fill_mode='nearest') # any blank space filled by
interpolation

test_data = ImageDataGenerator(rescale = 1./255)

training_dataset = data.flow_from_directory(data_dir+"/train", target_size =
(img_height, img_width), class_mode="categorical")
validation_dataset = test_data.flow_from_directory(data_dir+"/val",
target_size = (img_height, img_width), class_mode="categorical")
testing_dataset = test_data.flow_from_directory(data_dir+"/test",
target_size = (img_height, img_width), class_mode="categorical")
```

Architectures

I chose three established architectures for their ubiquity and historical significance. AlexNet is rather archaic, but its simplicity turned out to be well-suited for the simple problem at hand, and given its training speed, was used for most generalizable experimentation. VGGNet was chosen to provide a more powerful contrast to AlexNet. Besides training these models with only random starting weights, I also utilized Keras' pre-trained models to see if fine-tuning models trained on larger datasets and more complex problems could yield better results. For this task, I used VGGNet and EfficientNetV2, the latter of which was chosen to provide a simple and faster alternative to VGGNet in order to identify possible overfitting or other training issues.

Hyperparameters were tuned in stages. Nine different stages of changes were ran across the three models, with some architectures left unchanged because a given model suggested the proposed change to be a waste of computing resources. Strict empirical observation methodology was not used because training takes a significant amount of time. Instead, tuning was primarily based upon educated guesses, or more so heuristics.

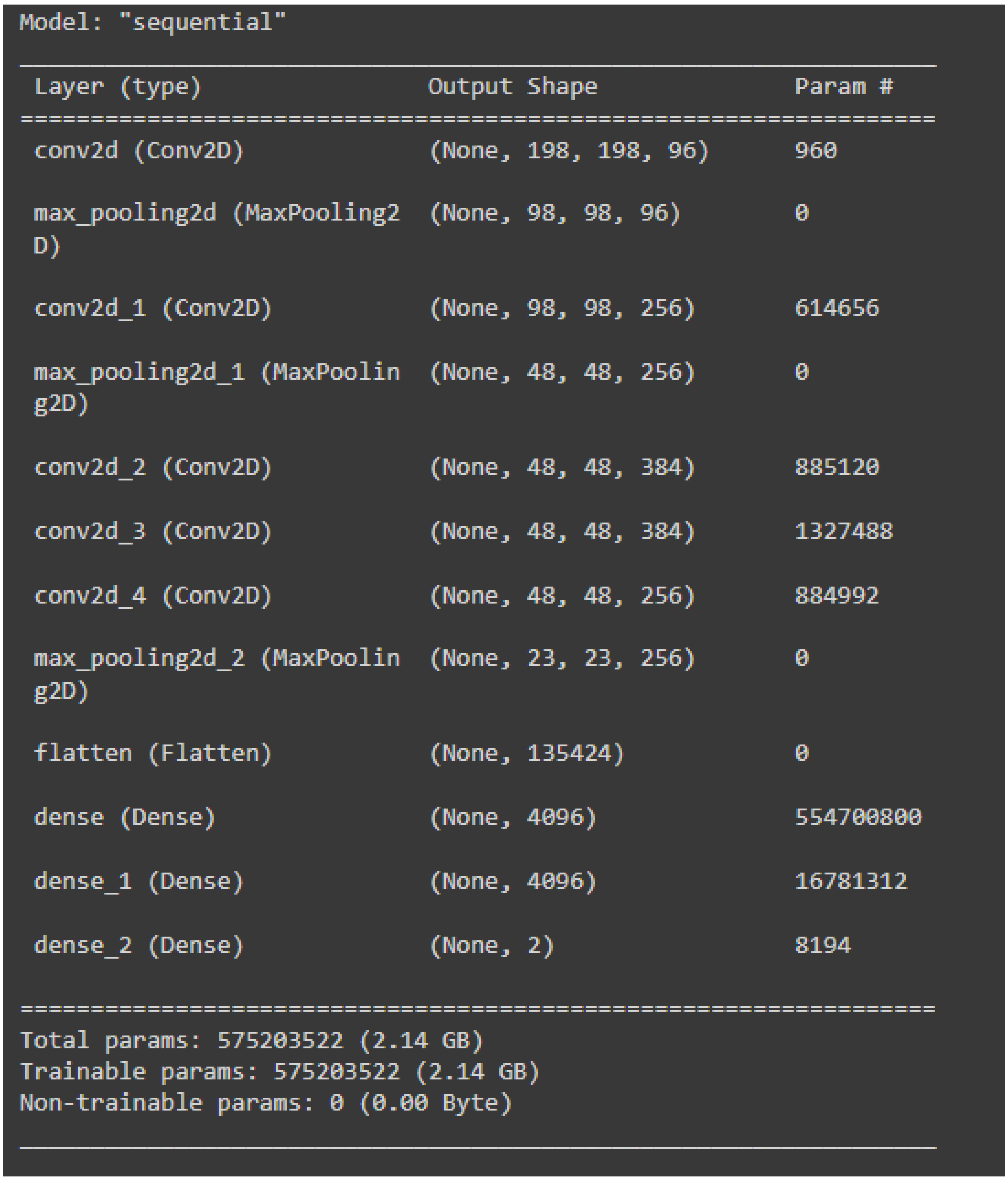
The Categorical Cross-Entropy loss function was used in all stages, since it is more than suitable for the problem at hand. Other functions such as the Quadratic Cost function suffers from learning slow-down, while the Cross-Entropy function does not slow down when a neuron has a very poor loss.

I kept the stride as the default in every architecture because I didn't see much value in losing spatial information in exchange for down-sampling, given the simplicity of the problem and size of the dataset. I similarly kept the filter sizes at their generally small sizes for the same reason; small images with a small area for detection of the opacity and edges of the problem area.

In all architectures dense layers follow the convolutional layers, because the convolutions identify features and the dense layers perform varying levels of classification from those features; complete connectivity is necessary for this so that the features can be compared appropriately.

AlexNet

The original publication of AlexNet used two GPUs, which trained parts of a very large dataset separately and then converged. However, we use a much smaller dataset and a single CPU (at times TPU) for training. Thus only the architecture itself takes any significance upon itself. However, the size of the filters and the stride are smaller, which makes sense for the given dataset and the size and channel size of our images.



```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 198, 198, 96)      960

 max_pooling2d (MaxPooling2  (None, 98, 98, 96)        0
 D)

 conv2d_1 (Conv2D)           (None, 98, 98, 256)       614656

 max_pooling2d_1 (MaxPoolin  (None, 48, 48, 256)       0
 g2D)

 conv2d_2 (Conv2D)           (None, 48, 48, 384)       885120

 conv2d_3 (Conv2D)           (None, 48, 48, 384)       1327488

 conv2d_4 (Conv2D)           (None, 48, 48, 256)       884992

 max_pooling2d_2 (MaxPoolin  (None, 23, 23, 256)       0
 g2D)

 flatten (Flatten)           (None, 135424)            0

 dense (Dense)               (None, 4096)              554700800

 dense_1 (Dense)             (None, 4096)              16781312

 dense_2 (Dense)             (None, 2)                 8194

=================================================================
Total params: 575203522 (2.14 GB)
Trainable params: 575203522 (2.14 GB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

VGGNet

A very large network for complex problems that require large datasets. The most notable aspect of this architecture, besides the depth, is the consistently small stride size which provides lots of spatial information to the rest of the network. This architecture aims for feature-completeness over training speed. The pooling layers are then able to highlight the most salient information among that which is provided by the post-filter convolutional layers.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 zero_padding2d (ZeroPaddin  (None, 204, 204, 1)       0
 g2D)

 conv2d (Conv2D)             (None, 204, 204, 64)      640

 conv2d_1 (Conv2D)           (None, 204, 204, 64)      36928

 max_pooling2d (MaxPooling2  (None, 102, 102, 64)      0
 D)

 conv2d_2 (Conv2D)           (None, 102, 102, 128)     73856

 conv2d_3 (Conv2D)           (None, 102, 102, 128)     147584

 max_pooling2d_1 (MaxPoolin  (None, 51, 51, 128)       0
 g2D)

 conv2d_4 (Conv2D)           (None, 51, 51, 256)       295168

 conv2d_5 (Conv2D)           (None, 51, 51, 256)       590080

 conv2d_6 (Conv2D)           (None, 51, 51, 256)       590080

 max_pooling2d_2 (MaxPoolin  (None, 25, 25, 256)       0
 g2D)

 flatten (Flatten)           (None, 160000)            0

 dense (Dense)               (None, 4096)              655364096

 dense_1 (Dense)             (None, 128)               524416

 dense_2 (Dense)             (None, 2)                 258

=================================================================
Total params: 657623106 (2.45 GB)
Trainable params: 657623106 (2.45 GB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

EfficientNetV2

A relatively new architecture (2021) intended to provide exceptional training speed by utilizing information available during training to augment the training process. It can adjust regularization and image size adaptively.

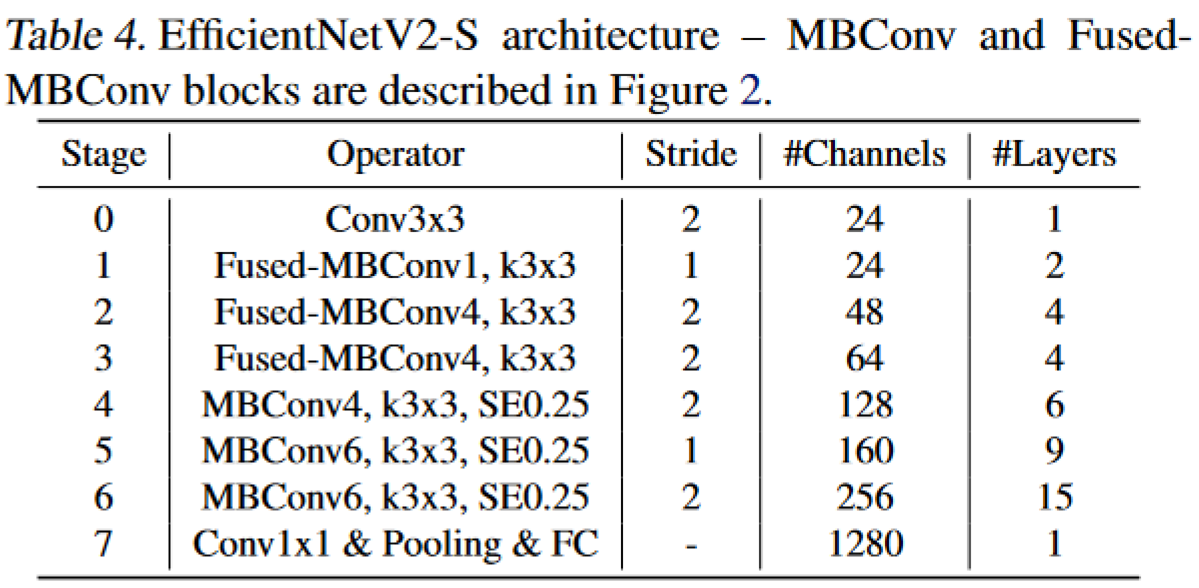


Table 4. EfficientNetV2-S architecture – MBConv and Fused-MBConv blocks are described in Figure 2.

Stage	Operator	Stride	#Channels	#Layers
0	Conv3x3	2	24	1
1	Fused-MBConv1, k3x3	1	24	2
2	Fused-MBConv4, k3x3	2	48	4
3	Fused-MBConv4, k3x3	2	64	4
4	MBConv4, k3x3, SE0.25	2	128	6
5	MBConv6, k3x3, SE0.25	1	160	9
6	MBConv6, k3x3, SE0.25	2	256	15
7	Conv1x1 & Pooling & FC	-	1280	1

Architecture table from the EffNet paper. The MB and Fused-MB convolutional layers are designed in such a way that they minimize the size of parameters.

Training and Hyperparameter Tuning

During training, callbacks (objects that perform different acts at during different stages of training - see Keras documentation) were used to make adjustments to the training process and save time and resources, when the number of epochs exceeded the given patience level. This reduces the learning rate on an observed plateau in an attempt to overcome learning stagnation.

```
callbacks = None
if CNNmodel=="AlexNet":
    callbacks = [EarlyStopping(patience=10, verbose=1),
      ReduceLROnPlateau(factor=0.1, patience=5, min_lr=0.00001, verbose=1),
      ModelCheckpoint('model-AlexNet.h5', verbose=1, save_best_only=True,
save_weights_only=True),
      TensorBoard(log_dir='./logs')]
elif CNNmodel=="VGGNet":
    callbacks = [EarlyStopping(patience=10, verbose=1),
      ReduceLROnPlateau(factor=0.1, patience=5, min_lr=0.00001, verbose=1),
      ModelCheckpoint('model-VGGNet.h5', verbose=1, save_best_only=True,
save_weights_only=True),
      TensorBoard(log_dir='./logs')]
```

And the models were all fit using ImageDataGenerator with the model's fit() method.

```
trained_model = model.fit(
    training_dataset,
    epochs=num_epochs,
    validation_data=validation_dataset,
    callbacks=callbacks,
    batch_size=batch_size
)
```

Stages

These are recorded stages, but other augmentations to the architectures and hyperparameters were utilized. Most importantly among them is that binary classification settings worked poorly or otherwise had no effect compared to categorical classification settings.

Stage 1: Standard configurations. Poor performance on our chosen metrics, but evidently within the ballpark.

Stage 2: Batch changes. Same as before; larger batch sizes do not seem to change much.

Stage 3: Adam optimizer instead of SGD. There is a 20% increase in accuracy both ways, but this could only be because SGD is sensitive to parameter tuning, though this is an assumption.

Stage 4: Here we address an obvious issue of overfitting as seen in the training accuracy w/ validation vs the test accuracy. We regularize the dense layers of each network. Accuracy is considerably worse. Mid-training VGGNet went a similar way as AlexNet so I stopped the process early.

Stage 5: We introduce image augmentation. Loss dramatically decreased. Marginal decrease in accuracy, but testing accuracy closely resembles training accuracy, indicating that overfitting had been reduced.

Stage 6: Changed batch size on newly improved model. Little difference.

Stage 7: Used a larger dataset, but there was no improvement. Reverted to smaller dataset for all other stages.

Stage 8: Lowered learning rate to 0.0001. This configuration yielded the best model.

Stage 9: Pretrained weights generally increased accuracy and greatly lowered loss, but more experimentation is needed, particularly with different optimizers and architectures. Still worse performance than stage 8.

Results

AlexNet

Stage 1: Standard AlexNet architecture. SGD optimizer with a learning rate of 0.01 and momentum of 0.9, batch size of 32, and 10 epochs. Test accuracy of 0.625 and a test loss of 0.692. Peak validation accuracy of 0.75 and miminum validation loss of 0.58. Everything done according to the original AlexNet research paper.

Stage 2: To be more truthful to the original paper, set batch size of 128 and a weight_decay of 0.0005. SGD with an LR of 0.01 and 0.9 momentum, 10 epochs. Test accuracy 0.625, test loss 0.689. Peak val accuracy 0.5, minimum val loss 0.688.

Stage 3: Batch size 128. Adam with LR 0.001, 10 epochs. Test accuracy 0.796, test loss 1.22. Peak val accuracy 0.937, minimum loss 0.10.

Stage 4: Batch size 128, Adam with LR 0.01, 10 epochs. Dense layers with L1 0.01 regularization. Test accuracy 0.625, test loss 69.668. Peak val accuracy 0.5, min val loss 69.671.

Stage 5: Batch size 128, Adam with LR 0.01, 10 epochs. Image augmentation. Test accuracy 0.767, test loss 0.556. Peak val accuracy 0.875, min val loss 0.318.

Stage 6: Batch size 32; only difference from 5th iteration. Adam with LR 0.01, 10 epochs, image augmentation. Test accuracy 0.809, test loss 0.475. Peak val accuracy 0.687, min val loss 0.610.

Stage 7: Larger dataset used. Batch size 128, Adam with LR 0.001, 10 epochs, image augmentation. Test accuracy 0.767, test loss 0.562. Peak val accuracy 0.76, min val loss 0.58.

Stage 8: Batch size 128, Adam with lr = 0.0001, 15 epochs, image augmentation. Test accuracy 0.883, test loss 0.323. Peak val accuracy 0.82, min val loss 0.48.

Stage 9: Skipped.

VGGNet

Stage 1: Standard VGGNet architecture. SGD optimizer with a learning rate of 0.01 and momentum of 0.9, batch size of 32, and 10 epochs. Test accuracy of 0.625 and a test loss of 0.687. Peak validation accuracy of 0.5 and minimum validation loss of 0.693. Everything done according to the original AlexNet research paper.

Stage 2: Set the batch size to 256 as recommended by the original paper. SGD with an LR of 0.01 and 0.9 momentum, 10 epochs. Test accuracy 0.625, test loss 0.690. Peak val accuracy 0.5, minimum val loss 0.646.

Stage 3: Batch size 256, 10 epochs. Adam optimizer with LR 0.001. Test accuracy 0.801, test loss 0.929. Peak val accuracy 0.937, minimimum val loss 0.215.

Stage 4-8: Skipped.

Stage 9: Fine tuned pretrained imagenet weights. SGD with LR 0.0001. Batch size 32, 15 epochs. Test accuracy 0.868, test loss 0.327. Val accuracy and loss not recorded due to printing error.

EfficientNetV2

Stage 1-8: Skipped.

Stage 9: 224x224 3-channel input images provided. Batch size 32, 15 epochs. SGD with LR 0.001. Test accuracy 0.652, test loss 2.059. Peak val accuracy 0.72, min val loss 0.

The pretrained model with SGD with LR of 0.0001 yielded test accuracy 0.733, test loss 0.648, peak val accuracy 0.62, peak loss 0. This was done by finetuning the imagenet model (freezing all but the top two layers) and using a softmax layer with a size of 2 after pooling.

When the adam optimizer was used instead, a test accuracy of 0.772 and test loss of 0.652 was achieved. Peak val accuracy 0.80 and min val loss 0.

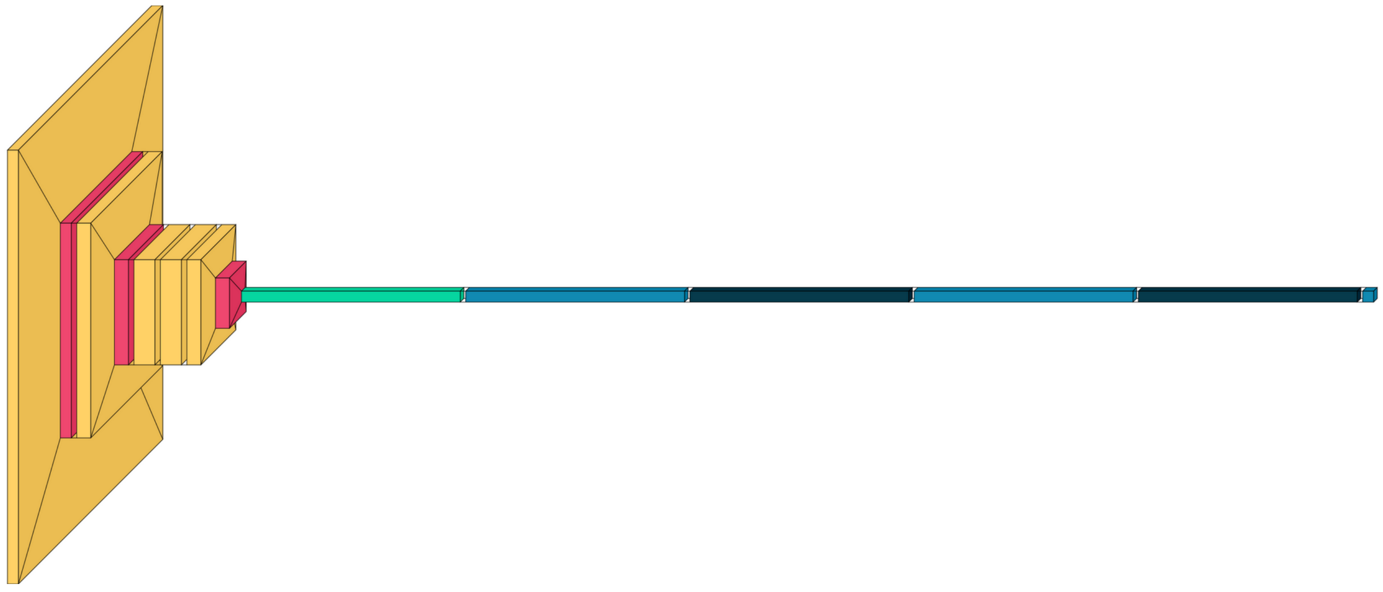
Discussion

The architecture of Stage 8's AlexNet yielded the highest accuracy and lowest loss out of any of the other models. This is likely due to the relative simplicity of the model and its suitability for the simplicity of pneumonia classification; the primary feature needed to detect pneumonia is an abnormal amount of opacity in a single part of the image. Of course, features such as edges and a general shape of the opaque area are still important, and this would clearly point to the high potential of a pretrained model which has already learned many simple features (such as one trained upon the ImageNet dataset), but the hyperparameters chosen for such models were most likely not tuned properly. Considering that a non-pretrained VGGNet performed poorly when using similar hyperparameters as AlexNet, it is possible that a properly tuned pretrained VGGNet could greatly outperform AlexNet.

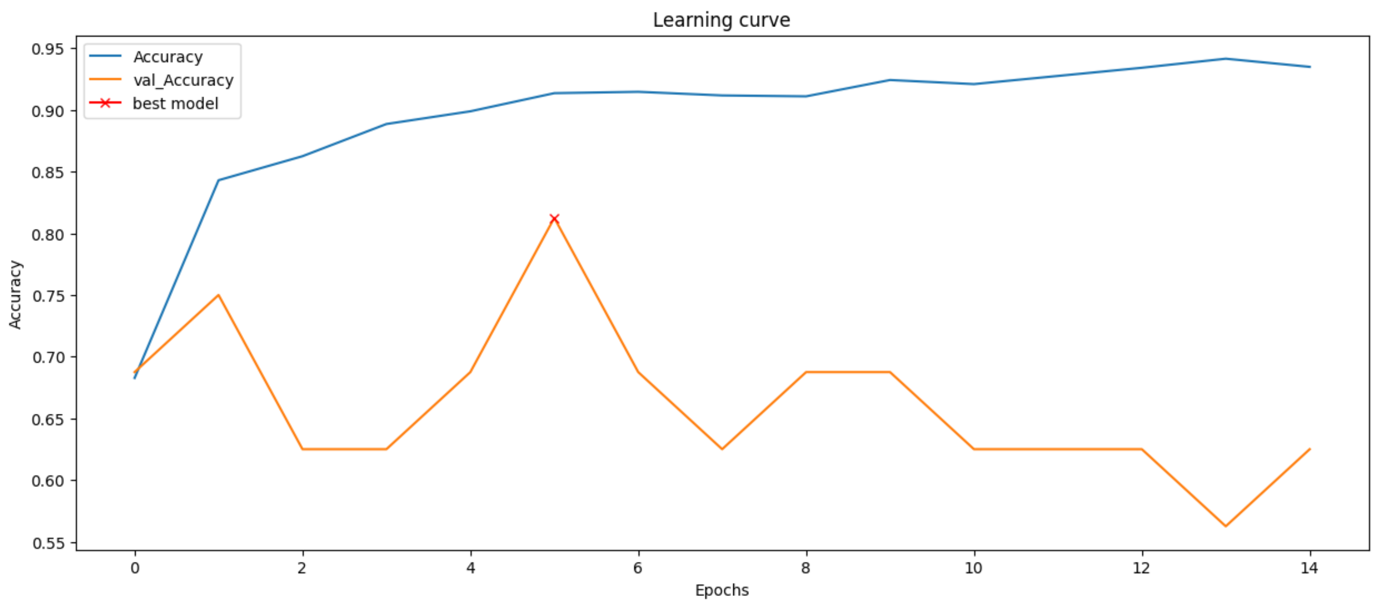
AlexNet Testing Metrics

```
20/20 [==============================] - 162s 8s/step - loss: 0.3232 -
accuracy: 0.8830
Test loss: 0.32321637868881226
Test accuracy: 0.8830128312110901
```

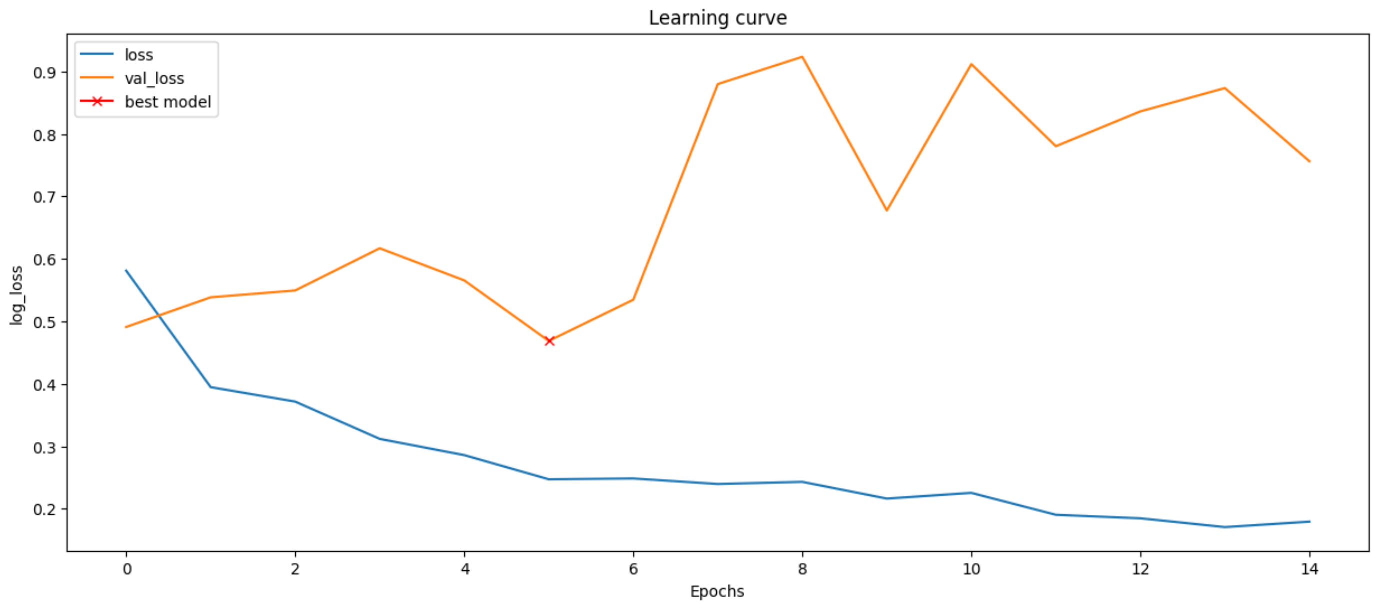
AlexNet Architecture



AlexNet Training Accuracy



AlexNet Training Loss



Summary

The detection of Pneumonia is a binary classification problem which requires numerous features for a neural network to learn. We presented several reasonably accurate two-category classification models, with AlexNet reaching 88% accuracy. Hyperparameter tuning occurred across nine different stages, with heuristic decision making. The layers were left largely unedited besides regularization because their reasoning had been meticulously derived from previous researchers. The dataset did not call for large filters nor strides, and as such alteration of padding was not needed. Similarly, pooling for the features required to detect opacity and the general shape of the problem area is of little concern compared to the optimizer. Learning rate and batch size were of critical importance due to the size of the dataset, as well as the optimizer itself - of which Adam proved to be the most efficient. Categorical Cross-Entropy as the cost function proved to be fit for the problem and was not seen as an issue regarding training; the same goes for the chosen activation function, ReLU - which is lauded in all of the research papers for the included architectures. There is great promise in pretrained weighting from larger datasets.

References

I used the lab programs to kickstart the project, and further modified them using Keras documentation. Throughout the project I often used the module textbooks and ChatGPT to describe concepts to me (which I double checked elsewhere, but didn't inform my decisions beyond my own learning), and I didn't use either to write the code (honestly, this project barely involved coding... most of the time was spent reading and staring at training progress).

I also want to stress that I found a very interesting paper about hyperparameter optimization, which stated that there were numerous heuristic algorithms to try to tune hyperparameters but that it was generally an intractable problem. I lost the file because it didn't sync on onedrive once I left the library, so I can't use it as an actual source - but it was a great resource and described a ton of different techniques and why some filter sizes etc. were better than others.

https://keras.io/api/callbacks/

https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html

https://keras.io/guides/transfer_learning/

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." *Advances in neural information processing systems* 25 (2012).

Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." *arXiv preprint arXiv:1409.1556* (2014).

Tan, Mingxing, and Quoc Le. "Efficientnetv2: Smaller models and faster training." *International conference on machine learning*. PMLR, 2021.