Computer Science Extended Essay

Analyzing the Impact of Configurable Hyperparameters on the Accuracy of different Convolutional Neural Network Architectures in Handwritten Signature Verification

Research Question: Under different hyperparameters, how does the accuracy of VGG16 compare to ResNet50 in the task of handwritten signature verification?

Word count: 3936

Table of Contents

Introduction

Signatures have long served as unique identifiers, serving significance in authorizing agreements and validating transactions. Signatures are valuable for authentication due to their uniqueness, as it is uncommon for two individuals to have identical signatures. The unique combination of strokes that makes up a person's signature serves as a personal identifier distinguishing individuals from another.

However, with the reliance of signatures for such means, an issue arises where individuals may attempt to impersonate others by forging their signatures with fraudulent intentions. Depending on the skill level of the forger, it may be indistinguishable from the owner's genuine signature. Occurrences of signature forgery in today's society shows a possible weakness in the security system when it comes to signature verification. Thus, a need for an accurate and state-of-the-art method to verify the authenticity of signatures is important.

Advancements in Machine Learning (ML) has brought prominence to Neural Networks. A Neural Network (NN) is a ML algorithm inspired by the human brain to process data and learn complex patterns, arriving to a conclusion. With the use of NNs, tasks that require heaps of data are able to be automated. The Convolutional Neural Network (CNN) is a type of NN specialized to analyse visual data. Over many years, the number of CNN architectures has grown rapidly with many improvements seen. With the various amounts of CNN architectures, accuracy is important when it comes to selecting the most appropriate CNN architecture.

Thus, the Research Question (RQ): ***"Under different optimizers, batch sizes, and epochs, how does the accuracy of VGG16 compare to ResNet50 in the task of handwritten signature verification?"***

Handwritten signature verification refers to the process of authenticating a signature. The authenticity is determined by the CNN. This paper will compare two different CNN architectures while also investigating how the accuracy of those architectures is affected by some of its configurable hyperparameters.

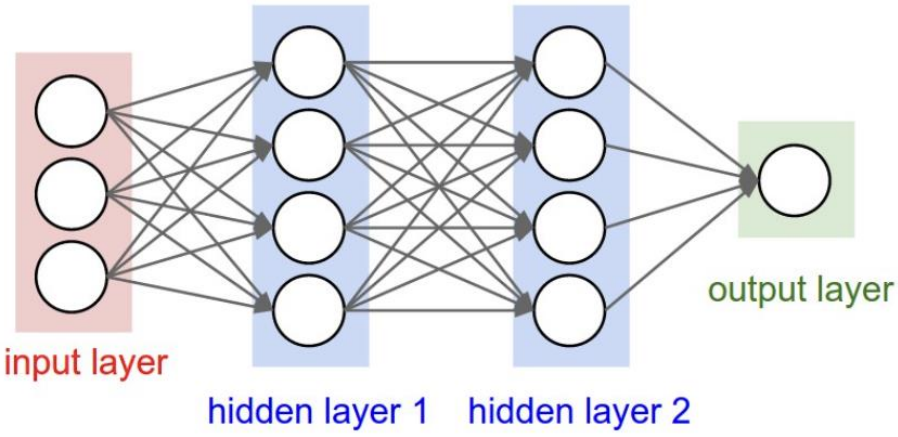
Background Information

Neural Networks

Neural Networks are foundational concepts in Deep Learning (DL). They are formed in layers where data flows through many layers, transforming at each step to produce an output. The architecture of NNs typically consists of three types of layers: an Input Layer, one or more Hidden Layer(s), and an Output Layer (Alzubaidi et. al, 2021).

Figure 1

Neural Network Architecture (Stanford University)



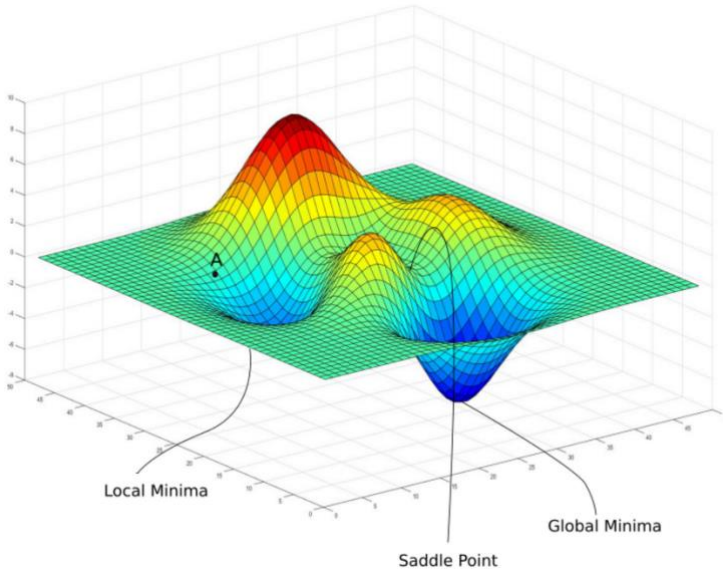
Each layer consists of a certain number of neurons (or nodes), which is where the processing of given data occurs. Each neuron in a layer is connected to other neurons in subsequent layers (Figure 1) through weights and biases, which control the strength of a connection between the neurons (AI Wiki). When a neuron receives an input from the previous layer, it multiplies each input by a corresponding weight, adds a bias, and passes the

result to an activation function which introduces non-linearity. This is significant as non-linear functions are able to handle complex and non-linear relationships within given data (O'Shea & Nash, 2015).

Optimization is important to train the networks. To do so, Gradient Descent is used in training to minimize an NN's error by iteratively updating the network's weights and biases, in other words, the loss function (GeeksforGeeks, 2024). The loss function measures the difference between actual and predicted outputs. The cost function on the other hand, is the value being minimized during training, navigating the optimizer's updates to the model's weights. Calculating the derivative of the function using Gradient Descent to find the minima will lead to the optimal model, where the error is the lowest (Ruder, 2016).

Figure 2

Convex cost function (Yadav, 2021)



This process aims for convergence, where the model's parameters stabilize, and the cost function reaches a low, stable value. However, if the parameters do not stabilize and the global minima of the function is not found, divergence may occur. This results in poor performance on both training and testing data (Pramod, 2023). The learning, which

determines the step size at each parameter update, rate plays a significant role in determining how quickly the algorithm converges to the minimum of the function (Yen, 2023).

Overfitting and Underfitting

Overfitting occurs when the NN learns the training data too well, capturing specific patterns in the data which hinders the network to generalize on unseen data. This can be caused by excessive parameters, poor training data, and training for too long on a data set (GeeksforGeeks, 2024).

Underfitting, on the other hand, occurs when the network is unable to capture the patterns in the data, most likely due to the model being too simple (GeeksforGeeks, 2024), resulting in high error rates which produce unreliable accuracy.

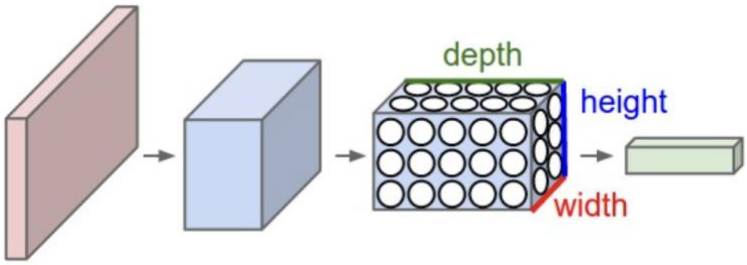
Convolutional Neural Networks

Input Layer

The input layer of a CNN holds the pixel values of the input. This allows CNNs to be more favorable than regular NNs for image-related tasks. CNNs are all about extracting features of an image, then classifying it based on learnt features. Unlike traditional NNs, which flatten inputs into being 1D, CNNs process 2D or 3D structured data, retaining spatial dimensions (height, width) and depth (e.g., RGB color channels) (Alzubaidi et. al, 2021). As the number of weights scales with the size of the input image, using traditional NNs would create an abundance of parameters, increasing the likelihood of overfitting. CNNs handle images more efficiently by assuming that the inputs are images. This is done by having each layer of the network being arranged in 3 dimensions: Width, Height, Depth (Figure 3).

Figure 3

Layers in a CNN (Stanford University)

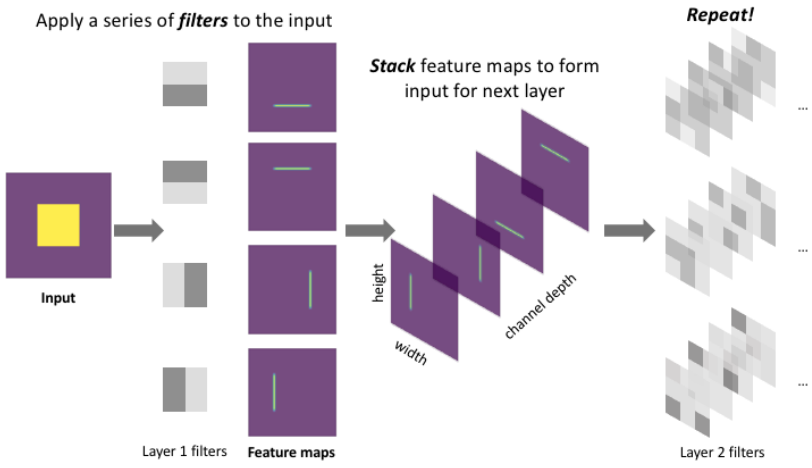


Convolution Layer

The Convolution Layer is essential to the CNN. This layer applies filters (kernels), which are small matrices of weights across the image. Each kernel slides over the image, performing element-wise multiplication with the pixels it overlaps and producing a single value that represents that area that gets stored into a feature map (Figure 4).

Figure 4

Feature Map (Jordan, 2017)



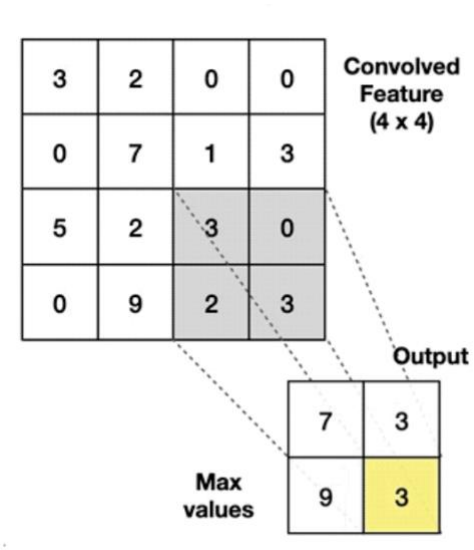
Every kernel will have a corresponding feature map. This will result layers feature maps stacked along the depth dimension to form the full output volume from the convolution layer (O'Shea & Nash, 2015).

Pooling Layer

The Pooling Layer reduces the spatial dimensions of feature maps, while still containing significant patterns, minimizing parameters and reducing the risk of overfitting. This operates over each feature map in the input, and rescales its dimensions (Qayyum, 2022). Common types of pooling operations are: Max Pooling, Sum Pooling, Average Pooling, etc. One of the most used pooling operations in CNNs is Max Pooling, which outputs the maximum value of the feature map based on the pool's kernel size and stride (Digital Ocean, 2024).

Figure 5

Max Pooling on a Feature Map (Qayyum, 2022)



Above is an example of a Max Pooling operation with a Feature map with a size of 4 × 4 and Pool with a size of 2 × 2 and a stride value of 2. Max Pooling is commonly used in CNNs to retain the most significant features of the image, specifically its pixels, emphasizing patterns while ignoring irrelevant and insignificant details.

Fully Connected Layer

Fully Connected Layers (or Dense Layers), are layers of neurons that connect all the previous activations from the feature maps to calculate predictions which are used for classification. Each neuron connects to all neurons in the preceding layer, combining the extracted features to produce a final prediction or classification (O'Shea & Nash, 2015). Simply put, after previous layers have extracted the features of the input image (in this case, handwritten signatures), this layer is responsible for the classification of that image, which in this case is whether the signature is authentic.

Architecture Overview

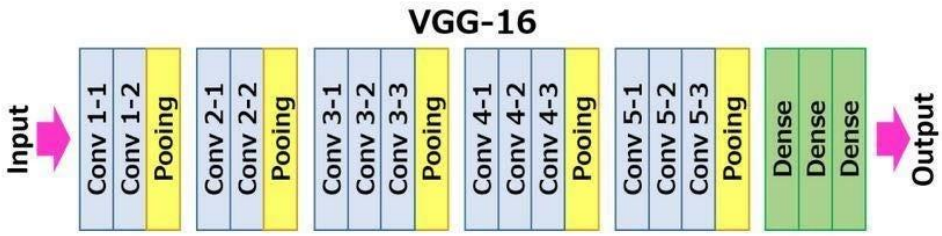
The two architecture models that will be examined are VGG16 and ResNet50.

VGG16

The VGG16 architecture was created by the Visual Geometry Group during the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) in 2014 (Simonyan K. & Zisserman A., 2015). This architecture is characterized by its depth, which consists of a total of 16 layers, including 13 convolution layers and 3 fully connected layers. This architecture follows a series of two and three consecutive convolution layers before a 2×2 max-pooling layer of stride 2.

Figure 6

Architecture of VGG16 (Great Learning)

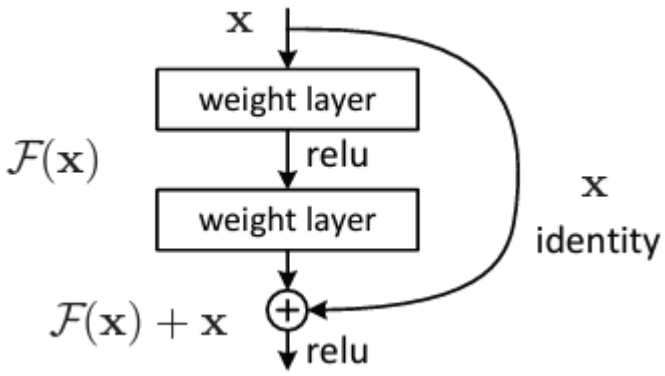


ResNet50

The ResNet50 architecture was developed during the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) in 2015. This deep and complex architecture contains a total of 50 layers, divided into four main blocks. This ResNet (Residual Network) approach was developed to address some challenges present within training deep networks, such as the vanishing gradient problem, where the gradients become very small as they are propagated back through each layer. This causes the network to stop learning efficiently as the weights are barely changing.

Figure 7

Residual learning (He et al., 2015)



Residual Learning is the main idea behind ResNet, where the ResNet layers learn the residual function or the difference between the input and the desired output. This is achieved through the use of skip connections or shortcut connections. This distinguishes ResNet50 from other architectures.

The residual block in ResNet50 is designed to learn the function $F(x) = H(x) - x$, where $H(x)$ is the desired output and x is the input. Using shortcut connections allow the

layers effectively learn the residual, $F(x)$, which simplifies training for very deep networks by allowing the gradient to bypass certain layers, preserving the learning signal and preventing gradients from vanishing (He et al., 2015). This enables ResNet50 to achieve high accuracy while maintaining computational efficiency, even with a large depth of 50 layers.

Comparison of selected Architectures

The VGG16 has a straightforward and simple architecture, albeit its depth. On the other hand, the ResNet50 provides a more complex architecture that distinguishes itself from many different CNN architectures.

These are two popular CNN architectures that have been widely used in computer vision tasks. Both architectures have their own strengths and weaknesses, and they significantly differ in their design, complexity, and performance. The contrasting complexities in these two architectures may reveal how factors such as network depth, layer configuration, and advanced features affect the accuracy of the network. This comparison highlights the relationships between simplicity and complexity in CNN architectures and how they capture patterns in data.

Experiment Methodology

Programming Platforms, Tools and Libraries

The experiment was conducted in the Google Colab Pro environment. A Python3 runtime was used along with the Nvidia A100 GPU hardware accelerator. The main libraries in the experiment are Tensorflow, Keras and OpenCV.

Dataset

The SigComp2011 dataset from the ICDAR2011 Signature Verification Competition (Liwicki et al., 2011) was used for this experiment, specifically the offline Dutch dataset. There are a total of 1287 images in the training dataset, with 648 samples of real signatures and 639 samples of forged ones. The training dataset consists of a total of 362 images, with 239 samples of real signatures and 123 forged ones. For a validation set, the 'validation_split' is set to 0.2 (20%) in the training process using the Keras library (See Appendix B). This means that 20% of the training dataset will be used for validation while training.

Images of the signatures were resized into 224×224×3, as those are the input shapes of both VGG16 and ResNet50.

To distinguish between forged and real signatures, forged signatures are labelled as 1, and real signatures will be labelled as 0. (See Appendix A)

Transfer Learning

Transfer learning is a method where features and learnt from one dataset is then re-used to assess another task with higher performance (Chollet, 2020). Transfer learning is used in this experiment using the pre-trained models of VGG16 and Resnet50 on the ImageNet dataset from the Keras library. (See Appendix B)

Independent Variables (Configurable Hyperparameters)

Other than the CNN architectures themselves, some configurable hyperparameter of the architectures will be an independent variable in this experiment.

Number of Epochs

An epoch refers to the total number of iterations of all the training data in one cycle for training the model (Kumar, 2024). Training with insufficient amounts of epochs may lead to underfitting, while an excessive number of epochs can lead to overfitting. This hyperparameter will be configured in the experiment to assess how both architectures perform with a certain range of epochs. The number of epochs that will be tested for both architectures are 5, 25, and 50.

Batch Size

The batch size determines the number of training examples used in one iteration before the model's parameters are updated. A smaller batch size leads to the model's parameters being more updated frequently. This however, may lead to a less stable gradient. A large batch often exceed lower batch sizes in estimates of the gradient. However, large batch sizes may lead the model into converging to sharp minima of the loss function. It is often the case that lower batch sizes yield better results (Devansh, 2022), but assessing different batch sizes would give further insight of the characteristics of the two architectures. Therefore, an assessment of different batch sizes would provide significant patterns of the performance of the selected architectures. The range of batch sizes in this experiment is 16, 32, and 64.

Optimizer

The optimizer affects how the model updates its weights based on the gradient of the loss function. The optimizers that will be tested are Stochastic Gradient Descent (SGD) and Root Mean Square Propagation (RMSprop).

SGD is one variant of Gradient Descent. SGD uses a single random or a small batch of samples of the training dataset at each iteration. (Roy, 2024). After looking at each training example, the parameters are updated. This frequent updating can lead to faster convergence, especially for large datasets. The noise introduced by the single sample update helps the model to escape local minima and potentially find a better solution. However, due to it updating its parameters one at a time, the path taken to reach the minima is much noisier. This may result in the model not converging to the most optimal solution.

RMSprop was developed to address the issues with the SGD algorithm. RMSprop maintains a moving average of the squared gradients for each parameter. This moving average is updated within each iteration to give more weight to recent gradients while decreasing the impact of earlier gradients. This slows down the learning rate for weights with large gradients and increased it for weights with small gradients (Dijkinga, 2024). However, although some models benefit from RMSprop's adaptive learning rates, it may lead to overfitting by quickly converging to a local minimum that fits the training data closely but does not generalize well to unseen data.

Dependent Variable

The dependent variable of this experiment will be the Testing Accuracy%. The testing accuracy results will be gained through the process below:

Figure 8

Testing Accuracy% extraction code

```
predictions = model.predict(test_data) #Use the trained model to predict test / unseen data

predicted_classes = np.argmax(predictions, axis=1)

accuracy = accuracy_score(test_labels, predicted_classes)
print(f"Testing Accuracy: {accuracy * 100:.2f}%") #Output the results of the Accuracy score (%)
```

After the network is trained, the model will be tested to predict whether the signatures in the testing dataset is real or forged.

Controlled Variables

Development Environment

- Google Colab Pro environment

Resource Specifications

- NVIDIA A100 Hardware Accelerator
- 83.5GB System Ram
- 40.0 GPU RAM
- 201.2GB Disk Space

Dataset

- ICDAR2011 SigComp2011 Dutch Dataset

Hypothesis

It can be hypothesized that the ResNet50 architecture will produce better accuracy results compared to the VGG16 architecture when preforming handwritten signature verification under similar conditions. This is due to its more complex architecture, especially with Residual Learning. Although VGG16 has a straightforward and deep architecture with 16 layers, it lacks the residual connections of ResNet50. Furthermore, ResNet50 is significantly deeper than VGG16, as ResNet50 contains 50 layers, whereas VGG16 has only 16.

However, the performance of the two architectures may vary depending on the values of the independent variables. Specifically, a combination of a low batch size and a high epoch would most likely result in higher accuracy. But ultimately, ResNet50 is hypothesized to demonstrate superior performance.

Experiment Results

Tabular Presentation of Experimental Results

Table 1

Accuracy Values of VGG16 and ResNet50 based on different configurations of hyperparameters

Optimizer	Batch Size	Epochs	Testing Accuracy% (VGG16)	Testing Accuracy% (Resnet50)
SGD	16	5	48.90	62.98
		10	79.27	76.24
		25	82.32	78.73
	32	5	50.55	49.17
		10	58.56	66.85
		25	70.17	66.57
	64	5	57.18	31.22
		10	44.48	66.02
		25	63.26	36.74
RMSProp	16	5	77.62	66.02
		10	77.90	85.64
		25	78.45	85.91
	32	5	82.04	66.02
		10	76.80	66.02
		25	77.35	66.02
	64	5	72.65	64.64
		10	70.72	69.3
		25	84.25	65.75

Table 2

Sorted Accuracy Values of VGG16

Optimizer	Batch Size	Epochs	Testing Accuracy% (VGG16)
SGD	64	10	44.48
SGD	16	5	48.9
SGD	32	5	50.55
SGD	64	5	57.18
SGD	32	10	58.56
SGD	64	25	63.26
SGD	32	25	70.17
RMSPROP	64	10	70.72
RMSPROP	64	5	72.65
RMSPROP	32	10	76.8
RMSPROP	32	25	77.35
RMSPROP	16	5	77.62
RMSPROP	16	10	77.9
RMSPROP	16	25	78.45
SGD	16	10	79.27
RMSPROP	32	5	82.04
SGD	16	25	82.32
RMSPROP	64	25	84.25

Table 3

Sorted Accuracy Values of ResNet50

Optimizer	Batch Size	Epochs	Testing Accuracy% (ResNet50)
SGD	64	5	31.22
SGD	64	25	36.74
SGD	32	5	49.17
SGD	16	5	62.98
RMSPROP	64	5	64.64
RMSPROP	64	25	65.75
SGD	64	10	66.02
RMSPROP	16	5	66.02
RMSPROP	32	5	66.02
RMSPROP	32	10	66.02
RMSPROP	32	25	66.02
SGD	32	25	66.57
SGD	32	10	66.85
RMSPROP	64	10	69.3
SGD	16	10	76.24
SGD	16	25	78.73
RMSPROP	16	10	85.64
RMSPROP	16	25	85.91

Table 4

Combined table of sorted Accuracy between ResNet50 and VGG16

Architecture	Optimizer	Batch Size	Epochs	Testing Accuracy%
ResNet50	SGD	64	5	31.22
ResNet50	SGD	64	25	36.74
VGG16	SGD	64	10	44.48
VGG16	SGD	16	5	48.9
ResNet50	SGD	32	5	49.17
VGG16	SGD	32	5	50.55
VGG16	SGD	64	5	57.18
VGG16	SGD	32	10	58.56
ResNet50	SGD	16	5	62.98
VGG16	SGD	64	25	63.26
ResNet50	RMSPROP	64	5	64.64
ResNet50	RMSPROP	64	25	65.75
ResNet50	SGD	64	10	66.02
ResNet50	RMSPROP	16	5	66.02
ResNet50	RMSPROP	32	5	66.02
ResNet50	RMSPROP	32	10	66.02
ResNet50	RMSPROP	32	25	66.02
ResNet50	SGD	32	25	66.57
ResNet50	SGD	32	10	66.85
ResNet50	RMSPROP	64	10	69.3
VGG16	SGD	32	25	70.17
VGG16	RMSPROP	64	10	70.72
VGG16	RMSPROP	64	5	72.65
ResNet50	SGD	16	10	76.24
VGG16	RMSPROP	32	10	76.8
VGG16	RMSPROP	32	25	77.35
VGG16	RMSPROP	16	5	77.62
VGG16	RMSPROP	16	10	77.9
VGG16	RMSPROP	16	25	78.45
ResNet50	SGD	16	25	78.73
VGG16	SGD	16	10	79.27
VGG16	RMSPROP	32	5	82.04
VGG16	SGD	16	25	82.32
VGG16	RMSPROP	64	25	84.25
ResNet50	RMSPROP	16	10	85.64
ResNet50	RMSPROP	16	25	85.91

Table 5

Frequency list of Independent Variables above and below the Average Testing Accuracy%

Independent Variables		Occurences	
		Above Average Testing Accuracy%	Below Average Testing Accuracy%
Architectures	ResNet50	5	13
	VGG16	12	6
Optimizers	RMsprop	12	6
	SGD	5	13
Batch Sizes	64	8	4
	32	4	8
	16	9	3
Amount of Epochs	25	7	5
	10	7	5
	5	3	9

Note. Independent variables with the most frequent occurrences are highlighted yellow

Graphical Presentation of Experimental Results

Figure 9

VGG16 Line chart of Comparison of Optimisers and Batch Sizes with respect to number of Epochs

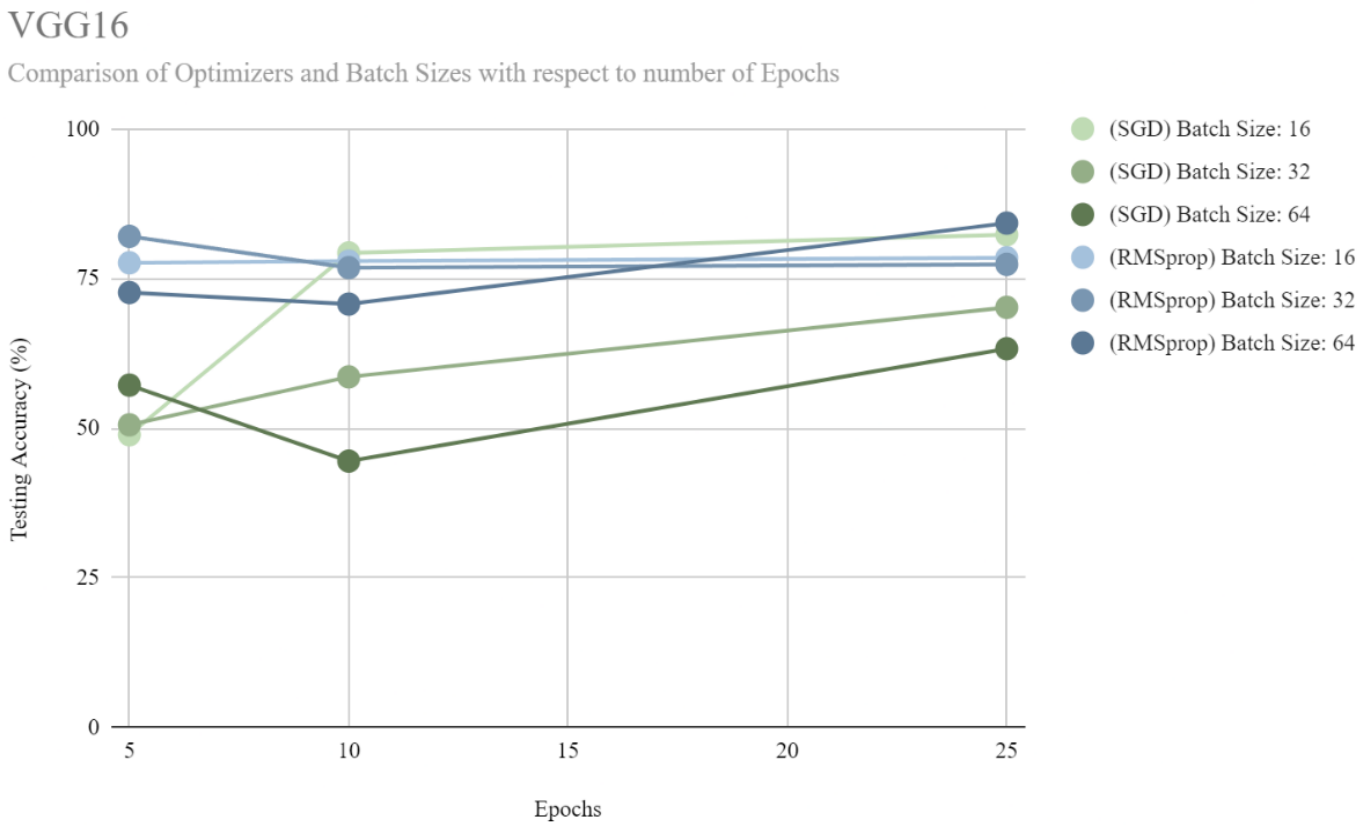
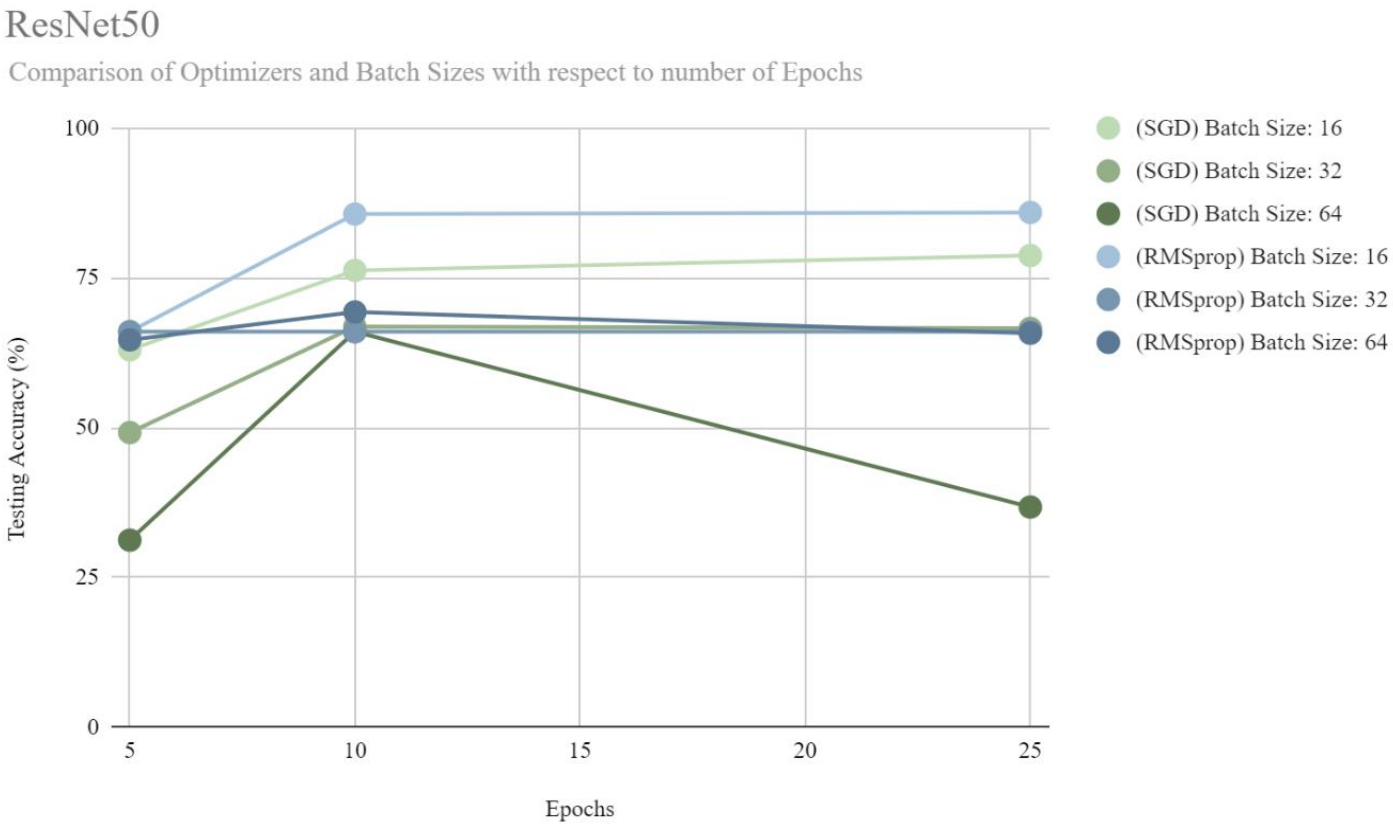


Figure 10

ResNet50 Line chart of Comparison of Optimisers and Batch Sizes with respect to number of Epochs



Data Analysis

Best Performing Models

ResNet50 with the RMSprop optimizer, a batch size of 16, and 25 epochs achieved the highest testing accuracy at 85.91% (see Table 4). ResNet50 with RMSprop and 10 epochs and VGG16 with RMSprop and a batch size of 64 at 25 epochs also performed well, indicating that RMSprop with higher epochs generally yields high accuracy, especially with ResNet50. The smaller batch size of 16 likely contributed to better generalization through more frequent parameter updates, which can help the optimizer escape from local minima and navigate the complex loss function in ResNet50.

Analysis Between Optimisers

Results have shown that RMSprop generally outperformed SGD across most configurations, providing more stable accuracy due to its adaptive learning rate, preventing large fluctuations and improves convergence (see Table 5). Both VGG16 and ResNet50's accuracies showed greater consistency with RMSprop, which helped address issues such as overfitting or getting stuck in suboptimal regions in the loss function. Although Appendix D shows some instability in validation loss, it still manages to come to a high testing and validation accuracy, and a low testing and validation loss.

With SGD, ResNet50's accuracy decreased from 66.02% to 36.74% at higher epochs with a batch size of 64 (see Table 1), which implies SGD's tendency toward instability and unpredictability in deeper networks. Figures D1 to D9 shows that SGD is sensitive to overfitting, as the training loss often decreases but the validation loss increases. Furthermore, SGD's unpredictable nature can be seen in VGG16's trends (Figure 9), with inconsistent accuracy values as epochs increase. Thus, it can be concluded that RMSprop shows superiority over SGD in terms of accuracy. This is mostly due to RMSprop's adaptive

learning rates that provide significant advantages over SGD, leading to higher accuracy and a more stable convergence. This complies ResNet50's skip connections, effectively managing gradients of the loss function within deeper layers which thus produces higher accuracies.

Analysis of Batch Sizes

Lower batch sizes, particularly 16, consistently leads to higher testing accuracy for both architectures (see Table 5). Smaller batches cause more frequent parameter updates, causing better generalization of data. This is evident in Table 5 where batch size 16 produces the most frequent accuracy values that are above the mean. Batch size 64, however, performed inconsistently, especially when paired with SGD in ResNet50, likely due to the two optimizer's lack of adaptability with large batches. ResNet50's highest accuracies were strongly connected to batch size 16 (see Table 3), suggesting that its deeper architecture and loss function benefit from smaller batches. VGG16, on the other hand, showed more flexibility with batch sizes, achieving high accuracy with both batch sizes of 16 and 64 (see Table 2). This may be due to VGG16's simpler architecture and shallower depth, making it less sensitive to batch size variations.

However, batch size 32 and larger often causes ResNet50 to get stuck at a saddle point where, during the training phase, the training loss plateaus after some epochs. An accuracy of 66.02% (see Table 1) is constantly reached with batch size 32, specifically using RMSprop. Figures D13 to D15 confirms this by showing inconsistent testing and validation losses, with testing and validation accuracies being stable. Figures D7 to D9 suggests that ResNet50 using batch size 64 tends to overfit during the training phase, where testing accuracy decreases in a stabilized manner, but the validation accuracy does not. This explains the trend in Figure 10, where higher batch sizes fall under the lines with higher batch sizes.

On the other hand, training results (see Figures E7 to E9, E16 to E18) implies that VGG16 performs well during training, shown by the stable decrease of both testing and validation loss, and increases of testing and validation accuracy. This justifies VGG16's highest accuracy (see Table 2).

Analysis of Epochs

Testing accuracy generally improved with more epochs, particularly at 10 and 25 epochs, which achieved above-average accuracy in most cases (Table 5). For ResNet50, accuracy increased significantly from 5 to 10 epochs, with smaller improvements between 10 and 25 epochs (Figure 10). However, high epochs sometimes led to overfitting, as seen with SGD at a batch size of 64. VGG16 followed a similar trend but showed occasional decreases in accuracy from 5 to 10 epochs, particularly with SGD, before improving substantially at 25 epochs, especially with RMSprop (Figure 9). Generally, a majority of the trials show that testing accuracies either increase or stabilize at higher epochs (Figure 9 and 10). However, an outlier is ResNet50 with batch size 64 at 25 epochs (Figure 10), where the accuracy drops substantially from 10 to 25 epochs. This suggests that the model may have overfitted over an extended period of iterations, which may be linked to the amount of batch sizes and the optimizer, as stated previously.

Analysis of Architecture

Between ResNet50 and VGG16, VGG16 showed more stable accuracy trends across different configurations, benefiting from extended epochs and larger batch sizes. This architecture's straightforward design enables consistent learning with various parameter choices, particularly with RMSprop. ResNet50, a deeper network, was more sensitive to the changes of its parameters, performing optimally with RMSprop, smaller batch sizes, and moderate epochs. RMSprop's adaptive learning rates allowed ResNet50 to navigate its

complex architecture effectively, achieving the highest accuracy. Although ResNet50 achieved the highest accuracy, the difference is marginal from VGG16's maximum accuracy (see Table 4).

ResNet50, being a deeper network with 50 layers and its residual connections, has shown to be more sensitive to changes in its parameters. Its performance was most optimal when trained with RMSprop, smaller batch sizes, and moderate epochs. The depth and complexity of ResNet50 make it more challenging to train, as it requires careful configuration of parameters for effective learning and convergence. RMSprop's adaptive learning rates allowed ResNet50 to navigate its complex architecture effectively by adjusting learning rates for each parameter, which is significant for deep networks where different layers may require different learning rates.

VGG16 shows more stable accuracy trends across different configurations, benefiting from larger epochs and larger batch sizes (see Figure 9). VGG16's simple architecture, using sequential layers of small 3 × 3 convolutions and a smaller depth, contributes to its consistency in learning across different parameter configurations. This simplicity allows VGG16 to train effectively without the problems that can occur in deep networks, such as ones in ResNet50.

Conclusion

Evaluation of Method and Extensions

While the method somewhat answered the Research Question sufficiently; considering the performance of the models and usage of resources in detail, many potential errors and rooms for improvement are seen.

Firstly, the lack of Data Augmentation in the method is a significant limitation. Handwritten signature verification is a task that is inherently variable. Signatures can differ

due to changes in writing speed, angle, pressure, and even the medium on which they are written. Using Data Augmentation methods would have increased versatility within the dataset, which could reveal further insight of this research. Since Data Augmentation was not incorporated, the models were only exposed to the limited variability present in the dataset. This could lead to overfitting and limited feature learning of the models. Prior results shown that some trials exhibited overfitting, which shows the need for Data Augmentation.

Secondly, the exclusion other performance metric such as Root Mean Squared Error, F1 Score, Precision, and thereof in this paper is a significant limitation. Though this paper aims do discuss accuracy, evaluating and analysing different evaluation metrics could provide more information regarding the models' accuracy.

Lastly, the Research Question could have been answered in a more generalizable manner if the analysis was conducted over multiple datasets. Doing so will introduce the architectures to a more versatile dataset, expanding the range and significance of this paper.

To extend, factors such as learning rate and other performance metrics can be evaluated to find the most optimal model. Additionally, the research could extend to more Neural Network Architectures such as InceptionV3, AlexNet, GoogLeNet, etc.

Final Conclusions

The main goal of this research is to compare the accuracy of VGG16 and ResNet50 architectures by varying its configurable hyperparameters on handwritten signature verification. Results have shown that while ResNet50 achieved slightly higher accuracy, it may not be inherently superior. VGG16's consistent performance suggests it may generalize well under limited data. Although ResNet50 achieved the greatest accuracy, VGG16 shows superior consistency with higher frequencies of results above the average accuracy. Regarding the hyperparameters, ResNet50 preforms the best when using lower batch sizes for

higher epochs. VGG16 is not far off, showing a higher degree of accuracy with varied epochs and batch sizes. These results imply that while ResNet50 has the theoretical advantage of being able to learn more complex representations due to its depth, in practice, VGG16 can achieve similar performance on the task of handwritten signature verification. The minute difference implies that the increased complexity and training sensitivity of ResNet50 may not provide a significant advantage over VGG16 in this specific application.

References

AI Wiki. (2021). *Weights and biases*. Paperspace. https://machine-learning.paperspace.com/wiki/weights-and-biases

Alzubaidi, L., Zhang, J., Humaidi, A. J., Al-Dujaili, A., Duan, Y., Al-Shamma, O., Santamaría, J., Fadhel, M. A., Al-Amidie, M., & Farhan, L. (2021). Review of deep learning: Concepts, CNN architectures, challenges, applications, future directions. *Journal of Big Data*, *8*(1), Article 53. https://doi.org/10.1186/s40537-021-00444-8

Chollet, F. (2020, April 15). *Transfer learning & fine-tuning*. Keras. https://keras.io/guides/transfer_learning/

Derushi, K. (2021, March 21). *Convolutional neural network (CNN) architectures*. GeeksforGeeks. https://www.geeksforgeeks.org/convolutional-neural-network-cnn-architectures/

Devansh. (2020, January 17). *How does batch size impact your model learning?* Medium. https://medium.com/geekculture/how-does-batch-size-impact-your-model-learning-2dd34d9fb1fa

DigitalOcean. (n.d.). *Pooling in convolutional neural networks*. https://www.digitalocean.com/community/tutorials/pooling-in-convolutional-neural-networks

Dijkinga, F. (2020, April 13). *The RMSprop optimizer*. Medium. https://medium.com/@fernando.dijkinga/the-rmsprop-optimizer-78f02efb63e9

GeeksforGeeks. (2020, February 15). *How does gradient descent and backpropagation work together?* GeeksforGeeks. https://www.geeksforgeeks.org/how-does-gradient-descent-and-backpropagation-work-together/

GeeksforGeeks. (2020, March 11). *ML | Underfitting and overfitting*. GeeksforGeeks. https://www.geeksforgeeks.org/underfitting-and-overfitting-in-machine-learning/

Great Learning. (2021). *Everything you need to know about VGG16*. Medium. https://medium.com/@mygreatlearning/everything-you-need-to-know-about-vgg16-7315defb5918

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 770–778). https://doi.org/10.1109/CVPR.2016.90

IBM. (n.d.). *What is gradient descent?* IBM. https://www.ibm.com/topics/gradient-descent

Jordan, J. (2017, July 26). *Convolutional neural networks*. https://www.jeremyjordan.me/convolutional-neural-networks/

Kumar, A. (2020a, December 4). *Different types of CNN architectures explained: Examples*. Vitalflux. https://vitalflux.com/different-types-of-cnn-architectures-explained-examples/

Kumar, A. (2020b, November 13). *What is epoch in machine learning?* Simplilearn. https://www.simplilearn.com/tutorials/machine-learning-tutorial/what-is-epoch-in-machine-learning

Liwicki, M., Blumenstein, M., van den Heuvel, E., Berger, C. E. H., Stoel, R. D., Found, B., Chen, X., & Malik, M. I. (2011). SigComp11: Signature verification competition for on- and offline skilled forgeries. In *2011 International Conference on Document Analysis and Recognition* (pp. 1480–1484). IEEE. https://doi.org/10.1109/ICDAR.2011.297

O'Shea, K., & Nash, R. (2015). *An introduction to convolutional neural networks*. arXiv preprint arXiv:1511.08458. https://arxiv.org/abs/1511.08458

Qayyum, R. (2020, August 12). *Introduction to pooling layers in CNN*. Towards AI. https://pub.towardsai.net/introduction-to-pooling-layers-in-cnn-dafe61eabe34

Roy, R. (2020, March 14). *ML | Stochastic gradient descent (SGD)*. GeeksforGeeks. https://www.geeksforgeeks.org/ml-stochastic-gradient-descent-sgd/

Ruder, S. (2016). *An overview of gradient descent optimization algorithms*. arXiv preprint arXiv:1609.04747. https://arxiv.org/abs/1609.04747

Simonyan, K., & Zisserman, A. (2015). *Very deep convolutional networks for large-scale image recognition*. arXiv preprint arXiv:1409.1556. https://arxiv.org/abs/1409.1556

Stanford University. (n.d.). *CS231n: Deep learning for computer vision*. https://cs231n.stanford.edu/

Yadav, P. (2021, February 25). *The journey of gradient descent—From local to global*. Medium. https://medium.com/analytics-vidhya/journey-of-gradient-descent-from-local-to-global-c851eba3d367

Yen, N. (2020, May 31). *A beginner's guide to gradient descent in machine learning*. Medium. https://medium.com/@yennhi95zz/4-a-beginners-guide-to-gradient-descent-in-machine-learning-773ba7cd3dfe

Appendix A

This appendix shows the setup and processing of the dataset.

Figure A1

Imports and dependencies

```
import numpy as np
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
import cv2
import os
import random
import glob
import tensorflow
from tensorflow.keras.layers import Conv2D, Flatten, MaxPooling2D, Dense, Dropout, BatchNormalization
from tensorflow.keras.models import Sequential
from tensorflow.keras.preprocessing import image
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
from tensorflow.keras.callbacks import ReduceLROnPlateau
from tensorflow.keras.applications.vgg16 import VGG16
from sklearn.model_selection import train_test_split
from warnings import filterwarnings
filterwarnings("ignore")
```

Figure A2

Path and directory configuration

```
[ ]  from google.colab import drive
     drive.mount('/content/drive')

     Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).

[ ]  path = "/content/drive/MyDrive/icdar_re0/icdar_re/"
     directory = "/content/drive/MyDrive/icdar_re0/icdar_re/"
```

Figure A3

Dataset processing and labelling

```
train_dir = path + "train/"
test_dir = path + "test/"


SIZE = 224

train_data = []
train_labels = []

for i in os.listdir(path + 'train/'):
    for j in glob.glob(path + 'train/'+i+'/*.*'):
        img = cv2.imread(j)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        img = cv2.resize(img, (SIZE,SIZE))
        train_data.append(img)
        if i == 'FORGED':
            train_labels.append(np.array(1))
        else:
            train_labels.append(np.array(0))

train_data = np.array(train_data)/255.0
train_labels = np.array(train_labels)


test_data = []
test_labels = []

for i in os.listdir(path + 'test/'):
    for j in glob.glob(path + 'test/'+i+'/*.*'):
        img = cv2.imread(j)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        img = cv2.resize(img, (SIZE,SIZE))
        test_data.append(img)
        if i == 'FORGED':
            test_labels.append(np.array(1))
        else:
            test_labels.append(np.array(0))

test_data = np.array(test_data)/255.0
test_labels = np.array(test_labels)
```

Figure A4

Reshaping and shuffling processed images

```
from keras.utils import to_categorical
train_labels = to_categorical(train_labels)


train_data = train_data.reshape(-1, SIZE,SIZE, 3)
test_data = test_data.reshape(-1, SIZE,SIZE, 3)


from sklearn.utils import shuffle
train_data,train_labels = shuffle(train_data,train_labels)
test_data,test_labels = shuffle(test_data,test_labels)
```

Appendix B

Figure B1

Configurable hyperparameters and Dense Layers

```
from keras.models import Sequential, Model, load_model
from keras import applications
from keras import optimizers
from keras.layers import Dropout, Flatten, Dense
from keras.optimizers import Adam, SGD, RMSprop

ARCHITECTURE = "ResNet50" #{ResNet50, VGG16}

if ARCHITECTURE == "ResNet50":
    base_model = applications.ResNet50(weights='imagenet', include_top=False, input_shape=(224,224,3))
elif ARCHITECTURE == "VGG16":
    base_model = applications.VGG16(weights='imagenet', include_top=False, input_shape=(224,224,3))
base_model.summary()

add_model = Sequential()
add_model.add(Flatten(input_shape=base_model.output_shape[1:]))
add_model.add(Dense(256, activation='relu'))
add_model.add(Dense(2, activation='softmax'))

model = Model(inputs=base_model.input, outputs=add_model(base_model.output))

OPTIMIZER = "SGD" #{SGD, RMSprop}

if OPTIMIZER == "SGD":
   model.compile(loss='categorical_crossentropy', optimizer=optimizers.SGD(learning_rate=0.0001),
              metrics=['accuracy'])
elif OPTIMIZER == "RMSprop":
    optimizer = model.compile(loss='categorical_crossentropy', optimizer=optimizers.RMSprop(learning_rate=0.0001),
              metrics=['accuracy'])

model.summary()
```

Figure B2

Training process

```
EPOCHS = 5 #{5, 10, 25}
BS = 32 #{16, 32, 64}
progess = model.fit(train_data,train_labels, batch_size=BS,epochs=EPOCHS,validation_split=.2)
```

Appendix C

Figure C1

Predictions of trained model

```
predictions = model.predict(test_data)

predicted_classes = np.argmax(predictions, axis=1)

accuracy = accuracy_score(test_labels, predicted_classes)
print(f"Testing Accuracy: {accuracy * 100:.2f}%")
```

Appendix D

This appendix shows training results for ResNet50 across all configurations of hyperparameters.

Figure D1

Training Results for ResNet50 with SGD and batch size 16 and 5 epochs

```
Epoch 1/5
65/65 ━━━━━━━━━━━━━━━━━━━━ 2s 34ms/step - accuracy: 0.9989 - loss: 0.0169 - val_accuracy: 0.5465 - val_loss: 1.2671
Epoch 2/5
65/65 ━━━━━━━━━━━━━━━━━━━━ 2s 31ms/step - accuracy: 0.9988 - loss: 0.0156 - val_accuracy: 0.5659 - val_loss: 1.2137
Epoch 3/5
65/65 ━━━━━━━━━━━━━━━━━━━━ 2s 31ms/step - accuracy: 0.9998 - loss: 0.0132 - val_accuracy: 0.5465 - val_loss: 1.2763
Epoch 4/5
65/65 ━━━━━━━━━━━━━━━━━━━━ 2s 30ms/step - accuracy: 0.9933 - loss: 0.0212 - val_accuracy: 0.5543 - val_loss: 1.2596
Epoch 5/5
65/65 ━━━━━━━━━━━━━━━━━━━━ 2s 31ms/step - accuracy: 0.9972 - loss: 0.0149 - val_accuracy: 0.5543 - val_loss: 1.6576
```

Figure D2

Training Results for ResNet50 with SGD and batch size 16 and 10 epochs

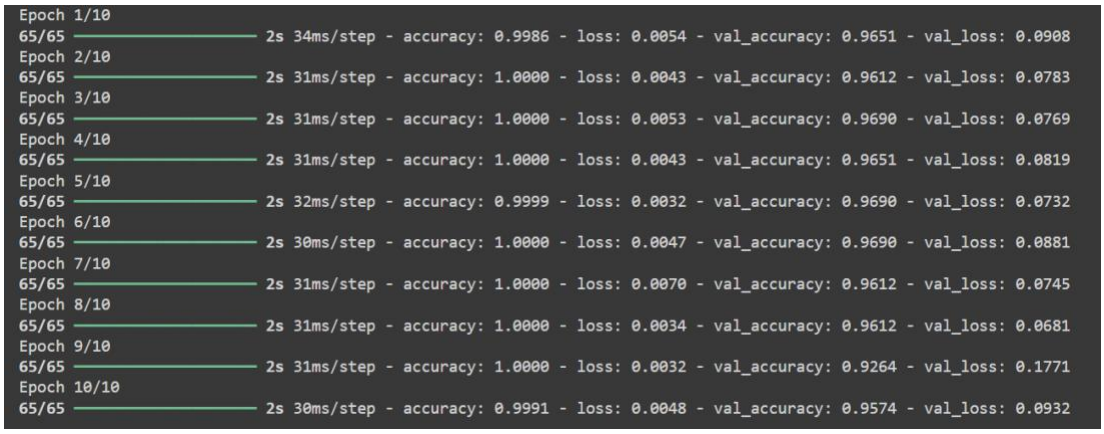


Figure D3

Training Results for ResNet50 with SGD and batch size 16 and 25 epochs

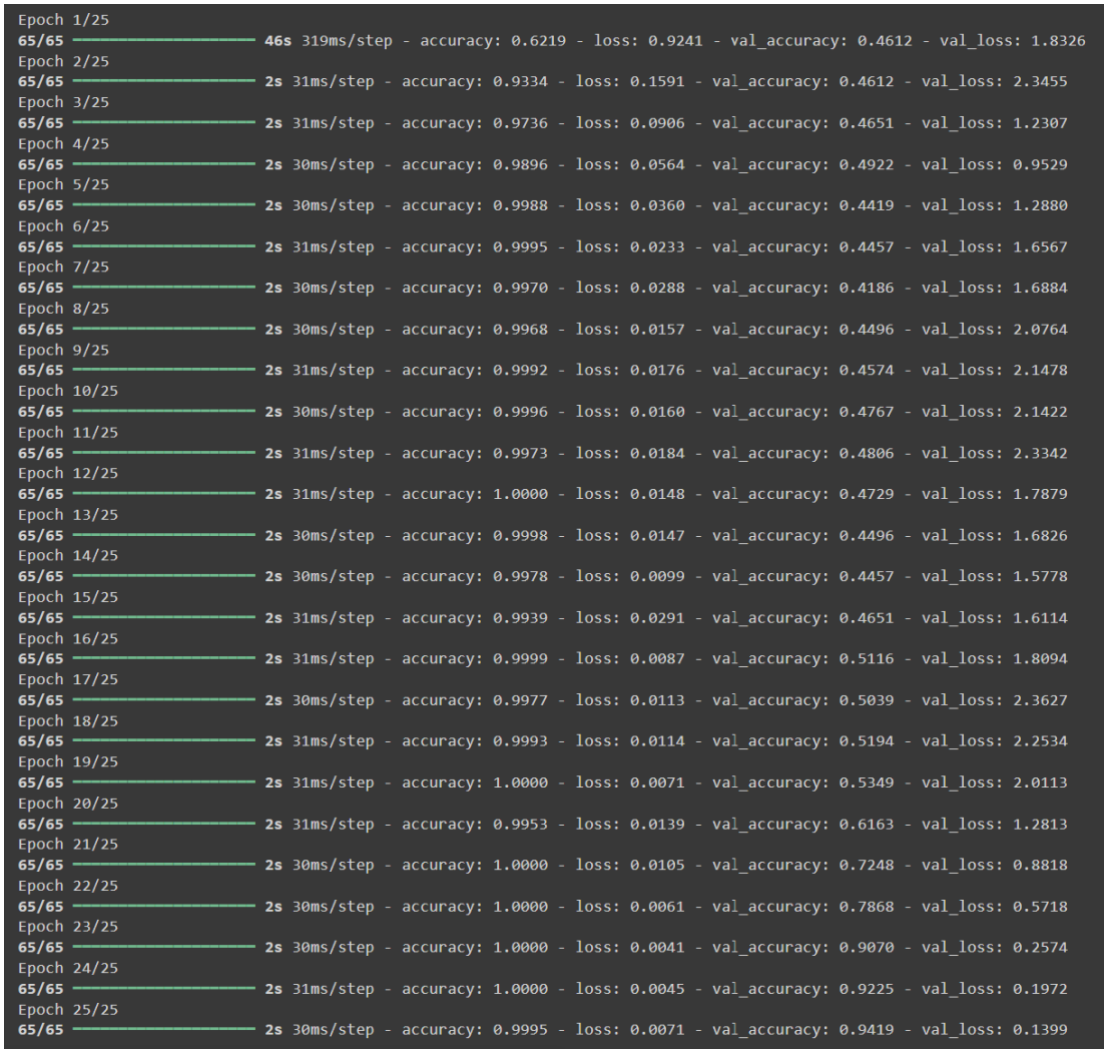


Figure D4

Training Results for ResNet50 with SGD and batch Size and 5 epochs

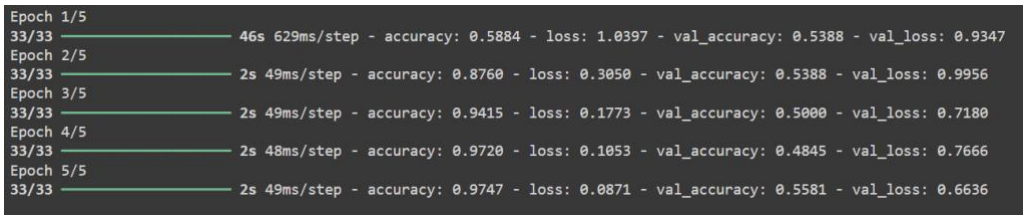


Figure D5

Training Results for ResNet50 with SGD and batch Size 32 and 10 epochs

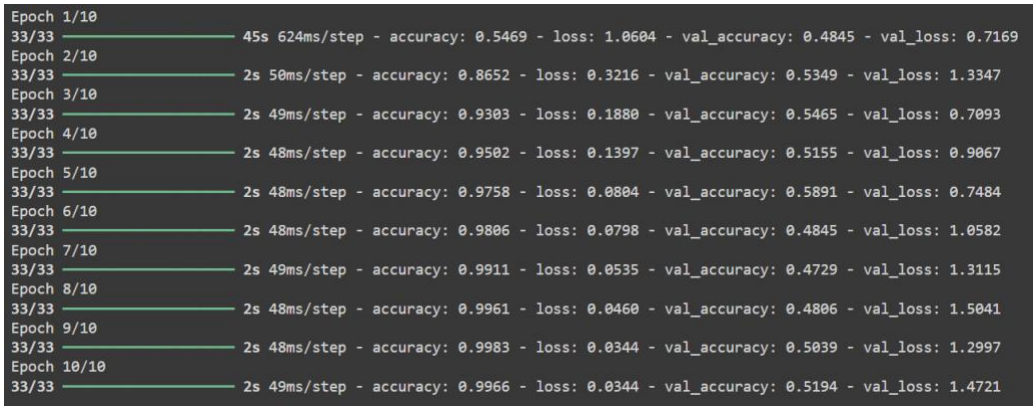


Figure D6

Training Results for ResNet50 with SGD and batch Size 32 and 25 epochs

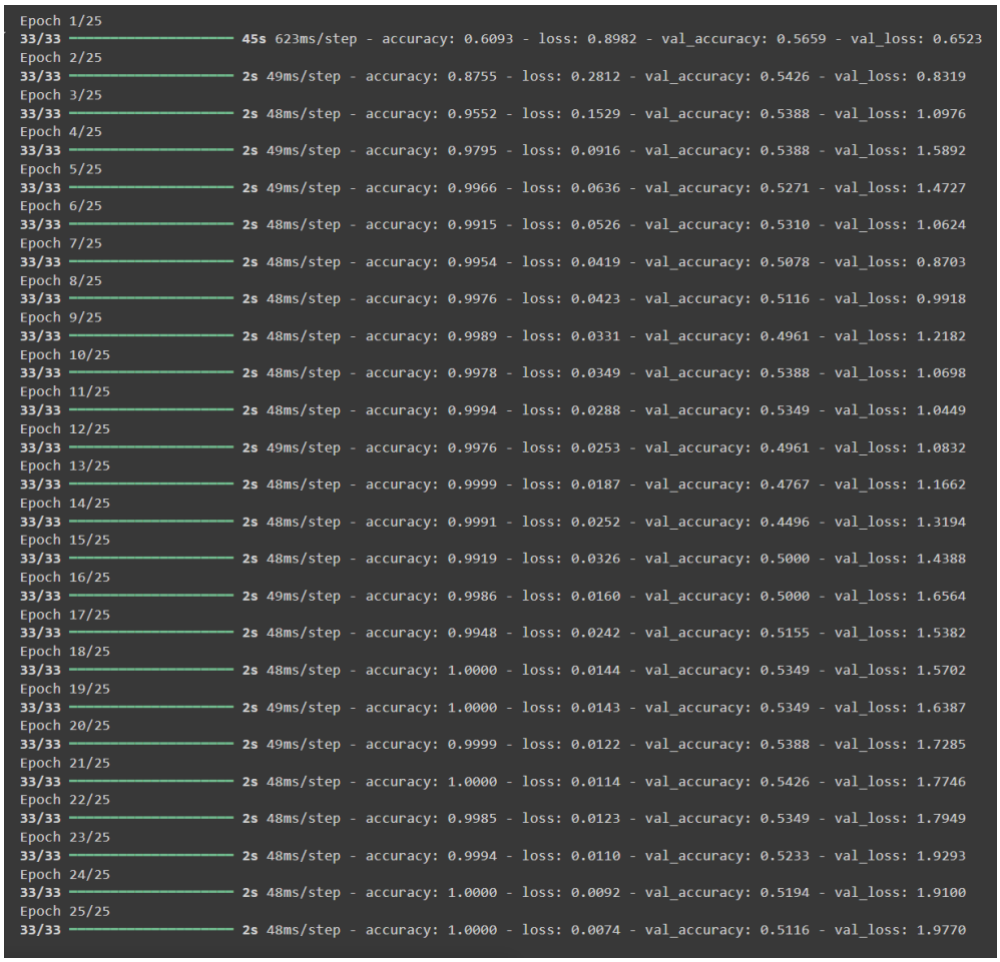


Figure D7

Training Results for ResNet50 with SGD and batch size of 64 and 5 epochs

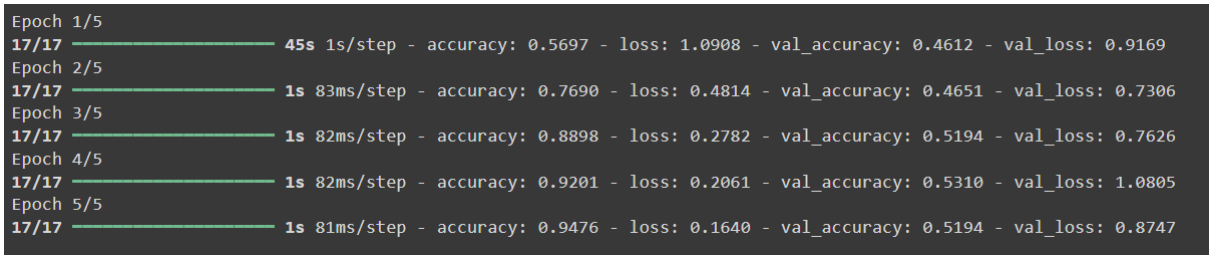


Figure D8

Training Results for ResNet50 with SGD and batch size of 64 and 10 epochs

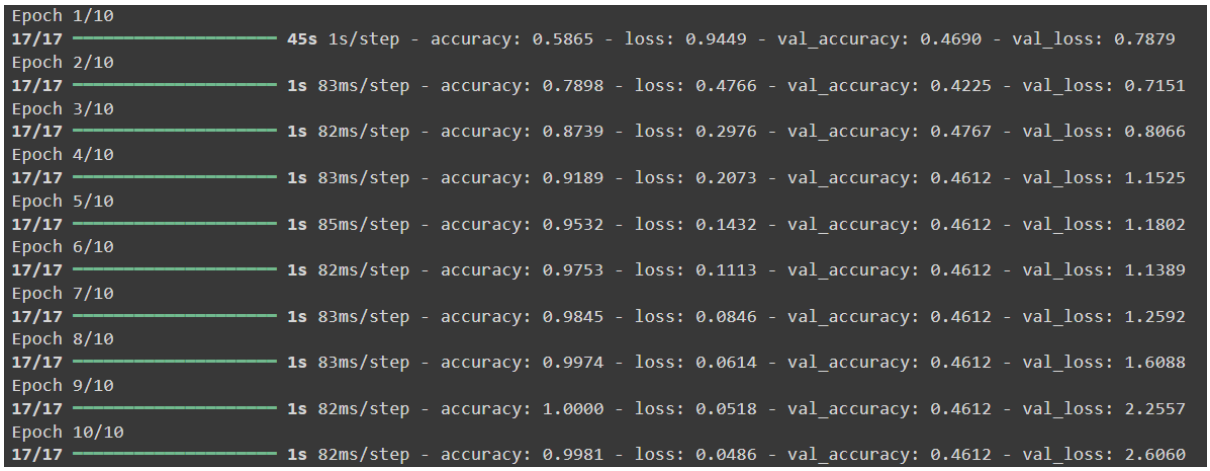


Figure D9

Training Results for ResNet50 with SGD and batch size of 64 and 25 epochs

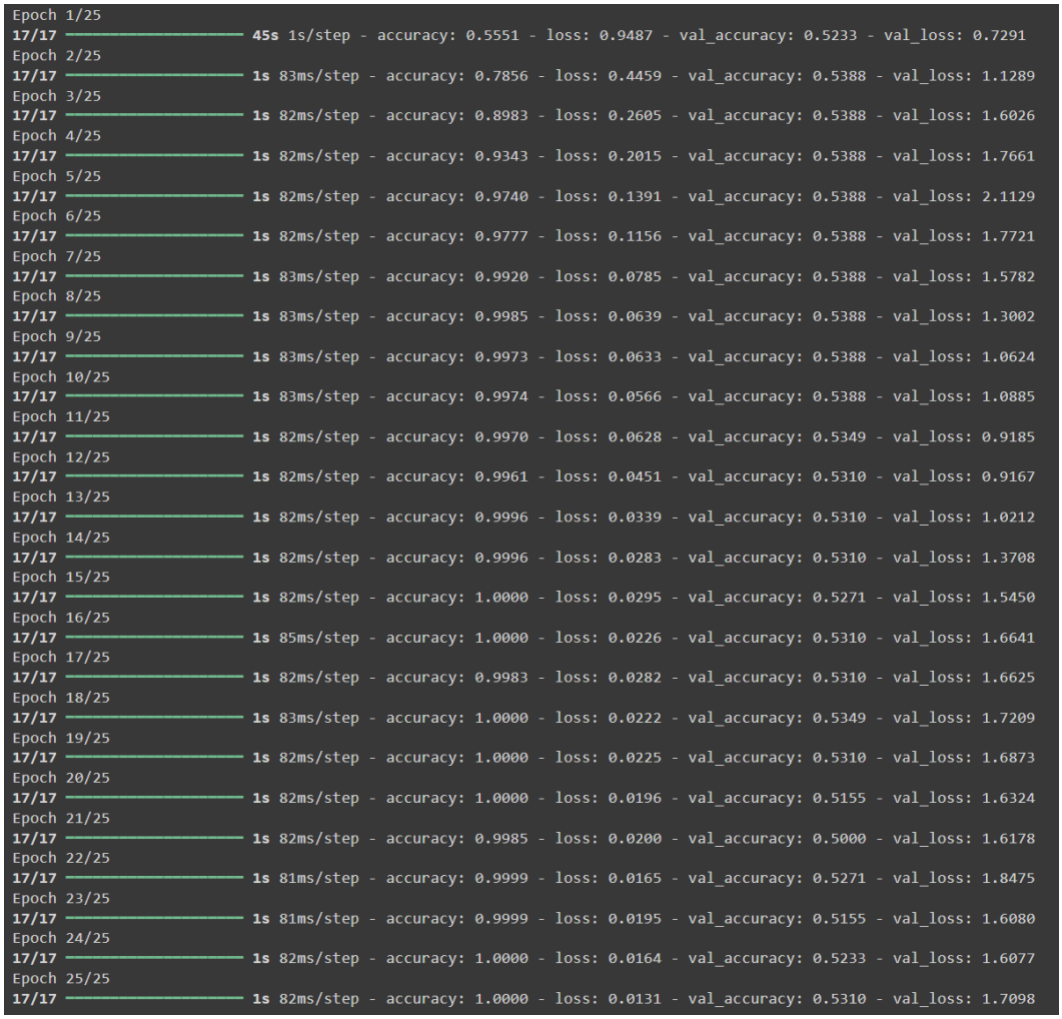


Figure D10

Training Results for ResNet50 with RMSprop and batch size of 16 and 5 epochs

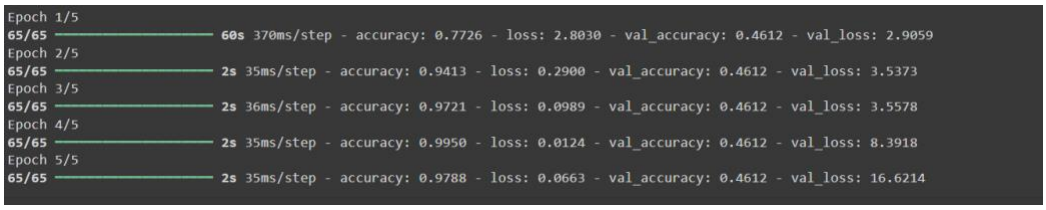


Figure D11

Training Results for ResNet50 with RMSprop and batch size of 16 and 10 epochs

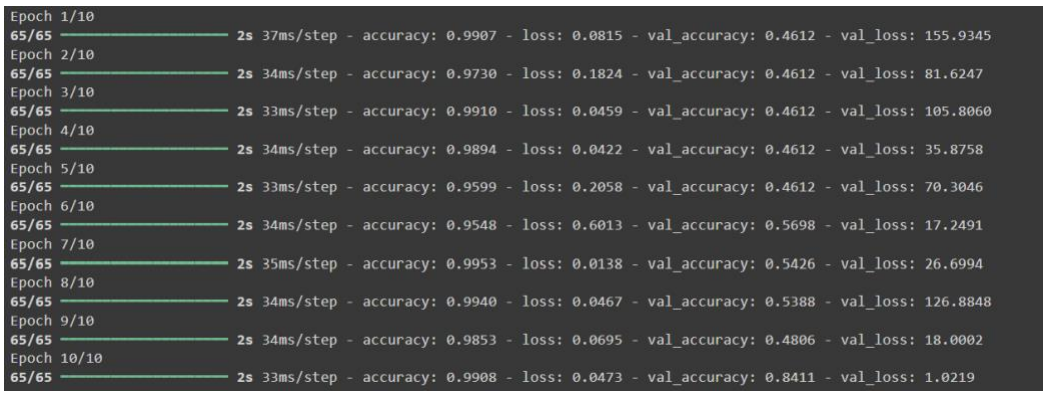


Figure D12

Training Results for ResNet50 with RMSprop and batch size of 16 and 25 epochs

Figure D13

Training Results for ResNet50 with RMSprop and batch size of 32 and 5 epochs

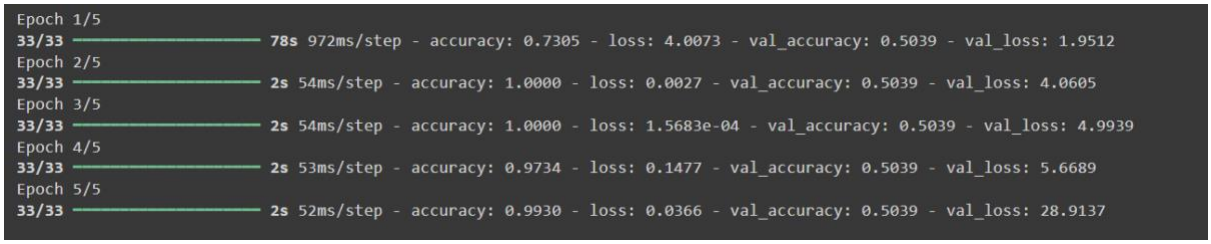


Figure D14

Training Results for ResNet50 with RMSprop and batch size of 32 and 10 epochs

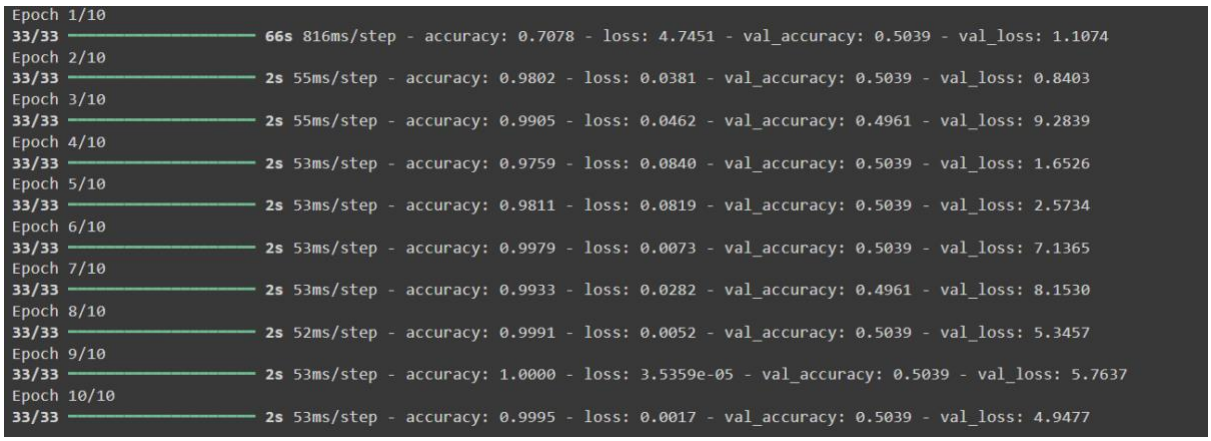


Figure D15

Training Results for ResNet50 with RMSprop and batch size of 32 and 25 epochs

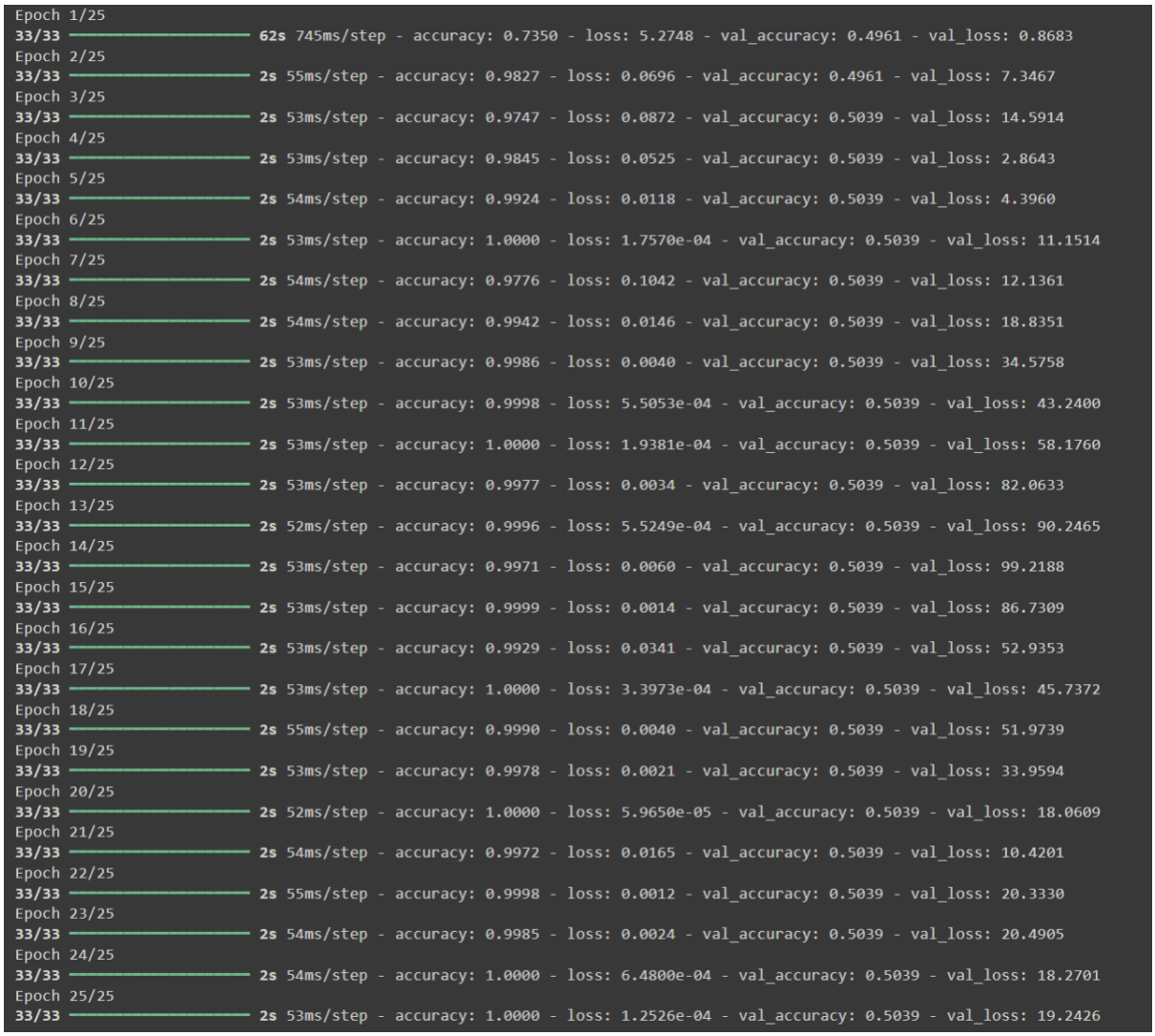


Figure D16

Training Results for ResNet50 with RMSprop and batch size of 64 and 5 epochs

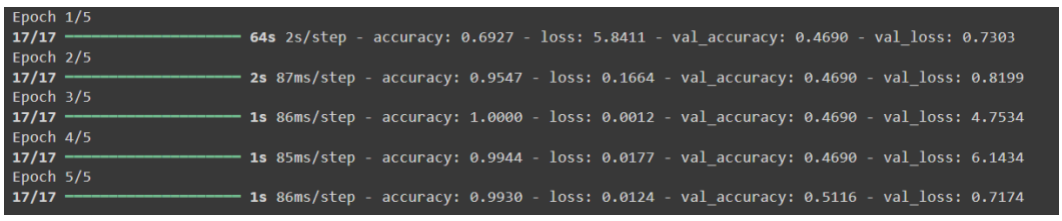


Figure D17

Training Results for ResNet50 with RMSprop and batch size of 64 and 10 epochs

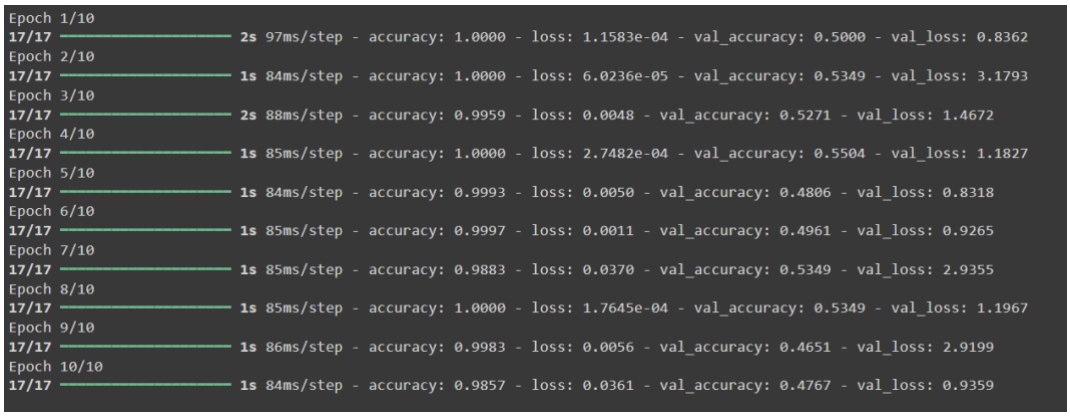
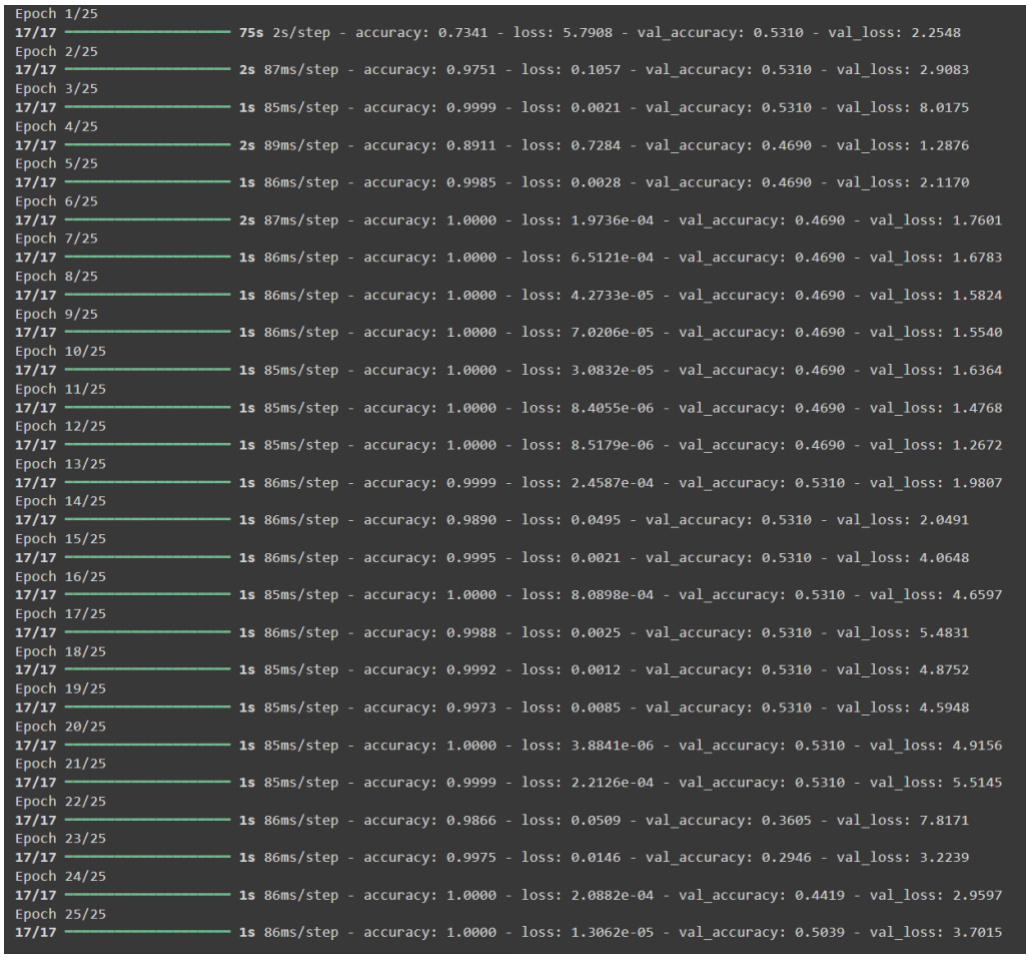


Figure D18

Training Results for ResNet50 with RMSprop and batch size of 64 and 25 epochs



Appendix E

This appendix shows training results for VGG16 across all configurations of parameters.

Figure E1

Training Results for VGG16 with SGD and batch size 16 and 5 epochs

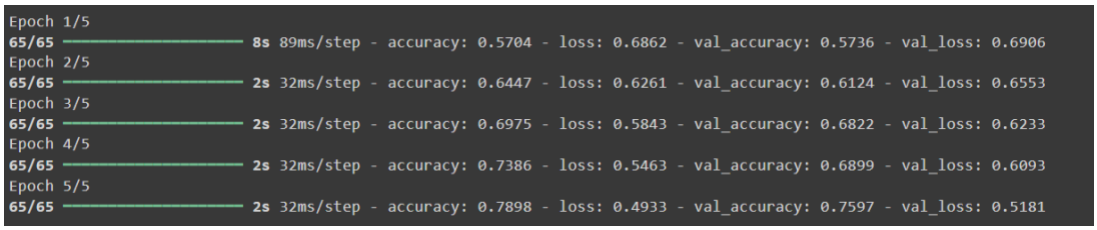


Figure E2

Training Results for VGG16 with SGD and batch size 16 and 10 epochs

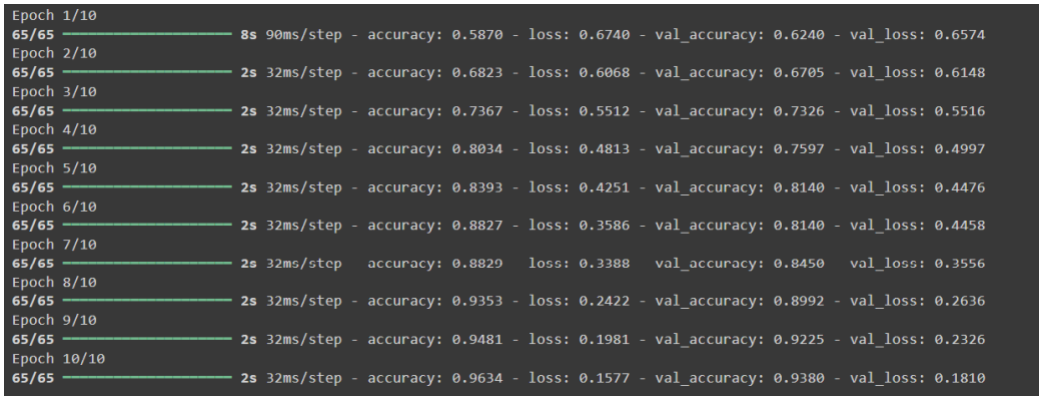


Figure E3

Training Results for VGG16 with SGD and batch size 16 and 25 epochs

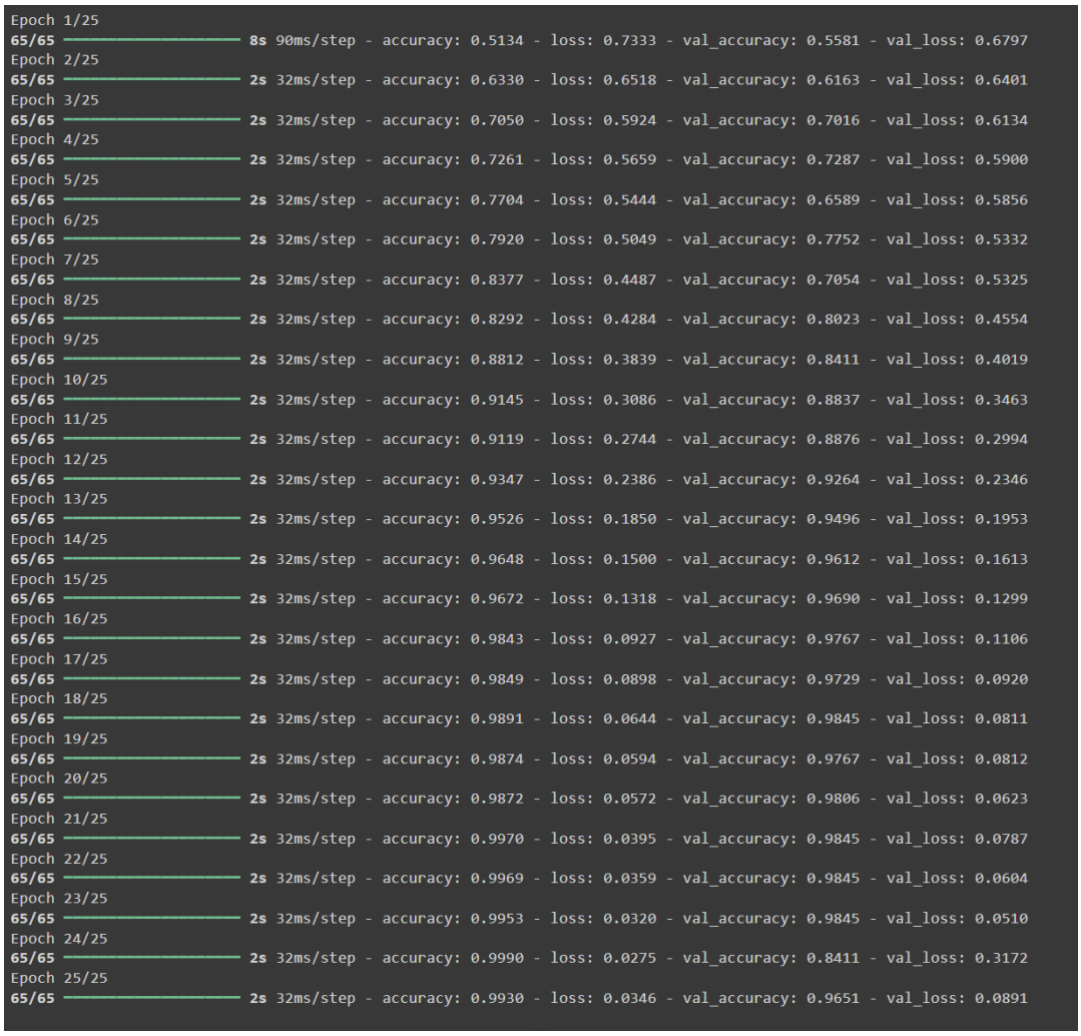


Figure E4

Training Results for VGG16 with SGD and batch size 32 and 5 epochs

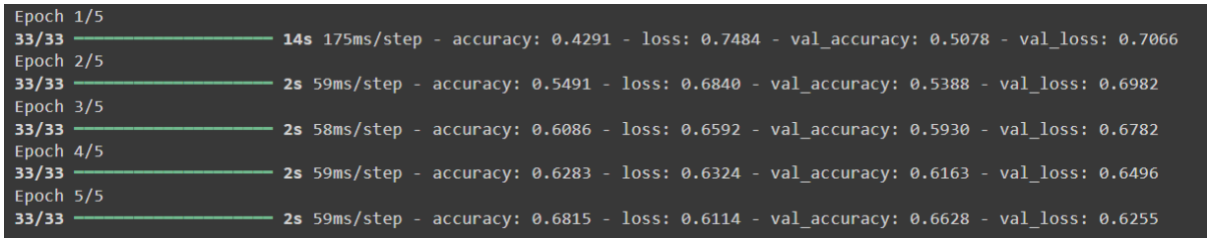


Figure E5

Training Results for VGG16 with SGD and batch size 32 and 10 epochs

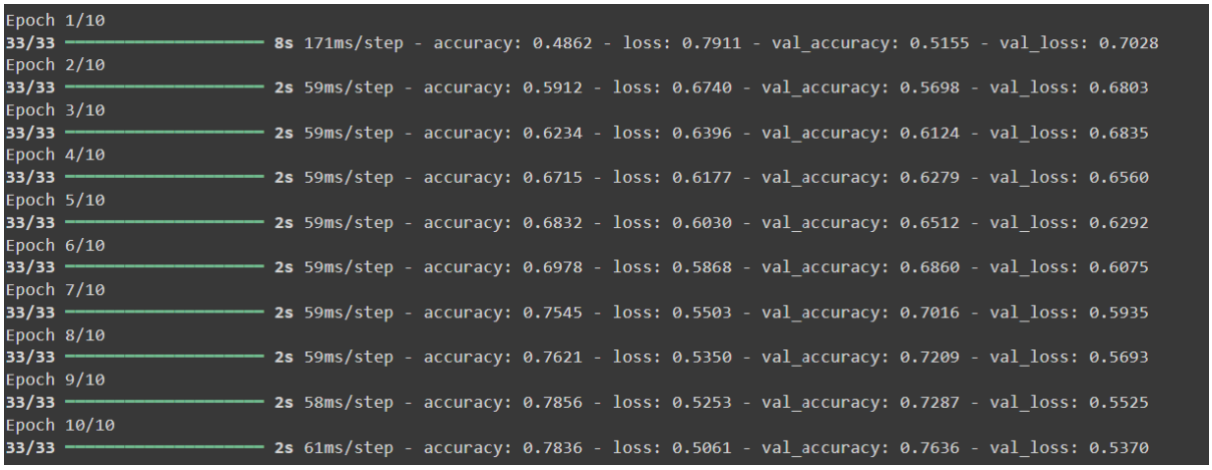


Figure E6

Training Results for VGG16 with SGD and batch size 32 and 25 epochs

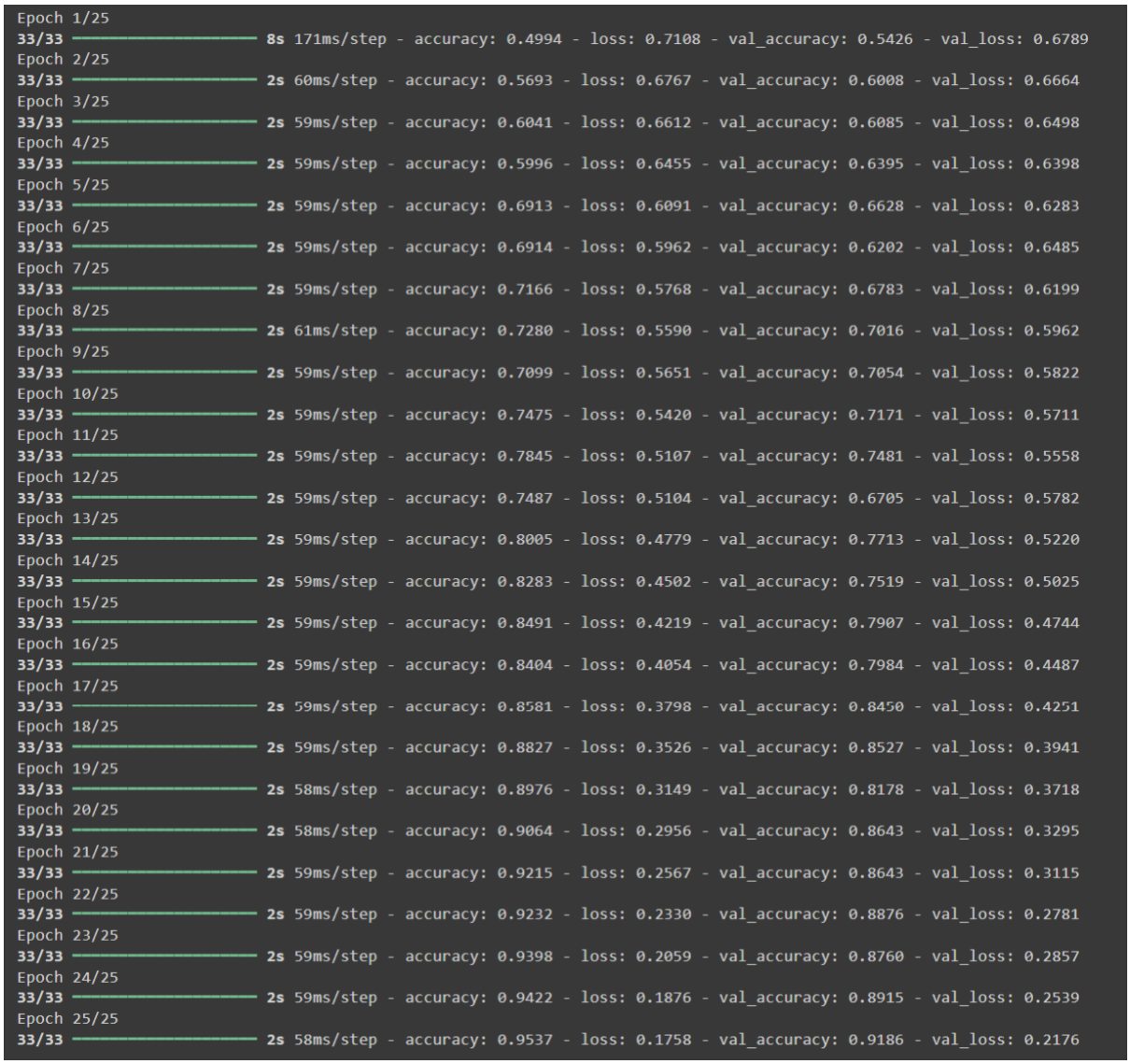


Figure E7

Training Results for VGG16 with SGD and batch size 64 and 5 epochs

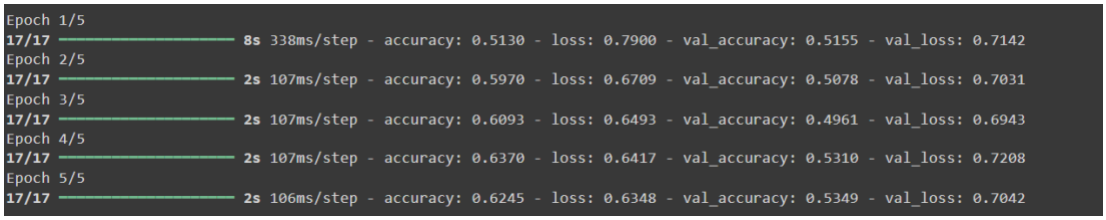


Figure E8

Training Results for VGG16 with SGD and batch size 64 and 10 epochs

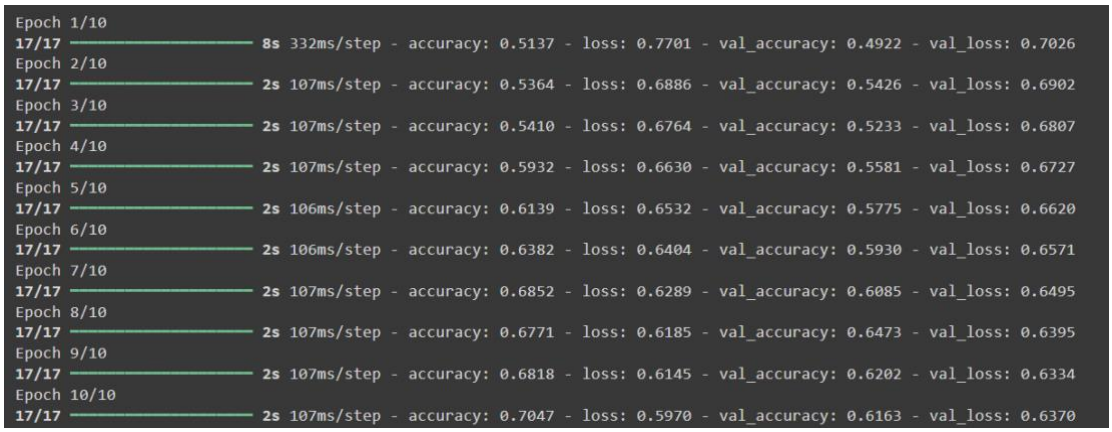


Figure E9

Training Results for VGG16 with SGD and batch size 64 and 25 epochs

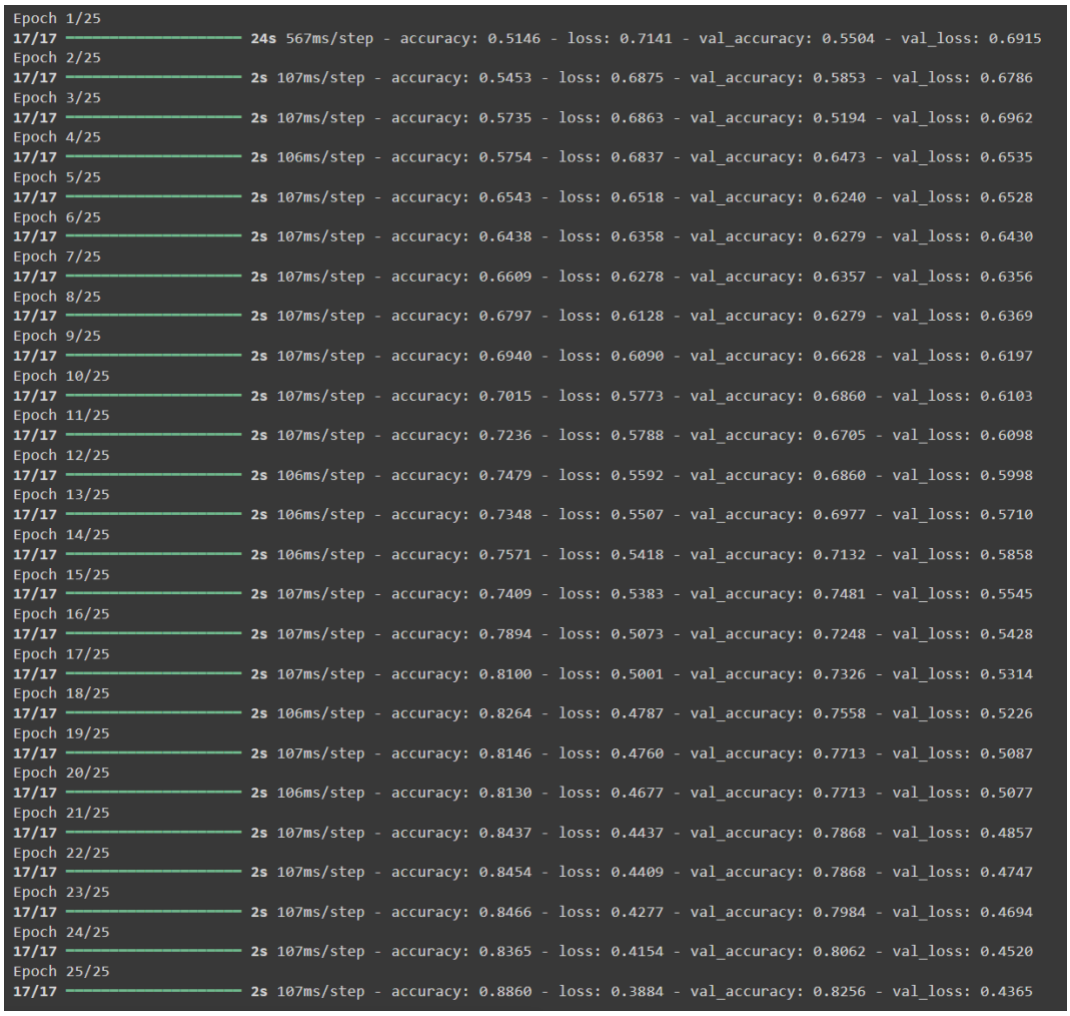


Figure E10

Training Results for VGG16 with RMSprop and batch size 16 and 5 epochs

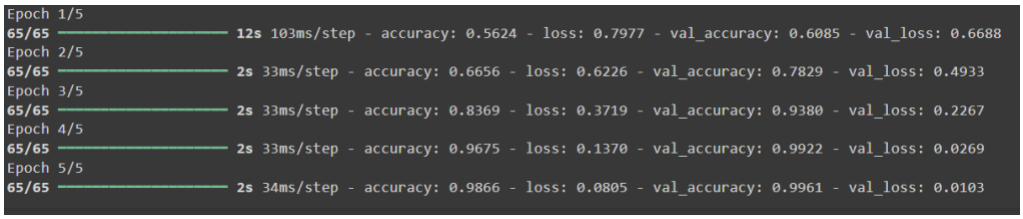


Figure E11

Training Results for VGG16 with RMSprop and batch size 16 and 10 epochs

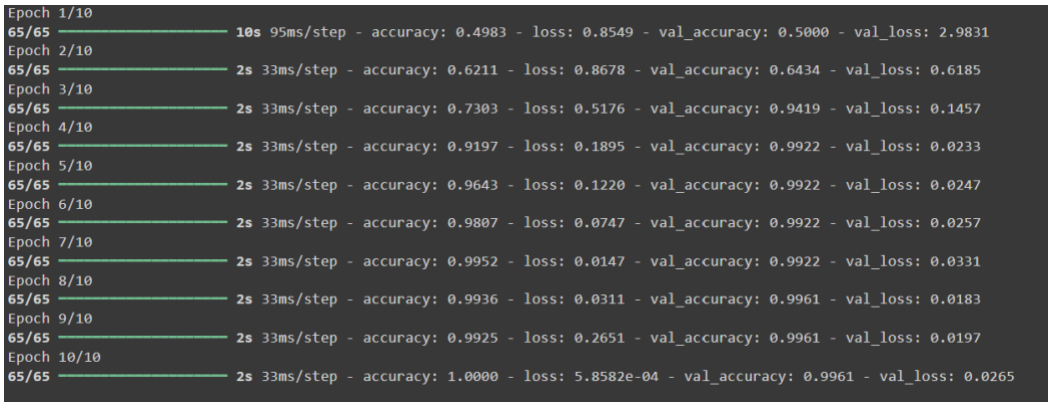


Figure E12

Training Results for VGG16 with RMSprop and batch size 16 and 25 epochs

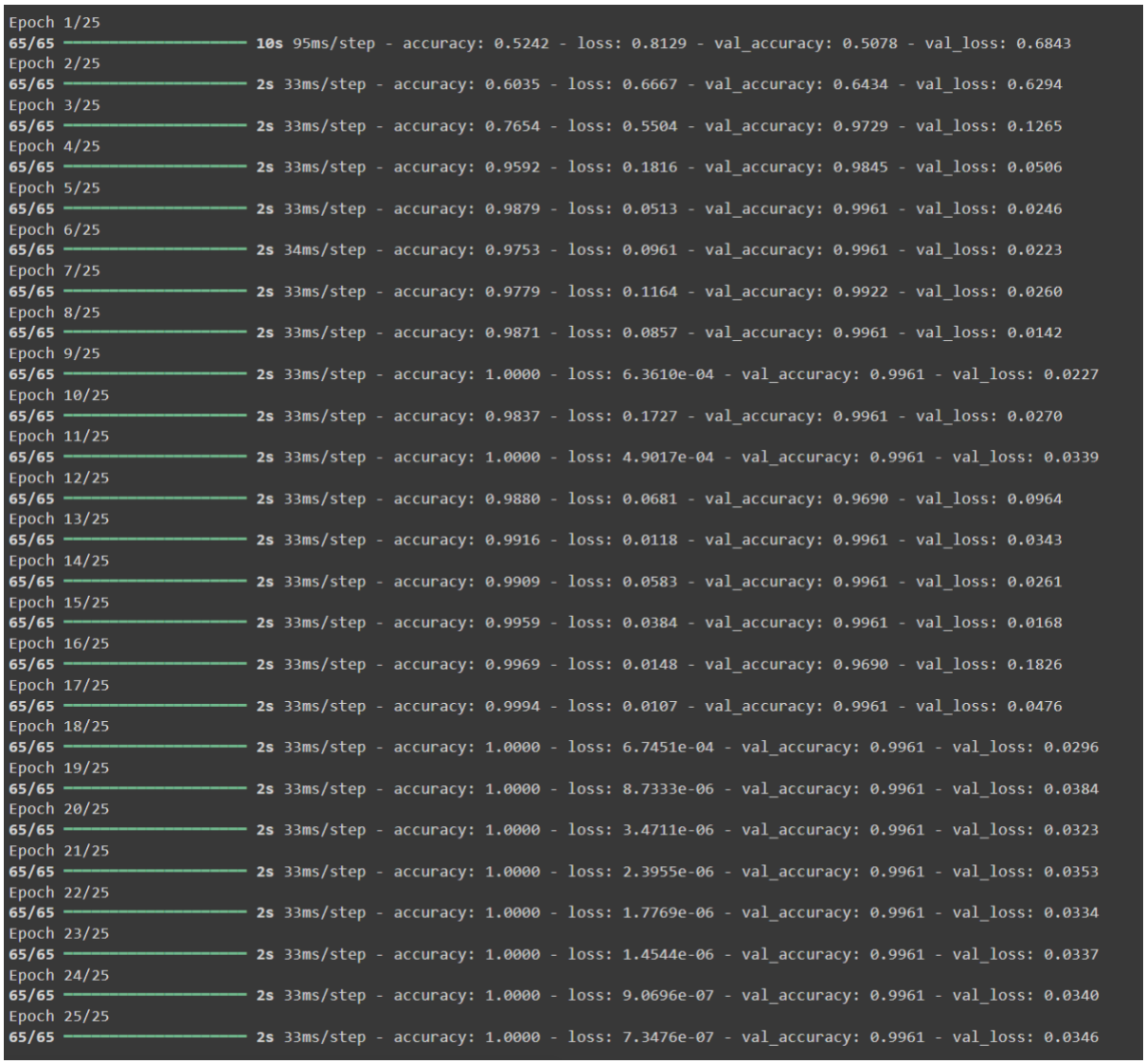


Figure E13

Training Results for VGG16 with RMSprop and batch size 32 and 5 epochs

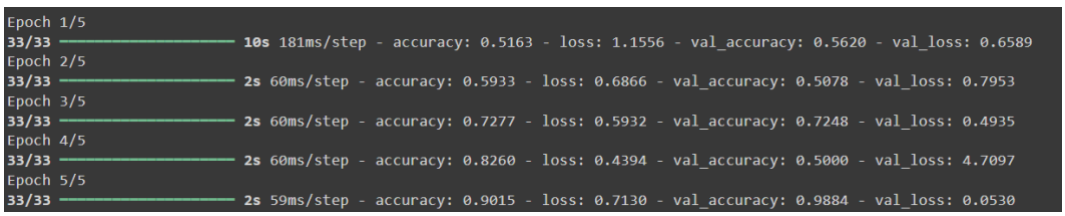


Figure E14

Training Results for VGG16 with RMSprop and batch size 32 and 10 epochs

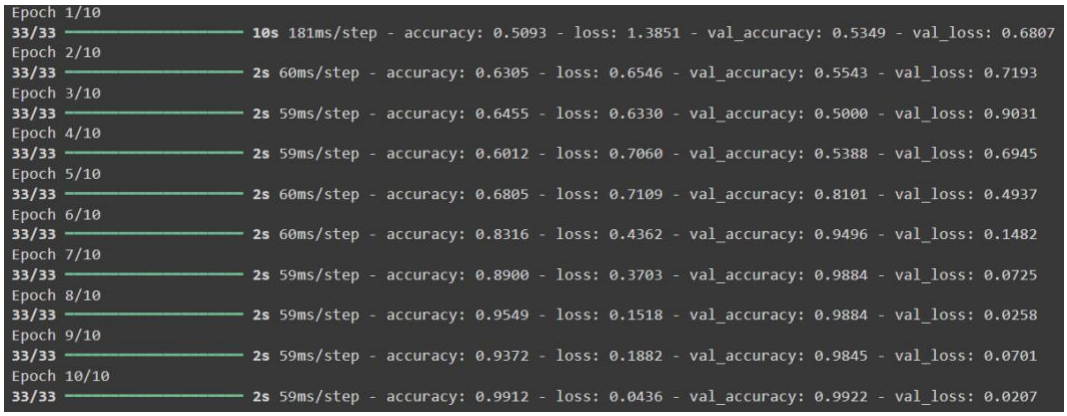


Figure E15

Training Results for VGG16 with RMSprop and batch size 32 and 25 epochs

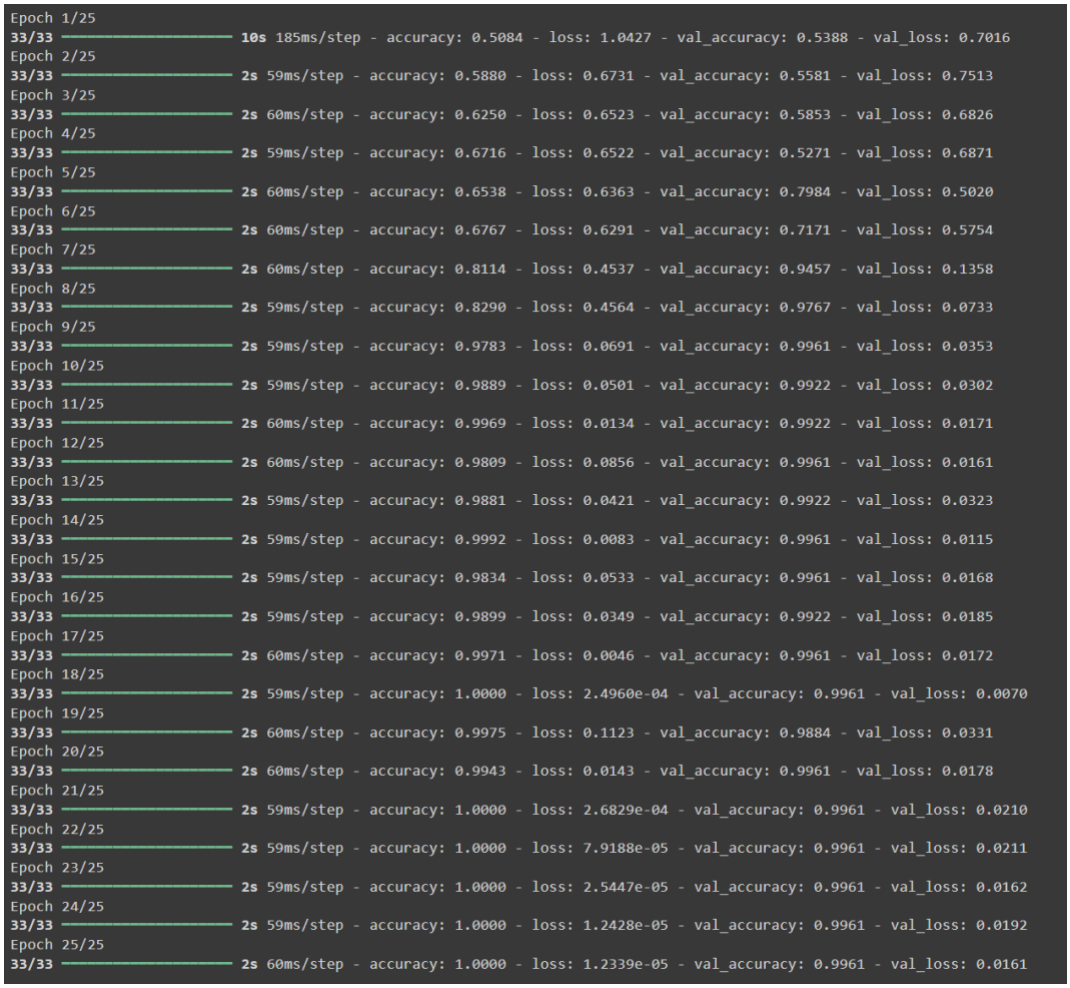


Figure E16

Training Results for VGG16 with RMSprop and batch size 64 and 5 epochs

Figure E17

Training Results for VGG16 with RMSprop and batch size 64 and 10 epochs

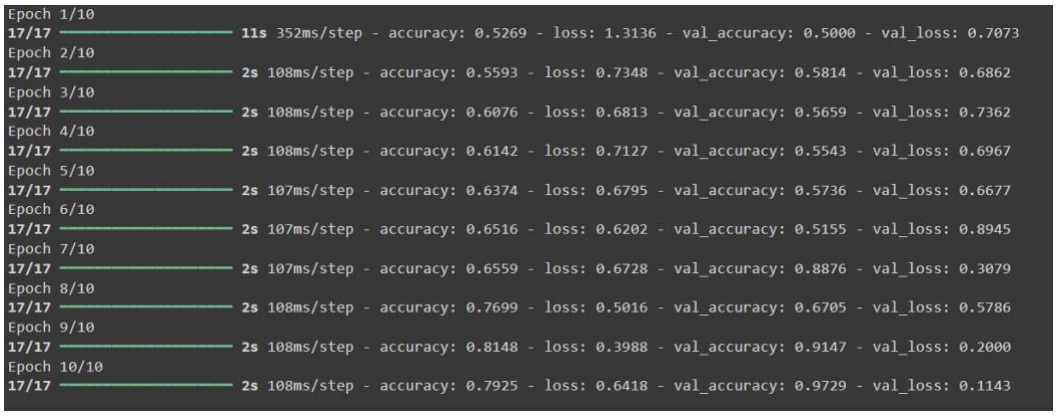


Figure E18

Training Results for VGG16 with RMSprop and batch size 64 and 25 epochs