COMP3330 — Deep Learning Project

Alexander Budden[1 [3354767]], Jaydon Cameron[2 [3329145]], Panhanith Sokha[3 [3347676]], Sebastian Hadley[4 [3349742]], Thomas Bandy[5 [3374048]]

University of Newcastle, Callaghan, NSW 2308, Australia

Abstract. This report elucidates the development and optimisation of machine learning models for two distinct classification tasks: environment type classification from images, and tweet topic classification, as part of our Machine Intelligence course assignment. The first task employs an image dataset from Intel comprising six environment categories, for which various deep learning models were developed, experimented with, The second task utilises the TweetTopic dataset, requiring semantic understanding of tweet contents to associate them with one of the six topics. For this task, we developed experimented with using different Bag-of-Words and LSTM architectures, and optimised using Optuna, an open-source hyperparameter optimisation framework. We achieved greater performance by fine-tuning pre-trained models. The report details our iterative model development process, the influences of different hyperparameters, as well as the outcomes of our model performances.

Keywords: Classification · NLP · ANN

1 Introduction

1.1 Problem Overview

This report presents our exploration and experimentation process in solving two distinct classification tasks as part of our Machine Intelligence course assignment. The tasks involve classifying images based on their environment type and classifying tweets based on their topics. The primary objective of the assignment is to develop effective models and find suitable hyperparameters to achieve high classification accuracy on the test datasets for both tasks.

Classifying Images by the Environment Type The first task involves the classification of images based on their environment type. The goal is to develop a model that can accurately distinguish between different types of environments such as buildings, forests, glaciers, mountains, seas, and streets. The specifics of the dataset used for this task and the pre-processing steps undertaken are detailed in **??**.

Classifying Tweets by Topic The second task requires the classification of tweet contents into one of six topic classes. This involves developing a model capable of understanding the semantic content of the tweets and associating them with the appropriate topics. As with the first task, the goal is to achieve high classification accuracy and understand the impact of various hyperparameters on model performance. The specifics of the dataset used for this task and the pre-processing steps undertaken are detailed in **??**.

1.2 Optuna

Optuna [1] is an open-source hyperparameter optimisation framework [1] designed to automate the trial-and-error process of searching for the optimal hyperparameters for a given machine learning model [3]. It uses a tree-structured Parzen estimator (TPE) as the default algorithm to guide its search for the optimal hyperparameters. In this report, Optuna is utilised to tune and optimise the neural network models; Optuna is a suitable choice for the tasks in this report, where the objective is to not only achieve high accuracy but also to understand the impact of different hyperparameters on the model performance.

2 Intel Image Classification

2.1 ResNet

ResNet or Residual Network is an artificial neural network (ANN) that was introduced to address the ”Vanishing/Exploding Gradient” problem. It allows ANNs with many layers to train more efficiently and accurately. This is achieved through ”residual connections” (also known as ”Skip Connections”), which are Neural Network (NN) connections that

allow layers of an ANN to be bypassed if they hurt the performance of the model. [4] [5]

In this project, ResNet18 will be used as it is the simplest variation and will be suitable for this task.

Hyper-parameters that were tested can be found in Table 10

Optimal hyper-parameters will be determined through testing the difference values (outlined above) for each hyper-parameter. The beginning values for this will be the lowest value detailed above of each hyper-parameter. After testing all the values of one hyper-parameter the next round will utilise the highest performing value of the previous test. Table 12 displays all of the results from all hyper parameters tested. Raw data with a learning graph can be found in Appendix B..

The model that had the best performance was with the following hyper-parameters -

- LR - 0.001
- Epochs - 50
- Width - 50
- AF - ReLU

This model achieved an accuracy of 83%.

Learn Rate and activation function were the hyper-parameters that showed the biggest change in accuracy. With a higher learn rate, the model's step size in the optimisation process is larger. This means that the model is more likely to overshoot the optimal point in the weight space where the error is minimised. Changing the activation function from ReLU to CELU or Sigmoid also yielded a negative change in accuracy, likely due to their lower efficiency when compared to ReLU.

2.2 AlexNet

AlexNet is a convolutional neural network (CNN) architecture that significantly influenced the field of computer vision in 2012. It was developed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton for the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC).

The AlexNet architecture consists of five convolutional layers, some of which are followed by max-pooling layers, and three fully connected layers. The network usually uses the ReLU activation function, which helps reduce the training time as compared to the CeLU and Sigmoid functions. The use of dropout layers in the fully connected sections help to prevent overfitting, while the images were transformed to expand the training dataset. In these trials width is not a hyper-parameter that is tested, as the Alexnet architecture is sensitive to width changes. Please note that batch normalisation was used to speed up training times. The ReLU will also be the only activation function tested as it has been shown to be the most suitable activiation function for the Alexnet architecture.

Table 40 displays all of the results from all hyper parameters tested. Raw data with a learning graph can be found in Appendix D..

Hyper-parameters that were tested can be found in Table 38

The model that had the best performance was with the following hyper-parameters -

- LR - 0.001
- Epochs - 5 and 15
- AF - ReLU

This model achieved an accuracy of 75%.

2.3 MobileNet

MobileNet is a neural network architecture that was introduced specifically to assist the low-end device that has limited computational resources and capabilities with running and training the image classification task effectively. This MobileNet was introduced by Google Team and currently has 3 separate versions which are MobileNetV1, MobileNetV2, and MobileNetV3 that have been released in 2017, 2018, and 2019 respectively. The difference between MobileNetV1 and MobileNetV2 is its unique layers. MobileNetV1 has a unique layer called the depthwise separable convolutions layer, which consists of a depthwise convolution and a point-wise convolution with the purpose of reducing the number of parameters and computation while maintaining the best result with high accuracy. MobileNetV2 also has this unique depthwise separable convolutions layer along with the upgrade feature known as Linear bottlenecks which is designed to reduce the usage of the computational resource while maintaining the result to archive high accuracy.

MobileNetV1 consists of 27 conventional layers including 13 depthwise layers along with 1 fully connected layer and 1 global average pooling layer. the following diagram indicated the MobileNet V1 layers.(See Figure 41 and Figure 42 in Appendix E.)

In this project, MobileNet V1 will be used as it is the simplest variation as well as suitable for devices with limited computation capabilities.

Hyper-parameters the following values were used to complete the image classification task -

- Learning Rate (LR) - 0.001
- Epochs (E) - 10
- Loss Function - Cross-Entropy
- Activation Function (AF) - ReLU
- Optimizer - Adam Optimizer

After completing the image classification training and validating process of 10 epochs, the result has been obtained along with a satisfactory validation accuracy of above 78 percent. The following image indicated the result of the image classification using MobileNet V1.(See Figure 43 and Figure 44 in Appendix E.)

2.4 VGGNet

VGGNet or known as Visual Geometry Group is a deep Conventional Neural Network architecture with multiple layers. The VGG architecture is the basis of object recognition models. For this Question, We used the VGG11 model which supports 11 layers, of which 8 are conventional layers and 3 are fully connected layers.

Hyper-parameters modified were for the following values-

- Learning rate (lr) - 0.1, 0.01
- Epoch(E) - 10
- Loss Function - Cross-Entropy
- Activation Function (AF) - ReLU
- Optimizer - Adam Optimizer

After completing the training and validating process of 10 epochs it got an accuracy of 17%

2.5 LeNet

LeNet, also known as LeNet-5, is a conventional neural network (CNN) architecture developed by Yann LeCun et al. in 1998 for handwritten digit recognition. It was one of the earliest successful applications of deep learning. LeNet consists of seven layers, including two conventional layers, two sub sampling layers, and three fully connected layers. The conventional layers use small filters of size 5x5 and the sub sampling layers use filters of size 2x2.

Hyper-parameters modified were for the following values-

- Learning rate (lr) - 0.1, 0.01
- Epoch(E) - 10
- Loss Function - Cross-Entropy
- Activation Function (AF) - ReLU
- Optimizer - Adam Optimizer

After completing the training and validating process of 10 epochs it got an accuracy of 70%

2.6 Inference Script

The Resnet18 model yeilded the highest accuracy with 83%. this model was trained and then saved to be run with the inference script.

3 Tweet Topic Classification

This section outlines the experimentation process for classifying tweets into 1 of 6 categories using the TweetTopic dataset [2] (see **??** for an overview of the dataset). The experimentation process involves testing various model architectures, with the aim of exploring different model architectures to handle the challenges outlined in section 1.1 and identifying the best model and hyperparameters to achieve a target accuracy of 70% when evaluated on the test set.

Several base classes have been developed to facilitate the experimentation process. The `BaseModel` class is a generic base class for the various model implementations, providing basic functions such as moving tensors to the specified device; it is the foundation for implementing specific model architectures. The `BaseTrainer` class is a generic base class for the various model training implementations, and the `BaseOptimiser` class provides the foundation and the starting point for executing the Optuna studies.

The following subsections discuss the implementation of the various model architectures and their performances, including the process of data pre-processing and hyperparameter tuning.

3.1 Experimenting with an LSTM Model

An LSTM (Long Short-Term Memory) model is a type of sequence model and recurrent neural network (RNN) that is capable of learning long-term dependencies in sequences of input data. This makes LSTM a good choice for processing and predicting data where order is important, like sentences in an NLP task. Traditional RNNs suffer from the vanishing gradient, where the network ‘forgets’ information from earlier in the input sequence. The LSTM model maintains two states to overcome this problem: a hidden state and a cell state. The hidden state maintains the time-step data, carrying information from the previous time-step. The cell state represents the current state of the LSTM cell, and carries data important for context over the duration of the model’s life. The LSTM cell passes time-step data through gates that essentially determine whether to maintain or discard the information as needed [7].

The LSTM cell consists of three main components [9]:

- **Forget Gate**: This decides what information should be thrown away or kept. Input from the previous hidden state and the current input is passed through this gate. It uses a sigmoid function, which outputs a value between 0 and 1. A value close to 0 means ’forget’, and a value close to 1 means ’keep’.
- **Input Gate**: This updates the cell state with new information. It has two parts: a sigmoid layer called the ”input gate layer” which decides which values to update, and a tanh layer which creates new candidate values that could be added to the state.

– **Output Gate**:This decides the next hidden state (output). The output will be based on our cell state, but will be a filtered version. It first applies a sigmoid function to decide which parts of the cell state make it to the output, then puts the cell state through tanh (to push values to be between -1 and 1) and multiplies it by the output of the sigmoid gate.

Pre-processing the Data A various selection of text standardisation techniques were used to pre-process the tweets. First, the text data is converted to lowercase to ensure consistency and reduce the dimensionality of the data. Then punctuation and stop words were removed, as these do not usually contain useful information for classifying the content of the text data. Next, the text is tokenised by splitting each tweet into individual words. Listing 1 shows the code that was used to perform the standardisation process.

Following the text standardisation process, the tokenised tweets need to be converted to a format that can be interpreted by the LSTM model. To achieve this, a vocabulary of all the unique words in the dataset was created and used to assign unique numerical values to each word and the numerical values were used to represent the tokenised data. Listing 2 shows the method `create_vocab()` that is used to create the vocabulary using `torchtext.Vocab` [6]. Listing 4 shows the method `vocabularise_text()` that is used to convert the word tokens into a numerical form using the vocabulary that was created earlier using `create_vocab()`.

Implementing the LSTM Model The LSTM model is implemented in Python [8] as the `LSTMModel` class and extends `BaseModel`. The implementation of the LSTM model consists of three layers: the embedding layer, LSTM layer, and linear layer. The embedding layer serves as the first layer of the LSTM model; it is used to convert the numerical tokens into dense vectors of fixed size that can provide useful semantics to the LSTM layer, the next layer of the model. The LSTM layer is the primary component of this model, applying a multi-layered or single-layered LSTM to the input sequence [6]. Additionally, the model supports the use of a bidirectional LSTM, which means the LSTM can learn from the sequence data in both forward and backward directions. The output of the LSTM layer is passed to the linear layer, which is used to output the predictions of the overall model. The output size of this layer corresponds to the number of classes in our target data, which is six in this case.

The constructor of the `LSTMModel` class (the `__init__()` method) is designed to provide flexibility in the architecture and hyperparameters of the model. This is particularly useful for hyperparameter tuning, as it allows for key aspects of the model to be modified during the tuning process. The adjustable parameters are listed below.

– The `vocab_size` and `embedding_dim` parameters control the architecture of the embedding layer, determining the size of the vocabulary that our model can handle and the dimensionality of the embedding vectors, respectively.

– The `output_size` parameter sets the number of output features of the model, corresponding to the number of classes in our target data.

- The `hidden_size` and `n_layers` parameters govern the architecture of the LSTM layer. `hidden_size` sets the number of hidden units in the LSTM layer, while `n_layers` determines the number of LSTM layers in the model.
- The `pad_idx` parameter is used to specify the index of the padding token in the vocabulary, which helps the embedding layer handle variable-length sequences.
- The `dropout` parameter allows us to specify a dropout rate for the LSTM layer, which can help to prevent overfitting during training.
- The `bidirectional` parameter determines whether our LSTM layer is bidirectional or not, providing the model with the capability to learn from sequence data in both forward and backward directions.

Tuning and Optimising the Model Optuna was utilised for the hyperparameter tuning process of the LSTM model, where multiple studies were performed using a range of suggested values for the hyperparameters discussed in section 3.1. Initial studies were focused on understanding the factors that affect the performance of the LSTM model. The experimentation process originally only considered the impact of the Adam optimiser and transitioned to consider other optimisers such as Adagrad. The overall exploration provided several key insights.

It was observed that smaller learning rates, often in the range of 1e-3 to 1e-2, were associated with high accuracy, suggesting a more steady convergence. The majority of high-performing trials used Adagrad as the optimiser, indicating the effectiveness of its adaptive learning rate. Trials with smaller batch sizes, especially 64, frequently outperformed others, implying a possible regularisation effect.

In terms of other parameters, the ExponentialLR scheduler, combined with smaller learning rates and Adagrad, seemed effective, as did dropout values around 0.35 - 0.45. Embedding dimensions of 200 and 300, as well as larger hidden sizes (256, 512, 1024), also appeared beneficial. Although most high-performing trials used fewer epochs (5, 10), a study with 50 epochs contradicted this trend by achieving a higher score.

The initial experiments were unable to achieve a validation accuracy greater than 50%. However, after modifying the minimum frequency for a word from the dataset to be included in the vocabulary from 5 to 1, the performance of the model saw a significant boost with the model reaching an accuracy of 81%.

3.2 Bag-of-Words Model

A BOW (Bag-Of-Words) model is a simplifying representation method that does not consider the sequence of the words within text and instead represents the text as a collection of the words within it. This model is constructed by creating a vocabulary from each unique word in the document and attaching each word within the document to an ID and then counting the occurrences of each word to store as a vector with the same dimensions as the vocabulary. Despite not capturing the order of words in a document, the BOW model is still useful for NLP tasks like topic classification and sentiment analysis as the frequency of the words within the document can indicate the class or sentiment.

Pre-processing the Data Simliarly to the LSTM Model the BOW Model begins the data pre-processing by standardising the text in the same way aswell as creating a vocabulary for use by the BOW Model, following this a process called numericalisation is performed that replaces the tokens within the text with it's corresponding index within the vocabulary. The numericalized data is then multi-hot encoded, this involves the data being transformed into a binary matrix representation in which each row represents a sample of text and each column represents a unique token in the vocabulary, with 1 indicating the occurrence of a token in the text and 0 being the absence. Listing 3 shows the code used to multi-hot encode the dataset.

Implementing the BOW Model The BOW Model is implemented in Python [8] with the `BOWModel` class which extends `BaseModel`. This implementation of the model contains 2 hidden layers, with the first performing the linear transformation on an input size equal to the size of the vocabulary, and an output size of 12, The second model has 12 input neurons with 16 output neurons and also uses the linear transformation, the RELU activation function is used in between layers to combat the vanishing gradients issue. The code used to implement the BOW Model used the same adjustable hyperparameters that were used for the LSTM Model.

Tuning and Optimising the Model The optimisation of the model used the same tools as the LSTM model to test 120 different combinations for the different hyperparameters and measure their impact on performance. This optimisation process allowed for a better understanding of the relationships between the hyper parameters and performance, and a few notable insights were identified, The details of each of the trials are saved to a csv allowing for further investigation.

The optimiser was seen to have the largest impact on the models success as seen in Figure 15. The Adam and RMSprop optimisation functions have a significantly higher average accuracy than the others, with the Adam being the most effective, details can be seen in Table 3. The learning rate had the second largest impact on the models performance with a correlation coefficient of 0.17, this indicates a light relationship between the learning rate and the accuracy of the model. The trials were able to achieve decent validation accuracy, however under performed in comparison to the LSTM Model, the highest test accuracy was 0.699, this model also achieved around a 0.9, details on this model can be seen in Figure 16 and Figure 17.

3.3 Test-Set Performance

The Top 3 Models of each type were evaluated with the test set to determine which architecture was more successful in developing an accurate model, the top 3 were used in case the random nature of seeding impacted the performance on the validation set.

Details about the configuration of the Models that were assessed are in this table Table 5, the third model in that table achieved the best results when evaluated on the test set, achieving an accuracy of 62.9%. This result is fairly good considering the relatively low complexity of the model only being made up of 2 hidden layers, however is not

amazing especially in comparison to the LSTM performance. This model also got a precision of 35.53% which is quite low indicating that this model is likely a poor fit for classifying this dataset despite having an acceptable accuracy, full results are shown in this Table 6

The LSTM Models assessed were configured as shown in Table 2, all three of these models performed significantly better then the BOW Models in every measure of performance that was assessed. The highest accuracy was 73.9%, the second model in the table achieved this result. The significant increase in performance over the BOW Model's is likely due to the bag of words not considering the order of the words within the text, this highlights a benefit of the LSTM for this specific classification task. The precision that this model achieved was 58.4%, this is significantly better then the highest from the BOW Models and further demonstrates the increase in performance, the full results for this model can be seen in Table 7

3.4 Fine-tuning A Pre-trained Model

Fine-tuning a pre-trained model involves initialising a model with the weights of a pre-trained model and then continuing training on a specific task, in this case, tweet topic classification. This process benefits from the general knowledge of language semantics that the model has learned during its pre-training, while further customising the model's knowledge to the specific task.

HuggingFace' Transformers [10] provides a variety of pre-trained models, including BERT, GPT-2, RoBERTa, and DistilBERT. These models have been trained on large corpora, allowing them to learn a wide array of language patterns and semantic representations. As part of this fine-tuning process, the pre-trained model 'cardiffnlp/twitter-robertabase-dec2021-tweet-topic-single-all'
[2] was used. This process was started by preprocessing the data using the same techniques discussed in section 3.1, using a limit of 200 tokens due to limitations in hardware. A series of trials were then performed using the
'cardiffnlp/twitter-roberta-base-dec2021-tweet-topic-single-all'
[2] pre-trained model to find the optimal hyperparameters for achieving the required 75% accuracy. Table 8 shows the achieved accuracy of the top performing hyperparameter combinations from evaluating the respective models on the validation set — analysing this table, it is clear that a smaller batch size and number of epochs is contributing to the high accuracy; the Adam optimiser would always perform well compared to other optimisers such as AdamW; and lowering the learning rate would improve the performance.

The performances of the different models were compared, and the best performing model was evaluated on the test set with great performance. As shown in Table 9, the model scored an accuracy of 90% on unseen data. The confusion matrix for this test is shown in **??**

Acknowledgements Please place your acknowledgements at the end of the paper, preceded by an unnumbered run-in heading (i.e. 3rd-level heading).

References

1. Akiba, T., Sano, S., Yanase, T., Ohta, T., Koyama, M.: Optuna: A Next-generation Hyperparameter Optimization Framework. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (2019)
2. Antypas, D., Ushio, A., Camacho-Collados, J., Neves, L., Silva, V., Barbieri, F.: Twitter Topic Classification. In: Proceedings of the 29th International Conference on Computational Linguistics. International Committee on Computational Linguistics, Gyeongju, Republic of Korea (2022)
3. Avhale, K.: Understanding of Optuna-A Machine Learning Hyperparameter Optimization Framework. (2021). https://medium.com/@kalyaniavhale7/understanding-of-optuna-a-machine-learning-hyperparameter-optimization-framework-ed31ebb335b9
4. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2016). https://doi.org/10.1109/cvpr.2016.90
5. Kashiwa, Implement resnet with pytorch, (2022). https://towardsdev.com/implement-resnet-with-pytorch-a9fb40a77448.
6. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: PyTorch: An Imperative Style, High-Performance Deep Learning Library. In: Advances in Neural Information Processing Systems 32, pp. 8024–8035. Curran Associates, Inc. (2019). http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf
7. T.J.J, R.: LSTMs Explained: A Complete, Technically Accurate, Conceptual Guide with Keras. Analytics Vidhya (2020). https://medium.com/analytics-vidhya/lstms-explained-a-complete-technically-accurate-conceptual-guide-with-keras-2a650327e8f2
8. Van Rossum, G., Drake, F.L.: Python 3 Reference Manual. CreateSpace, Scotts Valley, CA (2009)
9. Wikipedia contributors, Long short-term memory — Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=Long_short-term_memory&oldid=1148032239 (2023). [Online; accessed 13-May-2023].
10. Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Scao, T.L., Gugger, S., Drame, M., Lhoest, Q., Rush, A.M.: HuggingFace's Transformers: State-of-the-art Natural Language Processing, (2020). arXiv: 1910.03771 [cs.CL].

Appendices

Appendix A. Experimentation Results - TweetTopic

Table 1: The best performing LSTM models on the validation set.

Acc (%)	Dir	Batch	Drop	Embed	Epoch	Hidden	LR	Sched	Optim	Layers
80.07	Bi	128	0	500	10	256	5.83e-3	CosineAnnealingLR	Adam	1
80.07	Bi	64	1.06e-1	300	5	1024	2.94e-3	ExponentialLR	Adam	1
80.34	Bi	64	0	400	10	256	2.75e-3	ReduceLROnPlateau	Adam	1
79.45	Bi	64	0	500	10	256	3.61e-3	ReduceLROnPlateau	Adam	1
79.04	Bi	64	0	300	5	1024	1.29e-3	ExponentialLR	Adam	1
78.91	Bi	64	0	300	5	1024	4.32e-3	ExponentialLR	Adam	1

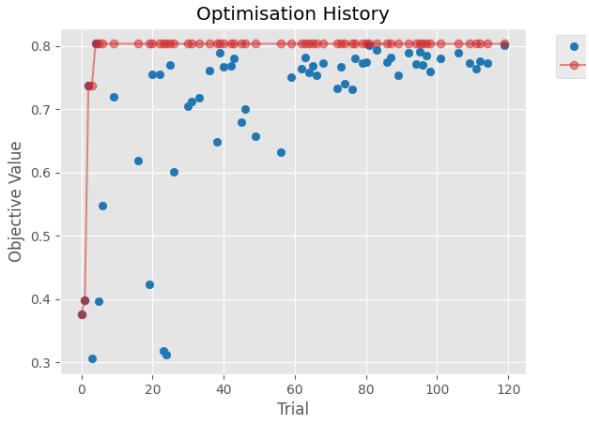


Fig. 1: The optimisation history for all of the trials performed for the LSTM model, where the 'Objective Value' represents the achieved accuracy of the model during the trial (blue=objective value, red=best value).

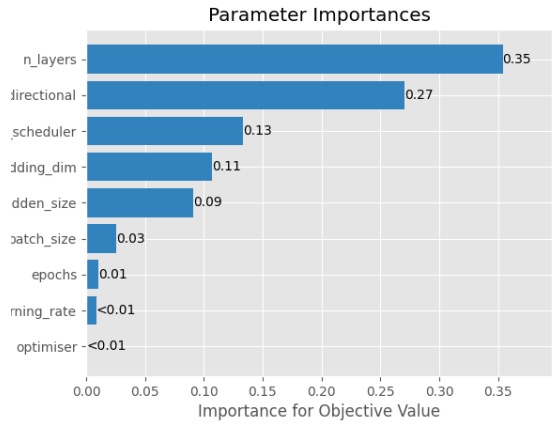


Fig. 2: The measured importance of each hyperparameter during the optimisation process of the LSTM model, where the 'Objective Value' is the accuracy of the model. The 'importance' of a hyperparameter indicates how the value of the hyperparameter impacted the performance of the model.

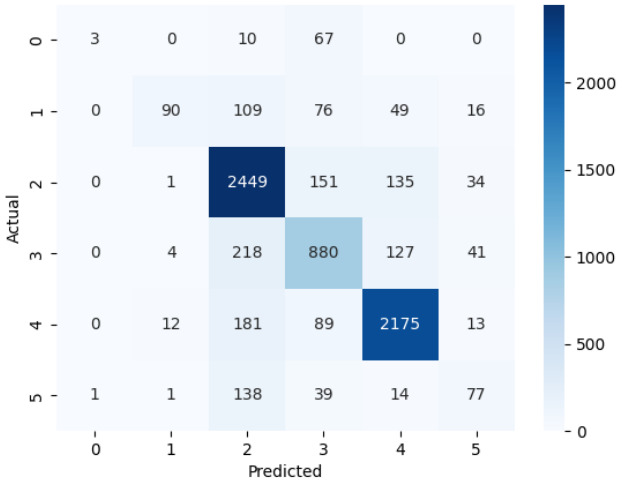


Fig. 3: Confusion matrix for the best performing LSTM architecture.

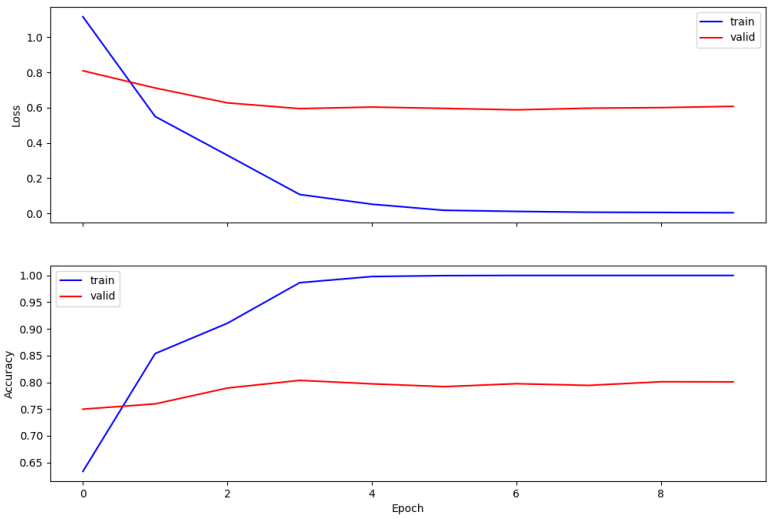


Fig. 4: The recorded loss and accuracy for the best performing LSTM architecture.

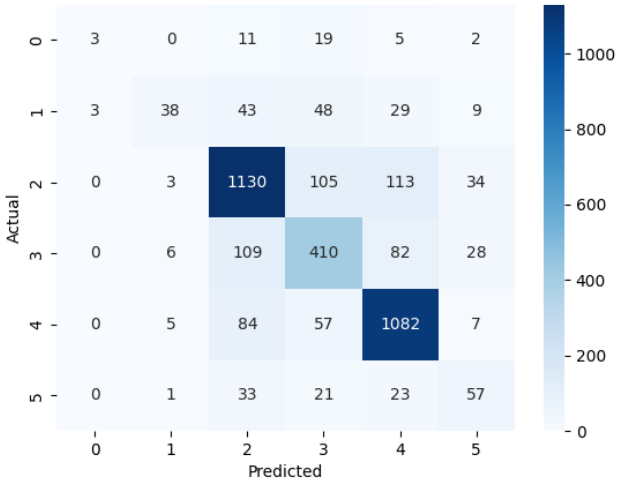


Fig. 5: The confusion matrix for the second best performing LSTM model.

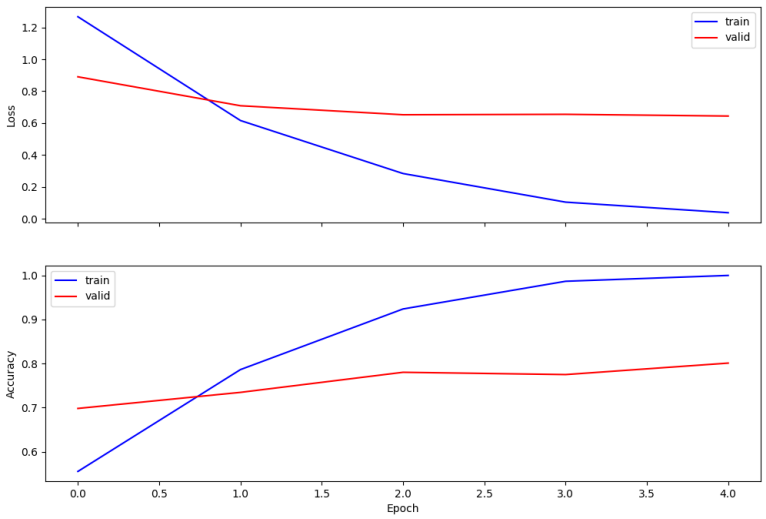


Fig. 6: Loss and accuracy curves for the the second best performing LSTM model.

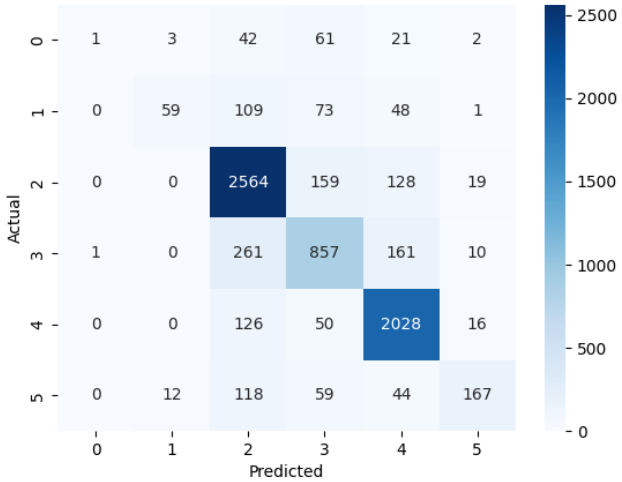


Fig. 7: The confusion matrix for the third best performing LSTM model.

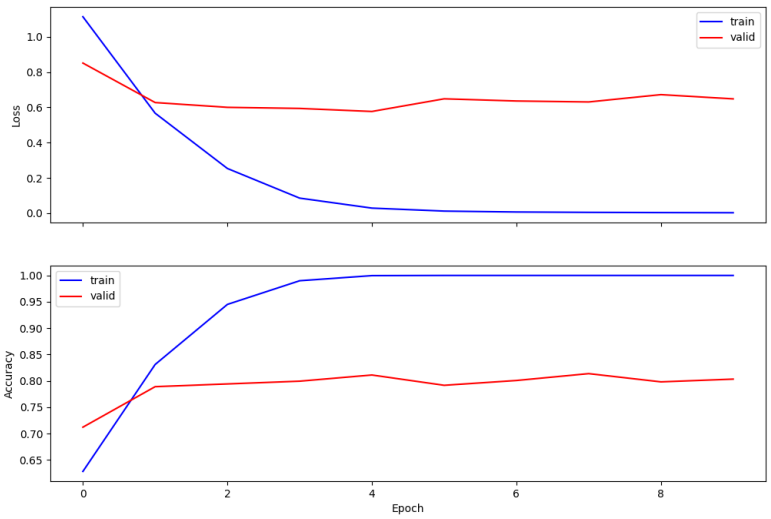


Fig. 8: The loss and accuracy curves for the third best performing LSTM model.

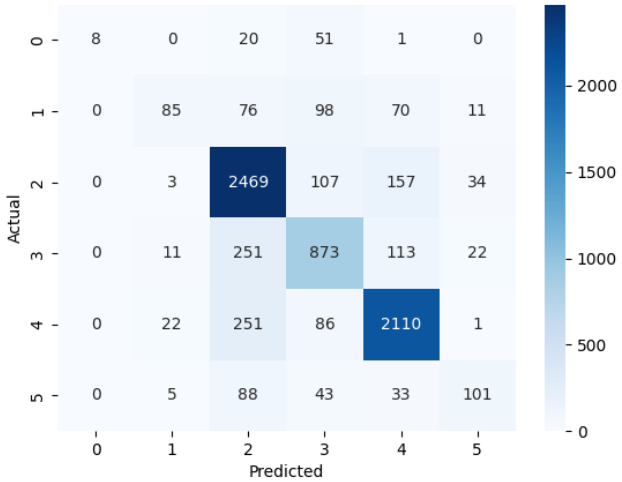


Fig. 9: The confusion matrix for the fourth best performing LSTM model.

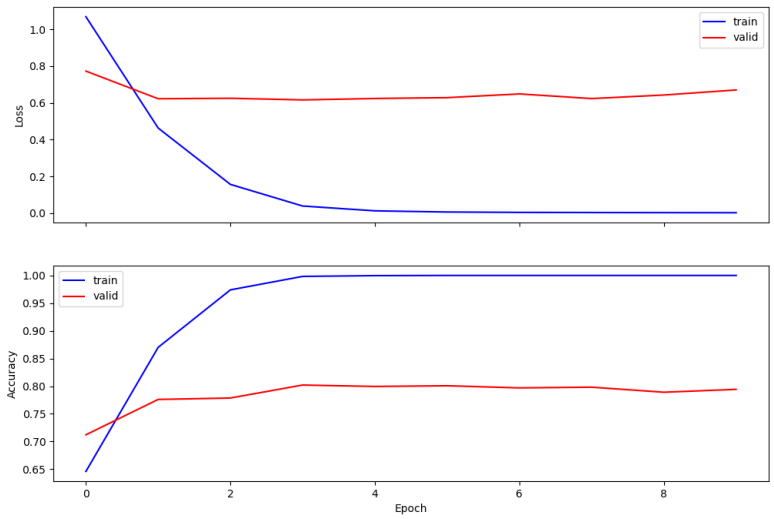


Fig. 10: The loss and accuracy curves for the fourth best performing LSTM model.

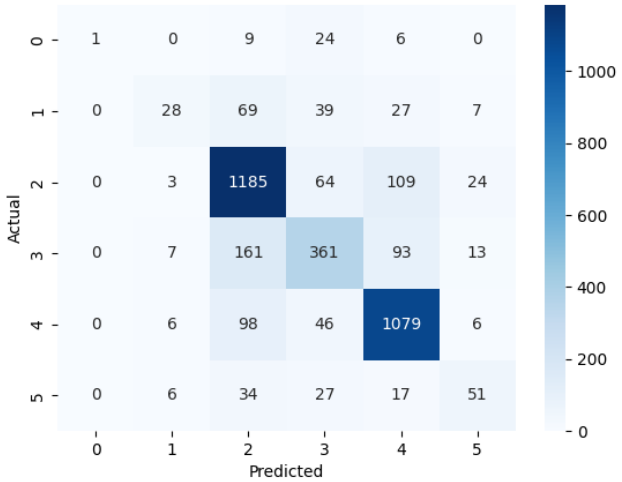


Fig. 11: The confusion matrix for the fifth best performing LSTM model.

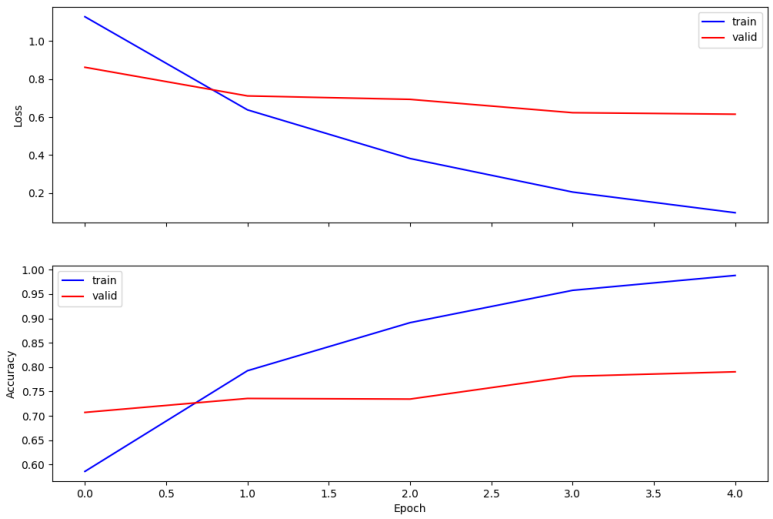


Fig. 12: The loss and accuracy curves for the fifth best performing LSTM model.

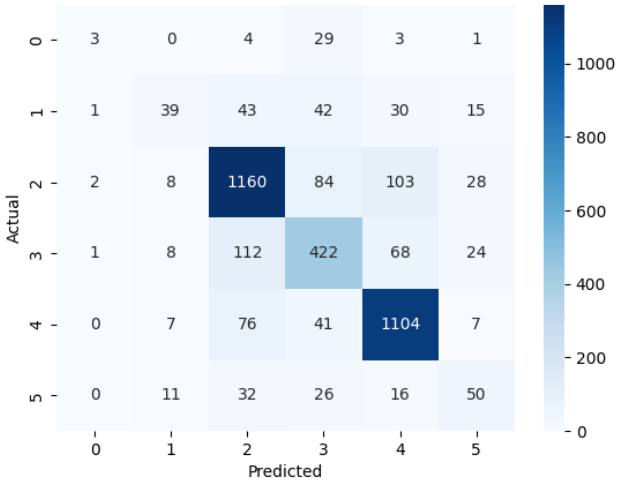


Fig. 13: The confusion matrix for the sixth best performing LSTM model.

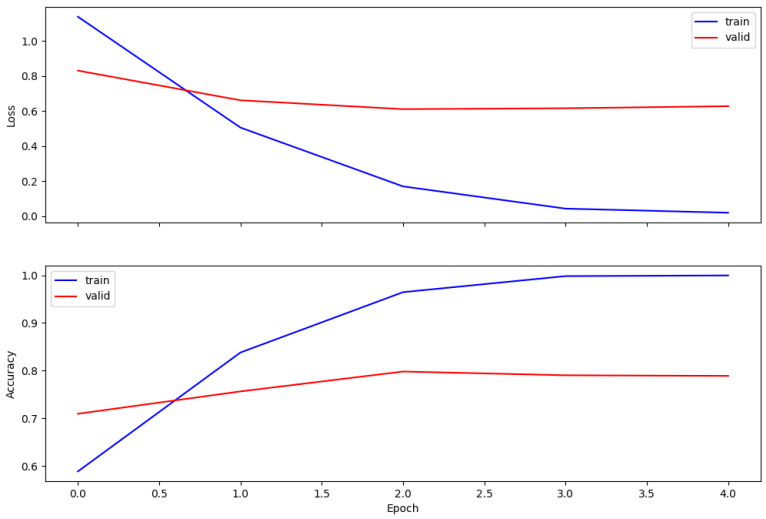


Fig. 14: The loss and accuracy curves for the sixth best performing LSTM model.

Table 2: Best LSTM Model Architectures

Learning Rate	Epochs	Batch Size	Optimizer	Scheduler	Hidden Neurons	Hidden Layers
1.49E-6	10	64	Adam	ReduceLROnPlateau	256	1
0.012527	5	64	Adam	ExponentialLR	1024	1
0.000798	10	8	Adam	CosineAnnealingLR	256	1

Optimizer	Average Accuracy
SGD	0.3507
Adam	0.6188
Adagrad	0.4816
RMSprop	0.6003

Table 3: Average Accuracy by Optimiser.

Table 4: Top 3 Models after Validation

Accuracy	Hyper-parameters				
	Batch Size	Epochs	Learning Rate	Max Tokens	Optimizer
0.70	8	5	0.000351	200	Adam
0.69	64	5	0.028545	600	Adam
0.69	8	20	0.000363	100	Adam

Table 5: Best BOW Model Architecture

Learning Rate	Epochs	Batch Size	Optimizer	Scheduler	Max Tokens	Hidden Neurons	Hidden Layers
0.000351	5	8	Adam	MultiStep	200	128	2
0.028545	5	64	Adam	StepLR	600	64	4
0.000363	20	8	Adam	Exponential	200	128	2

Table 6: The performance scores of the Best Bag Of Words Model on the test set.

Score	Value (%)
Accuracy	62.93
Precision	35.53
F1	31.53
Recall	33.84

Table 7: The performance scores of the best LSTM Model on the test set.

Score	Value (%)
Accuracy	73.9
Precision	59.4
F1	46.09
Recall	45.21

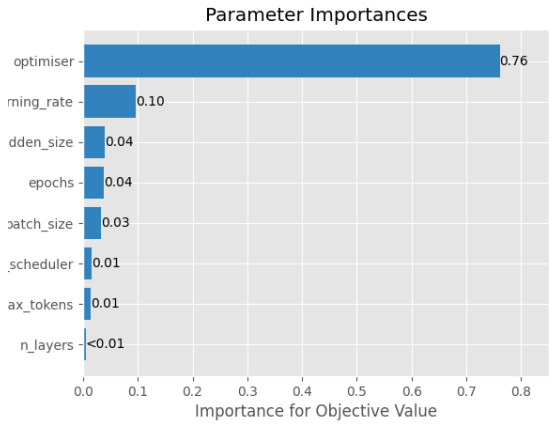


Fig. 15: The measured importance of each hyperparameter during the optimisation process of the BOW model, where the 'Objective Value' is the accuracy of the model. The 'importance' of a hyperparameter indicates how the value of the hyperparameter impacted the performance of the model.

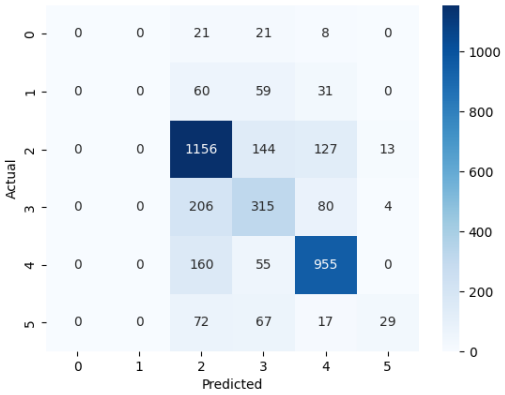
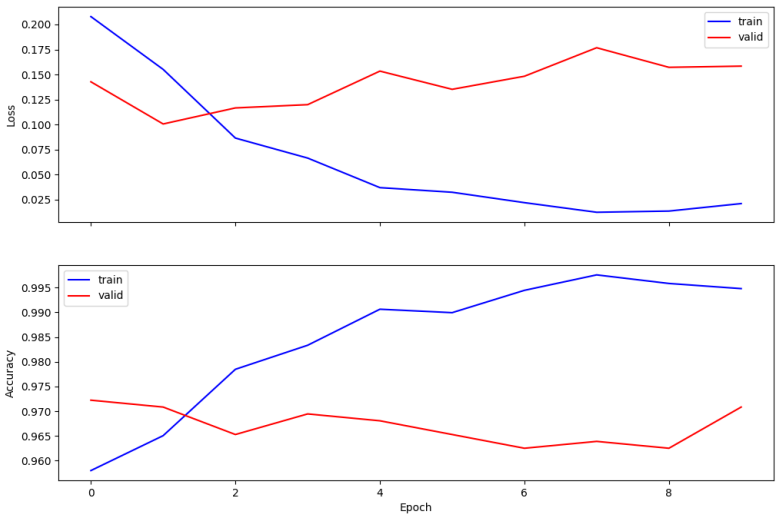


Fig. 16: Confusion matrix for the best performing BOW architecture.



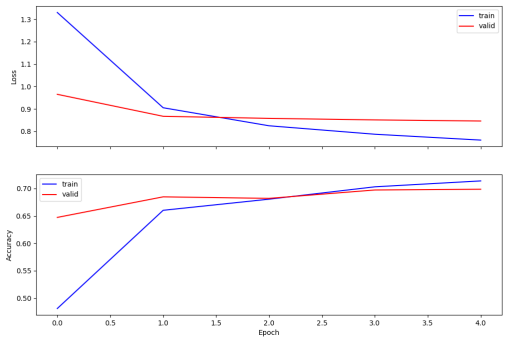


Fig. 17: Loss an accuracy curve of the best performing BOW model.

Table 8: Pre-trained model performances during the fine-tuning process.

Acc (%)	Batch	Epoch	LR	Optim	Sched
96.71	8	10	1.06e-5	Adam	Cosine
93.49	8	10	5.48e-5	Adam	Cosine
90.43	8	10	7.17e-5	Adam	Cosine
49.39	8	5	2.83e-3	AdamW	Cosine

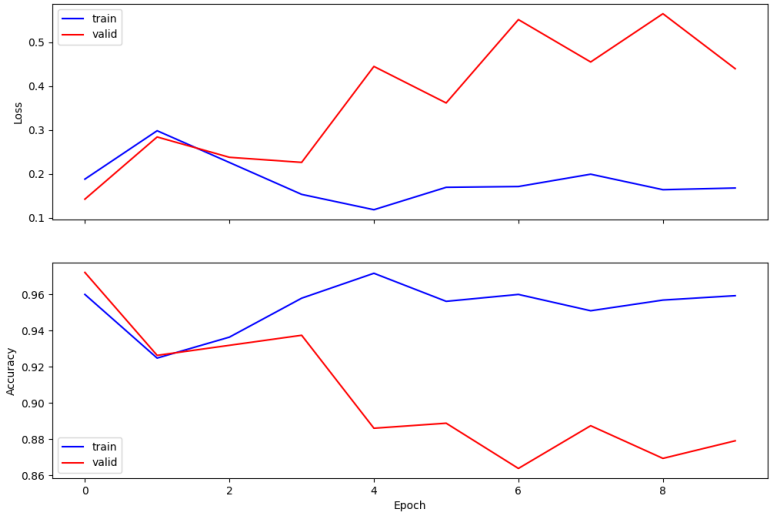


Fig. 19: The recorded loss and accuracy for the second best performing pretrained model architecture.

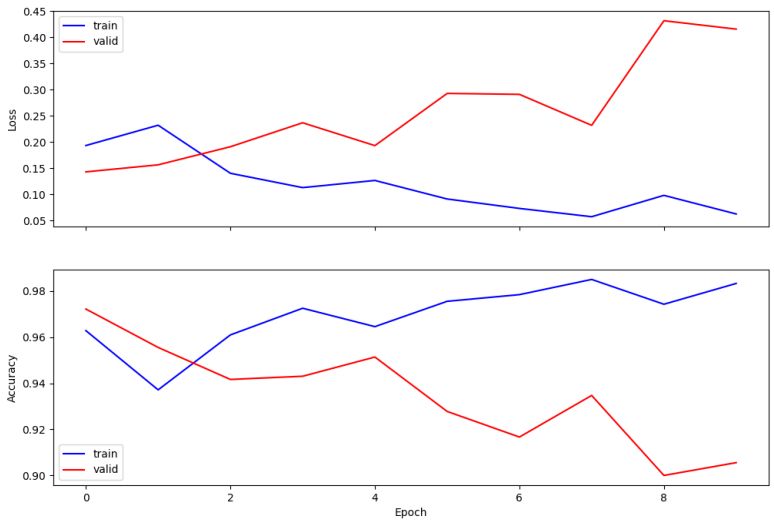


Fig. 20: The recorded loss and accuracy for the third best performing pretrained model architecture.

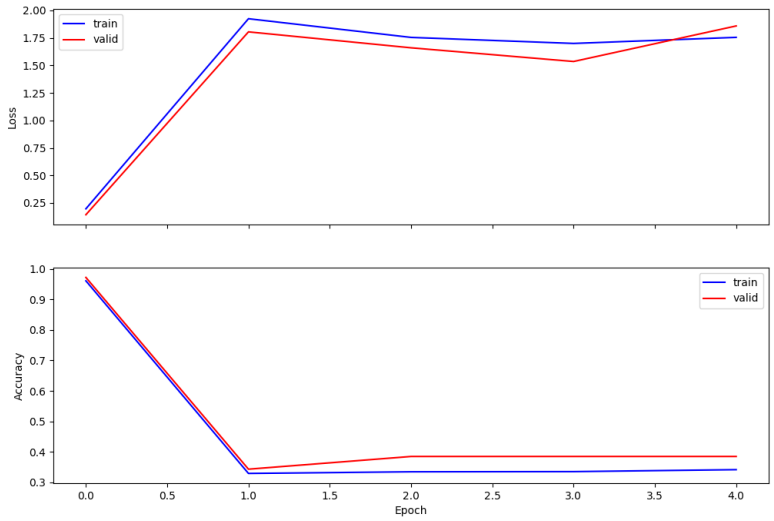


Fig. 21: The recorded loss and accuracy for the fourth best performing pretrained model architecture.

Table 9: The performance scores of the fine-tuned pretrained model on the test set.

Score	**Value (%)**
Accuracy	89.06
Precision	82.57
F1	78.01
Recall	75.52

Appendix B. Experimentation Results - Intel Image - Resnet CNN

Table 10: Trialled hyper-parameters

Learning Rate	0.0001, 0.001, 0.01, 0.1
Epochs	5, 10, 25, 50
Widths	10, 50, 100, 150
Activation Function	ReLU, CELU, Sigmoid

Table 12: Trialled hyper-parameters with results

W	**LR**	**E**	**AF**	**R**	**P**	**A**
10	0.0001	5	ReLU	0.75	0.78	0.75
10	0.001	5	ReLU	0.75	0.78	0.75
10	0.01	5	ReLU	0.58	0.58	0.58
10	0.1	5	ReLU	0.58	0.64	0.58
10	0.001	5	ReLU	0.75	0.78	0.75
50	0.001	5	ReLU	0.75	0.78	0.75
100	0.001	5	ReLU	0.67	0.56	0.67
150	0.001	5	ReLU	0.67	0.69	0.67
50	0.001	5	ReLU	0.33	0.19	0.33
50	0.001	10	ReLU	0.75	0.81	0.75
50	0.001	25	ReLU	0.75	0.69	0.75
50	0.001	50	ReLU	0.83	0.83	0.83
50	0.001	50	CeLU	0.67	0.72	0.67
50	0.001	50	Sigmoid	0.67	0.67	0.67

Trialled and Initial Values

Width = 10, No. Epochs = 5, LR = 0.0001, AF = ReLU

Confusion matrix - $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$

Table 14: Width = 10, No. Epochs = 5, LR = 0.0001, AF = ReLU

	Precision	**Recall**	**f1-Score**	**Support**
0	0.50	0.50	0.50	2
1	1.00	1.00	1.00	2
2	1.00	1.00	1.00	2
3	1.00	0.50	0.67	2
4	0.67	1.00	0.80	2
5	0.50	0.50	0.50	2
Accuracy			0.75	12
Macro Avg	0.78	0.75	0.74	12
Weighted Avg	0.78	0.75	0.74	12

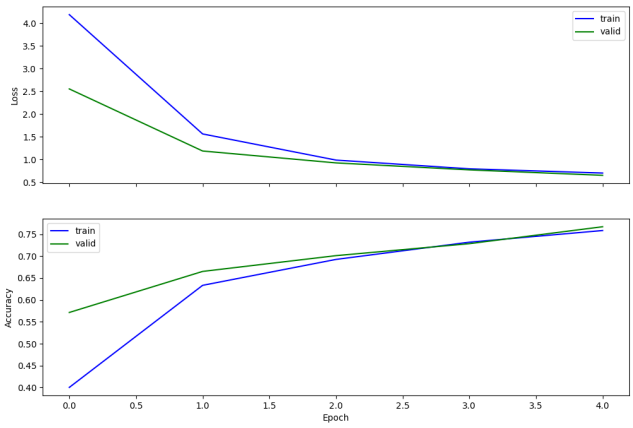


Fig. 22: Training Graph - Width = 10, No. Epochs = 5, LR = 0.001, AF = ReLU

Width = 10, No. Epochs = 5, LR = 0.001, AF = ReLU

Confusion matrix - $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$

Table 16: Width = 10, No. Epochs = 5, LR = 0.001, AF = ReLU

	Precision	**Recall**	**f1-Score**	**Support**
0	0.50	0.50	0.50	2
1	1.00	1.00	1.00	2
2	1.00	1.00	1.00	2
3	1.00	0.50	0.67	2
4	0.67	1.00	0.80	2
5	0.50	0.50	0.50	2
Accuracy			0.75	12
Macro Avg	0.78	0.75	0.74	12
Weighted Avg	0.78	0.75	0.74	12

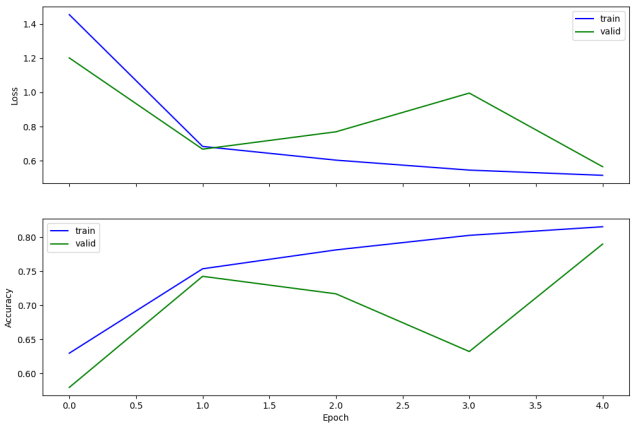


Fig. 23: Training Graph - Width = 10, No. Epochs = 5, LR = 0.001, AF = ReLU

Width = 10, No. Epochs = 5, LR = 0.01, AF = ReLU

Confusion matrix - $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$

Table 18: Width = 10, No. Epochs = 5, LR = 0.01, AF = ReLU

	Precision	**Recall**	**f1-Score**	**Support**
0	0.50	0.50	0.50	2
1	1.00	1.00	1.00	2
2	0.50	1.00	0.67	2
3	0.50	0.50	0.50	2
4	1.00	0.50	0.67	2
5	0.00	0.00	0.00	2
Accuracy			0.58	12
Macro Avg	0.58	0.58	0.56	12
Weighted Avg	0.58	0.58	0.56	12

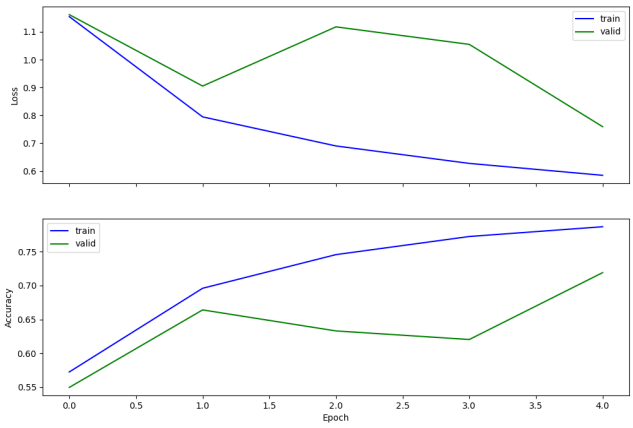


Fig. 24: Training Graph - Width = 10, No. Epochs = 5, LR = 0.01, AF = ReLU

Width = 10, No. Epochs = 5, LR = 0.1, AF = ReLU

Confusion matrix - $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$

Table 20: Width = 10, No. Epochs = 5, LR = 0.1, AF = ReLU

	Precision	**Recall**	**f1-Score**	**Support**
0	0.50	0.50	0.50	2
1	1.00	1.00	1.00	2
2	1.00	0.50	0.67	2
3	0.50	0.50	0.50	2
4	0.33	0.50	0.40	2
5	0.50	0.50	0.50	2
Accuracy			0.58	12
Macro Avg	0.64	0.58	0.59	12
Weighted Avg	0.64	0.58	0.59	12

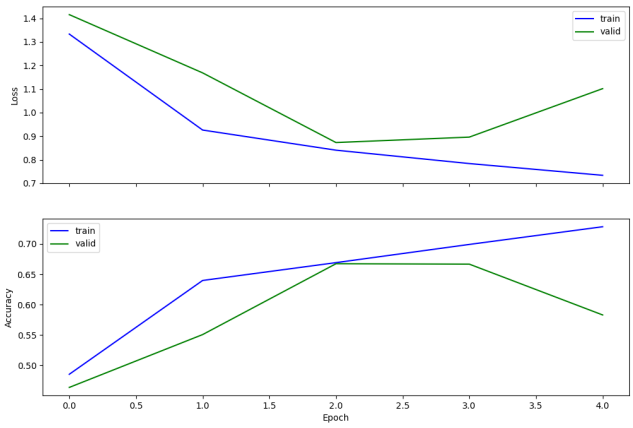


Fig. 25: Training Graph - Width = 10, No. Epochs = 5, LR = 0.1, AF = ReLU

Width = 50, No. Epochs = 5, LR = 0.001, AF = ReLU

Confusion matrix - $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$

Table 22: Width = 50, No. Epochs = 5, LR = 0.001, AF = ReLU

	Precision	**Recall**	**f1-Score**	**Support**
0	0.50	0.50	0.50	2
1	0.67	1.00	0.80	2
2	1.00	0.50	0.67	2
3	1.00	1.00	1.00	2
4	1.00	1.00	1.00	2
5	0.50	0.50	0.50	2
Accuracy			0.75	12
Macro Avg	0.78	0.75	0.74	12
Weighted Avg	0.78	0.75	0.74	12

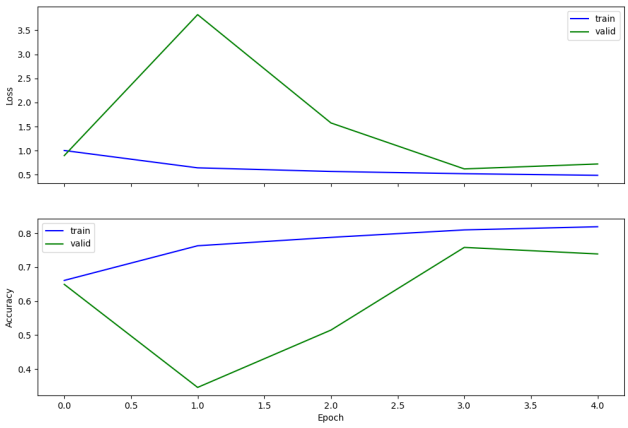


Fig. 26: Training Graph - Width = 50, No. Epochs = 5, LR = 0.001, AF = ReLU

Width = 100, No. Epochs = 5, LR = 0.001, AF = ReLU

Confusion matrix - $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$

Table 24: Width = 100, No. Epochs = 5, LR = 0.001, AF = ReLU

	Precision	**Recall**	**f1-Score**	**Support**
0	0.50	0.50	0.50	2
1	1.00	1.00	1.00	2
2	0.67	1.00	0.80	2
3	0.67	1.00	0.80	2
4	0.00	0.00	0.00	2
5	0.50	0.50	0.50	2
Accuracy			0.67	12
Macro Avg	0.56	0.67	0.60	12
Weighted Avg	0.56	0.67	0.60	12

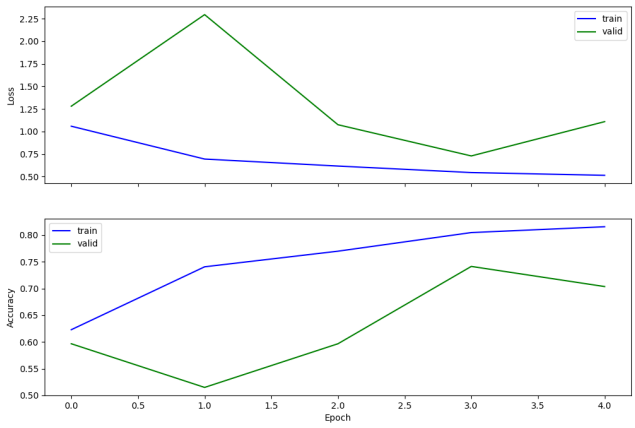


Fig. 27: Training Graph - Width = 100, No. Epochs = 5, LR = 0.001, AF = ReLU

Width = 150, No. Epochs = 5, LR = 0.001, AF = ReLU

Confusion matrix - $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$

Table 26: Width = 150, No. Epochs = 5, LR = 0.001, AF = ReLU

	Precision	**Recall**	**f1-Score**	**Support**
0	0.50	0.50	0.50	2
1	1.00	1.00	1.00	2
2	1.00	0.50	0.67	2
3	0.67	1.00	0.80	2
4	0.50	0.50	0.50	2
5	0.50	0.50	0.50	2
Accuracy			0.67	12
Macro Avg	0.69	0.67	0.66	12
Weighted Avg	0.69	0.67	0.66	12

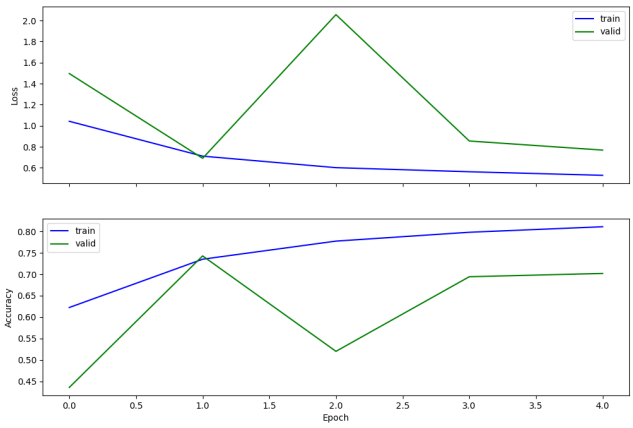


Fig. 28: Training Graph - Width = 150, No. Epochs = 5, LR = 0.001, AF = ReLU

Width = 50, No. Epochs = 10, LR = 0.001, AF = ReLU

Confusion matrix - $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$

Table 28: Width = 50, No. Epochs = 10, LR = 0.001, AF = ReLU

	Precision	**Recall**	**f1-Score**	**Support**
0	0.33	0.50	0.40	2
1	1.00	1.00	1.00	2
2	1.00	0.50	0.67	2
3	1.00	1.00	1.00	2
4	1.00	1.00	1.00	2
5	0.50	0.50	0.50	2
Accuracy			0.75	12
Macro Avg	0.81	0.75	0.76	12
Weighted Avg	0.81	0.75	0.76	12

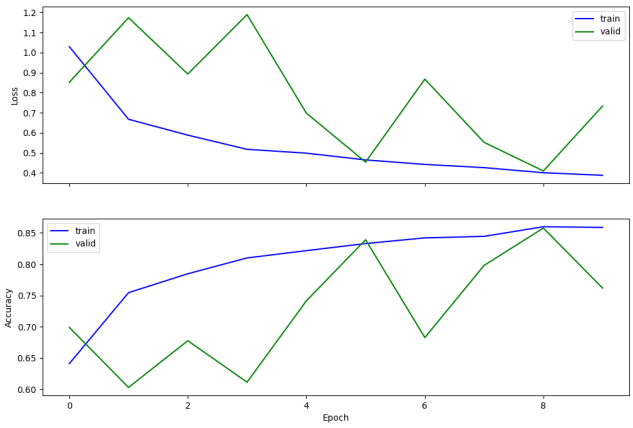


Fig. 29: Training Graph - Width = 50, No. Epochs = 5, LR = 0.001, AF = ReLU

Width = 50, No. Epochs = 25, LR = 0.001, AF = ReLU

Confusion matrix - $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$

Table 30: Width = 50, No. Epochs = 25, LR = 0.001, AF = ReLU

	Precision	**Recall**	**f1-Score**	**Support**
0	0.50	0.50	0.50	2
1	1.00	1.00	1.00	2
2	1.00	1.00	1.00	2
3	0.67	1.00	0.80	2
4	1.00	1.00	1.00	2
5	0.00	0.00	0.00	2
Accuracy			0.75	12
Macro Avg	0.69	0.75	0.72	12
Weighted Avg	0.69	0.75	0.72	12

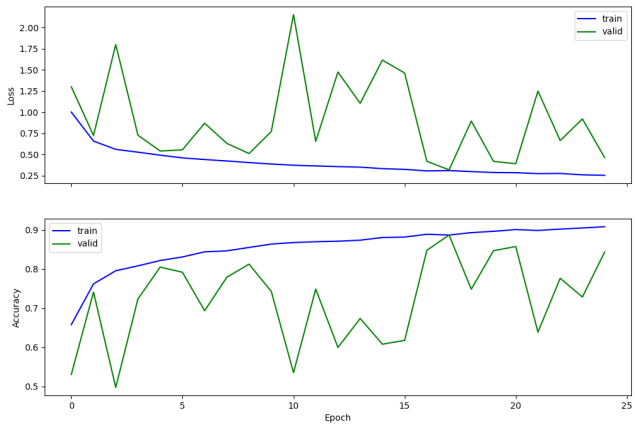


Fig. 30: Training Graph - Width = 50, No. Epochs = 25, LR = 0.001, AF = ReLU

Width = 50, No. Epochs = 50, LR = 0.001

Confusion matrix - $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$

Table 32: Width = 50, No. Epochs = 50, LR = 0.001, AF = ReLU

	Precision	**Recall**	**f1-Score**	**Support**
0	0.50	0.50	0.50	2
1	1.00	1.00	1.00	2
2	1.00	1.00	1.00	2
3	1.00	1.00	1.00	2
4	1.00	1.00	1.00	2
5	0.50	0.50	0.50	2
Accuracy			0.83	12
Macro Avg	0.83	0.83	0.83	12
Weighted Avg	0.83	0.83	0.83	12

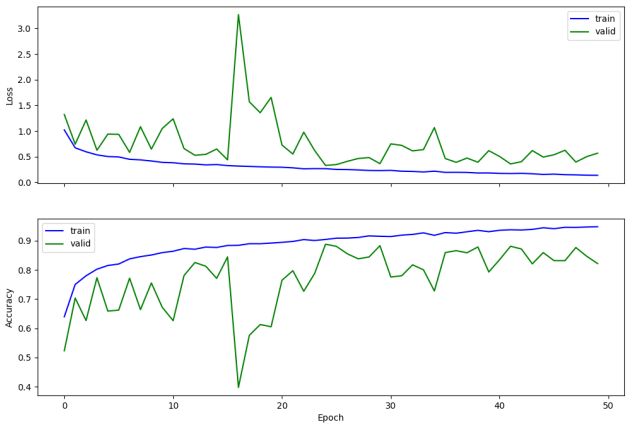


Fig. 31: Training Graph - Width = 50, No. Epochs = 50, LR = 0.001, AF = ReLU

Width = 50, No. Epochs = 50, LR = 0.001, AF = CELU

Confusion matrix - $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$

Table 34: Width = 50, No. Epochs = 50, LR = 0.001, AF = CELU

	Precision	**Recall**	**f1-Score**	**Support**
0	0.33	0.50	0.40	2
1	1.00	1.00	1.00	2
2	1.00	0.50	0.67	2
3	0.50	0.50	0.50	2
4	1.00	1.00	1.00	2
5	0.50	0.50	0.50	2
Accuracy			0.67	12
Macro Avg	0.72	0.67	0.68	12
Weighted Avg	0.72	0.67	0.68	12

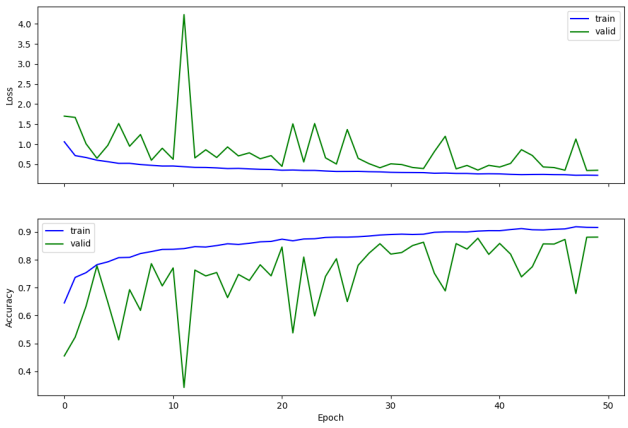


Fig. 32: Training Graph - Width = 50, No. Epochs = 50, LR = 0.001, AF = CELU

Width = 50, No. Epochs = 50, LR = 0.001, AF = Sigmoid

Confusion matrix - $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$

Table 36: Width = 50, No. Epochs = 50, LR = 0.001, AF = Sigmoid

	Precision	**Recall**	**f1-Score**	**Support**
0	0.33	0.50	0.40	2
1	1.00	1.00	1.00	2
2	1.00	0.50	0.67	2
3	0.50	0.50	0.50	2
4	1.00	1.00	1.00	2
5	0.50	0.50	0.50	2
Accuracy			0.67	12
Macro Avg	0.72	0.67	0.68	12
Weighted Avg	0.72	0.67	0.68	12

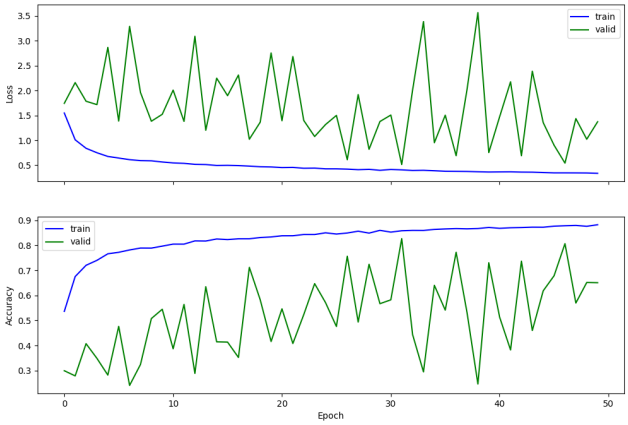


Fig. 33: Training Graph - Width = 50, No. Epochs = 50, LR = 0.001, AF = Sigmoid

Appendix C. Experimentation Results - Intel Image - Alexnet CNN

Appendix D. Tested Hyper Parameters

Table 38: Trialled hyper-parameters

Learning Rate	0.0001, 0.001, 0.01, 0.1
Epochs	5, 10, 25, 50
Activation Function	ReLU, CELU, Sigmoid

Table 40: Trialled hyper-parameters

LR	**E**	**AF**	**R**	**P**	**A**
0.1	5	ReLU	0.17	0.03	0.17
0.01	5	ReLU	0.50	0.47	0.50
0.001	5	ReLU	0.75	0.78	0.75
0.001	5	ReLU	0.67	0.75	0.67
0.001	10	ReLU	0.67	0.75	0.67
0.001	25	ReLU	0.75	0.78	0.75
0.001	50	ReLU	0.67	0.72	0.67

AlexNet, e:5, LR:0.1, AF = ReLU

Confusion matrix - $\begin{bmatrix} 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}$

Table 42: AlexNet, e:5, LR:0.1, AF = ReLU

	Precision	**Recall**	**f1-Score**	**Support**
0	0.00	0.00	0.00	2
1	0.00	0.00	0.00	2
2	0.17	1.00	0.29	2
3	0.00	0.00	0.00	2
4	0.00	0.00	0.00	2
5	0.00	0.00	0.00	2
Macro Avg	0.03	0.17	0.05	12
Weighted Avg	0.03	0.17	0.05	12

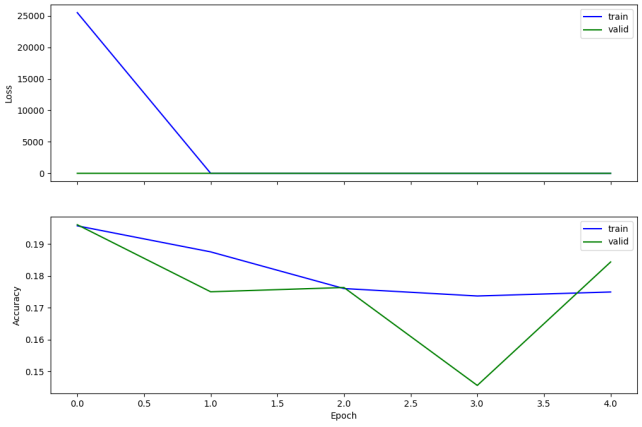


Fig. 34: Training Graph - AlexNet, e:5, LR:0.1, AF = ReLU

AlexNet, No. Epochs:5, LR = 0.01, AF = ReLU

Confusion matrix -

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \end{bmatrix}$$

Table 44: AlexNet, No. Epochs:5, LR = 0.01, AF = ReLU

	Precision	**Recall**	**f1-Score**	**Support**
0	0.00	0.00	0.00	2
1	0.00	0.00	0.00	2
2	1.00	0.50	0.67	2
3	0.50	1.00	0.67	2
4	1.00	0.50	0.67	2
5	0.33	1.00	0.50	2
Macro Avg	0.47	0.50	0.42	12
Weighted Avg	0.47	0.50	0.42	12

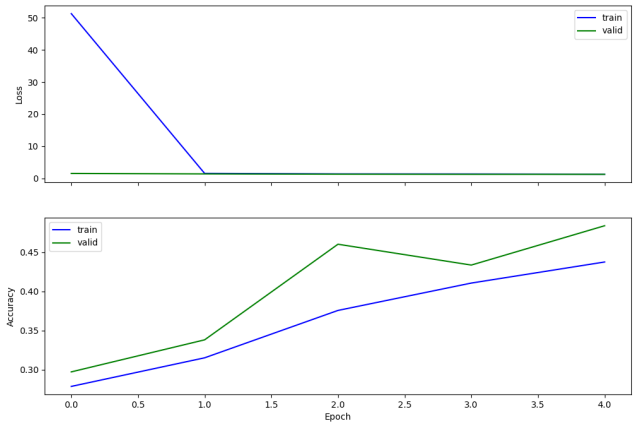


Fig. 35: Training Graph - AlexNet, No. Epochs:5, LR = 0.01, AF = ReLU

AlexNet, No. Epochs:5, LR = 0.001, AF = ReLU

Confusion matrix - $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$

Table 46: AlexNet, No. Epochs:5, LR = 0.001, AF = ReLU

	Precision	**Recall**	**f1-Score**	**Support**
0	0.50	0.50	0.50	2
1	1.00	1.00	1.00	2
2	0.67	1.00	0.80	2
3	1.00	1.00	1.00	2
4	1.00	0.50	0.67	2
5	0.50	0.50	0.50	2
Macro Avg	0.78	0.75	0.74	12
Weighted Avg	0.78	0.75	0.74	12

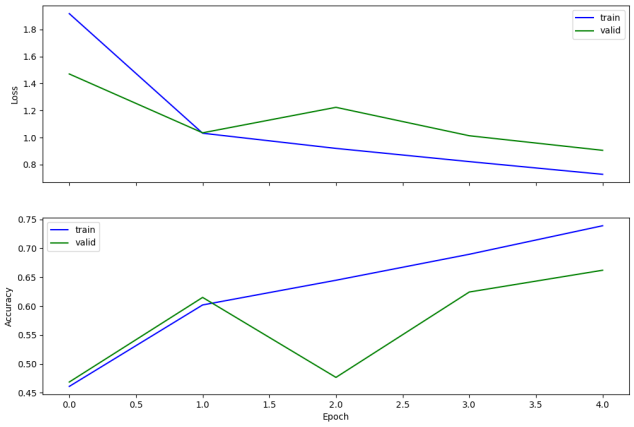


Fig. 36: Training Graph - AlexNet, w:10, No. Epochs:5, LR = 0.001, AF = ReLU

AlexNet, No. Epochs:5, LR = 0.0001, AF = ReLU

Confusion matrix - $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$

Table 48: AlexNet, No. Epochs:5, LR = 0.0001, AF = ReLU

	Precision	**Recall**	**f1-Score**	**Support**
0	0.33	0.50	0.40	2
1	1.00	1.00	1.00	2
2	1.00	0.50	0.67	2
3	1.00	1.00	1.00	2
4	0.67	1.00	0.80	2
5	0.00	0.00	0.00	2
Macro Avg	0.67	0.67	0.64	12
Weighted Avg	0.67	0.67	0.64	12

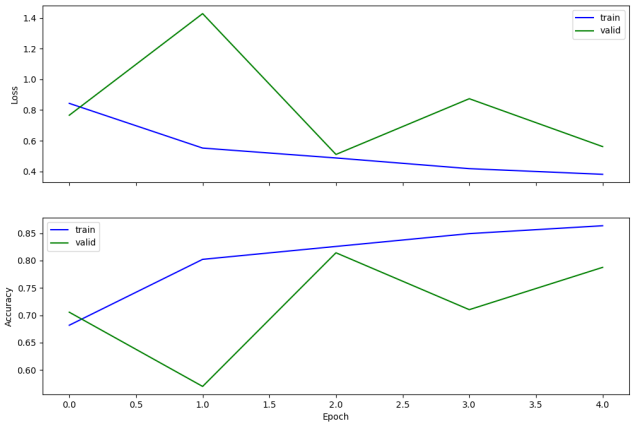


Fig. 37: Modified Training Graph - AlexNet, No. Epochs:5, LR = 0.0001, AF = ReLU

AlexNet, No. Epochs:10, LR = 0.001, AF = ReLU

Confusion matrix - $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$

Table 50: AlexNet, No. Epochs:10, LR = 0.001, AF = ReLU

	Precision	**Recall**	**f1-Score**	**Support**
0	0.50	0.50	0.50	2
1	1.00	0.50	0.67	2
2	1.00	1.00	1.00	2
3	0.67	1.00	0.80	2
4	0.50	0.50	0.50	2
5	0.50	0.50	0.50	2
Macro Avg	0.69	0.67	0.66	12
Weighted Avg	0.69	0.67	0.66	12

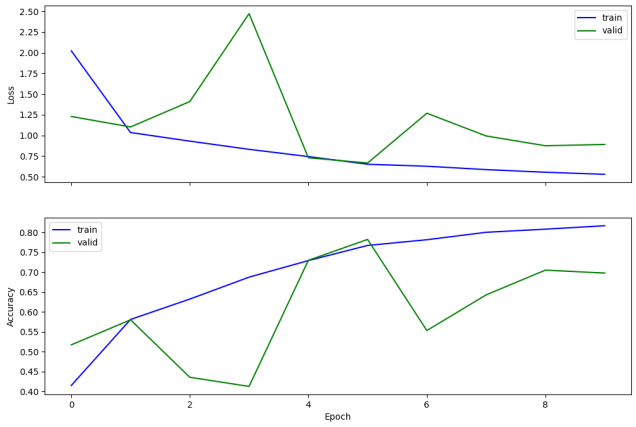


Fig. 38: Training Graph - AlexNet, No. Epochs:10, LR = 0.001, AF = ReLU

AlexNet, No. Epochs:25, LR = 0.001, AF = ReLU

Confusion matrix -

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Table 52: AlexNet, No. Epochs:25, LR = 0.001, AF = ReLU

	Precision	**Recall**	**f1-Score**	**Support**
0	0.50	0.50	0.50	2
1	1.00	1.00	1.00	2
2	0.67	1.00	0.80	2
3	1.00	0.50	0.67	2
4	1.00	1.00	1.00	2
5	0.50	0.50	0.50	2
Macro Avg	0.78	0.75	0.74	12
Weighted Avg	0.78	0.75	0.74	12

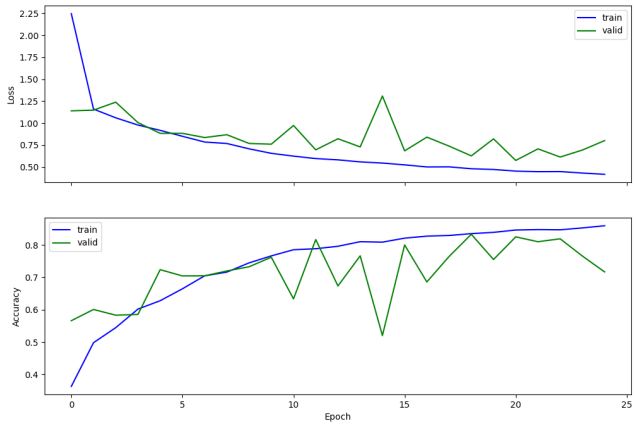


Fig. 39: Training Graph - AlexNet, No. Epochs:25, LR = 0.001, AF = ReLU

AlexNet, No. Epochs:50, LR = 0.001, AF = ReLU

Confusion matrix - $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$

Table 54: AlexNet, No. Epochs:50, LR = 0.001, AF = ReLU

	Precision	**Recall**	**f1-Score**	**Support**
0	0.50	0.50	0.50	2
1	0.67	1.00	0.80	2
2	1.00	0.50	0.67	2
3	1.00	0.50	0.67	2
4	0.67	1.00	0.80	2
5	0.50	0.50	0.50	2
Macro Avg	0.72	0.67	0.66	12
Weighted Avg	0.72	0.67	0.66	12

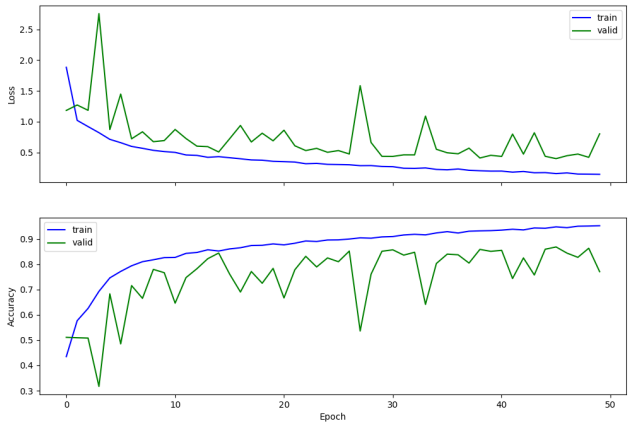


Fig. 40: Training Graph - AlexNet, No. Epochs:50, LR = 0.001, AF = ReLU

Appendix E. Mobile Net

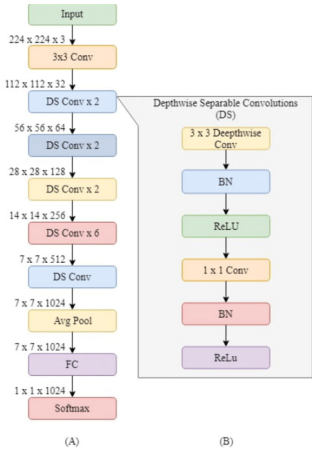


Fig. 41: MobileNet V1 Diagram.

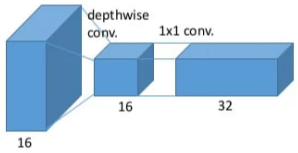


Fig. 42: Deptwise separated Conventional Layer

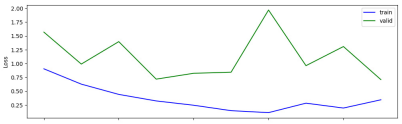


Fig. 43: Loss Result

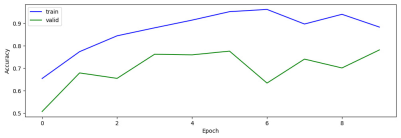


Fig. 44: Accuracy result

Appendix F. Source Code

```python
def standardise(self, text: str) -> List[str]:
if self.__lower_case:
    text = text.lower()

if self.__expand_contractions:
    text = contractions.fix(text)

if self.__remove_punctuation:
    text = "".join([char for char in text if char not in
    ↪ string.punctuation])

tokens: List[str] = word_tokenize(text=text, language="english")

if self.__remove_stop_words:
    tokens = [token for token in tokens if token not in
    ↪ self.__stop_words]

if self.__perform_normalisation:
    tokens = self.__normalise(tokens)

return tokens[:self.__max_tokens]
```

Listing 1: Python code snippet of the method for text standardisation, featuring conversion to lowercase, contraction expansion, punctuation removal, tokenisation, stop word removal, and optional lemmatisation or stemming.

```
def create_vocab(self, tokens: Iterable, min_freq: int = 1) ->
↪  None:
vocab: Vocab = build_vocab_from_iterator(
    tokens,
    specials=[self.__oov_token, self.__pad_token],
    min_freq=min_freq,
)
vocab.set_default_index(vocab["<unk>"])
self.__vocab = vocab
```

Listing 2: Python code snippet for the method that builds a vocabulary from an iterable of tokens. This method uses a minimum frequency to filter out rare words and includes special tokens for unknown and padding cases. The default index is set to the unknown token.

```
def multi_hot_encode(self, tokens: List[int]) -> ndarray:
if self.__vocab is None:
    raise ValueError("No vocabulary has been created. Please
    ↪  create a vocabulary before encoding.")
encoded: ndarray = zeros(len(self.__vocab), dtype=int)
encoded[[self.__vocab[str(token)] for token in tokens]] = 1
return encoded
```

Listing 3: Python code snippet for a method that converts a numerical data-set into a binary matrix representation.

```python
def vocabularise(self, tokens: List[str], create_vocab: bool = False) -> List[int]:
if self.__vocab is None:
    if create_vocab:
        self.create_vocab(tokens)
    else:
        raise ValueError("No vocabulary has been created. Please create a vocabulary before vocabularising.")

return [self.__vocab[token] for token in tokens]
```

Listing 4: Python code snippet of the method that converts a list of tokens into their corresponding numerical identifiers using a provided vocabulary. The output is a list of integers representing each token.