ECS8053 Final Assignment

Hugo Collins
Student Number: 40446925
Email: hcollins05@qb.ac.uk
MSc in Artificial Intelligence 2024

Word Count (Excluding Front Page, References, Table of Contents, Figure Titles, Titles and Tables): 3289

Abstract

This exploratory report investigates the creation of image classification models using hand-crafted features, neural network, and convolutional neural network approaches on the TinyImageNet100 dataset. This dataset contains 100 classes each having 500 64 x 64 sized images. In my report I document the process of creating the models using the limited dataset and I employ a split of 300/100/100 and use 15 classes for each model. Given the limited amount of data and the small image sizes, this study addresses the challenges posed by the dataset's complexity. Rather than being purely results-driven, the report emphasises understanding the relationships between variables and exploring the reasons behind the models' performance and addresses mechanisms to combat the low resource setting. This approach provides deeper insights into the data and the mechanisms influencing the models' predictions, aiming to guide future improvements in data processing and model development. The final CNN model achieved 62.27% accuracy on the test set.

TABLE OF CONTENTS

01 Introduction

Introduction 03

02 Data Preprocessing

Data Preprocessing 3/4

03 Methodology

Conducting image classification using the hand-crafted feature approach 4/5
Conducting image classification using the neural network approach 6/7
Conducting image classification using the convolutional neural network approach 7-12

04 Discussion

Significant Findings 12/14

05 Comparison of Approaches

Comparison of Results 15
Error Analysis 16/18

06 Conclusion 18

07 References 19

1. Introduction

In this report, I detail the development of three approaches for image classification on the TinyImageNet 100 dataset, focusing on performance with 15 classes. Image classification in computer vision involves assigning labels to images based on their content, and it is crucial for applications like object detection and facial recognition which is crucial in today's world. The report covers a handcrafted feature approach using a bag of words/Fisher vector model, a purely linear neural network, and a convolutional neural network (CNN). I document the process of improving accuracy, starting from 24.09% with the handcrafted approach to a successful 62.27% accuracy using a self-built CNN.

2. Data Preprocessing

The first step involved setting up a dictionary containing class names from the "class_name.txt" file. I then filtered the folders that began with "n0" and wrote a function to read all images in alphabetical order. To ensure reproducibility, I set a random seed and used the code to randomly select 15 folders along with their images. The number of folders to be selected can be adjusted by the user to allow for testing using many classes.

```
# Here I set my random seed for reproducibility
random.seed(42)

# Randomly select my 15 folders
folders_with_n0 = sorted([folder for folder in images_per_folder.keys() if folder.startswith('n0')])

if len(folders_with_n0) >= 15:
    # This function randomly choose 15 folders from my list of available folders
    random_folders = random.sample(folders_with_n0, 15)

    # This assigns my folders the labels which will be used during training and testing
    folder_to_label = {random_folders[i]: i + 1 for i in range(15)}

    print("Randomly Chosen Folders with Assigned Labels:")
    for folder in random_folders:
        print(f"Folder: {folder} - Label: {folder_to_label[folder]}")
else:
    print("Not enough folders starting with 'n0'.")
```

Fig 1. *Subsection of code used to extract random 15 folders*

The TinyImageNet100 dataset consists of 100 classes, each containing 500 images. To maintain consistency in training, I organised the data by folder. The images were sorted alphabetically and split such that the first 400 images of each class were used for training, with the remaining 100 for testing. I further divided the training set by taking the first 300 images for training and the next 100 for validation. This approach not only ensured a representative validation set but also facilitated a reproducible data split. (300/100/100 per folder)

The randomly selected folders, along with their respective labels, are listed below. Note that the original class labels were numbered 1 through 15, but to be compatible with tensors and PyTorch, I adjusted them to start at 0 and go up to 14.

```
Folder: n02808440 - Label: 0 - Class Name: bathtub, bathing tub, bath, tub
Folder: n01742172 - Label: 1 - Class Name: boa constrictor, Constrictor constrictor
Folder: n01443537 - Label: 2 - Class Name: goldfish, Carassius auratus
Folder: n02909870 - Label: 3 - Class Name: bucket, pail
Folder: n02481823 - Label: 4 - Class Name: chimpanzee, chimp, Pan troglodytes
Folder: n02364673 - Label: 5 - Class Name: guinea pig, Cavia cobaya
Folder: n02788148 - Label: 6 - Class Name: bannister, banister, balustrade, balusters, handrail
Folder: n01984695 - Label: 7 - Class Name: spiny lobster, langouste, rock lobster, crawfish, crayfish, sea crawfish
Folder: n02504458 - Label: 8 - Class Name: African elephant, Loxodonta africana
Folder: n03126707 - Label: 9 - Class Name: crane
Folder: n03179701 - Label: 10 - Class Name: desk
Folder: n02509815 - Label: 11 - Class Name: lesser panda, red panda, panda, bear cat, cat bear, Ailurus fulgens
Folder: n02730930 - Label: 12 - Class Name: apron
Folder: n03160309 - Label: 13 - Class Name: dam, dike, dyke
Folder: n02769748 - Label: 14 - Class Name: backpack, back pack, knapsack, packsack, rucksack, haversack
```

Fig 2. *Classes selected and Respective Labels*

3. Methodology

3.1. Conducting image classification using the hand-crafted feature approach

For the handcrafted features I used two different local feature extraction methods:

- **SIFT (**Scale-Invariant Feature Transform**)**
- **ORB (**Oriented FAST and Rotated BRIEF)

Due to their respective strengths and weaknesses each method required slightly different data preparation. I used the original data throughout all experiments. One shared preprocessing step was to covert the images to greyscale to make them compatible with both SIFT and ORB.

3.1.1 Feature Extraction

SIFT

Since SIFT is scale-invariant, resizing the images didn't provide any benefits, and using SIFT on larger images was too computationally expensive. At first, SIFT struggled to find useful features due to poor image quality. To fix this, I applied an enhancement filter to improve image clarity, making it easier for SIFT to extract features. I used two main techniques:

- Histogram Equalisation – Adjusted the intensity levels to improve exposure.
- Unsharp Masking – Increased contrast at edges and fine details by subtracting a blurred version of the image.

After these enhancements, SIFT was able to extract the necessary features successfully.

ORB

Originally, I tried to extract features using ORB on the 64x64 images but on closer inspection of the training feature vectors I discovered that only about 10 images were returning relevant features and the rest returned nothing. As a result, accuracy scores were incredibly low. As ORB is not scale-invariant but is far more computationally efficient than SIFT I was able to resize the images to 256 x 256. This resulted in far more effective feature extraction as seen below:

```
Images with keypoints (non-zero): 4490
Images with no keypoints (zero): 10
Total images: 4500
```

Fig 3. *Training images with ORB key points found*

2.1.2 Extracting Mid-Level Features

To build upon my previous low level feature extraction methods I then used two further techniques. These were a:

- Bag of Words Model
- Fisher Vector Approach

Bag of Words Model:

I used MiniBatchKMeans in my BoW model to cluster my SIFT descriptors which created my list of "key words". I used MiniBatchKMeans as it was more efficient. The vectors then represented each image as a histogram based on counts of these visual words.

Fisher Vector Approach

This approach differed from the BoW model as it didn't focus on count but mainly on statistical information about the distribution of my features. Instead of Kmeans clustering I fitted a Gaussian Mixture Model.

The benefit to using mid-level approaches was that they improved image comparison by grouping similar local features instead of relying on raw pixel values.

3.1.3 Classification using SVC

To assess the classification ability of my features I used a linear support vector machine to train. I used a validation set to tune the best C and max_iter parameters (0.1, 1000) and then ran several different tests with varying number of key words/clusters. I did experiment with a non-linear kernel also, but the results were too poor to include as the model had practically no predictive power and was just guessing label 1 for all.

The results of the iterations can be seen below. As the number of words passed 125 there appeared to be overfitting and test scores began to suffer

Model	Num_Words	Accuracy
SIFT Bag of Words	50	21.67%
SIFT Bag of Words	100	23.13%
SIFT Bag of Words	150	21.47%
SIFT Bag of Words	200	20.40%
ORB Bag of Words	50	24.07%
ORB Bag of Words	100	22.73%
ORB Bag of Words	150	22.40%
ORB Bag of Words	200	21.87%

Fig 4. *Bag of Words Model Results*

Model	Num_Clusters	Accuracy
SIFT Fisher Vector	25	14.07%
SIFT Fisher Vector	50	18%
SIFT Fisher Vector	100	15.73%
ORB Fisher Vector	25	20.60%
ORB Fisher Vector	50	24.09%
ORB Fisher Vector	100	22.67%

Fig 5. *Fisher Vector Approach Results*

3.2 Conducting image classification using the neural network approach

The next approach I took was creating a linear neural network to tackle this image classification task. The first step in all the models was to flatten the input size to 1D. As my input images were of size 64x64x3 this led to a flattened size of 12,288 which was fixed for all models. As a baseline I began with a V1 model.

Before designing my NNs, I performed some data preprocessing. The training dataset contained a mix of grayscale and RGB images, which impacted the results and caused errors due to varying channel sizes. To address this, I converted all grayscale images to RGB format. This process was slow and computationally expensive, so to avoid repeating it in every run, I saved the converted images and loaded them at the start of each experiment.

For initial testing I used Dataloader to batch my data into sizes of 32 for training, testing and validation. Images remained at size 64x64. I also used transforms.normalise to normalise my features also which improved results. As a starting point I decided to explore numerous different basic models which contained varying number of layers

3.2.1 Models and Architecture

LinearNNV1: Baseline

This model serves as my baseline.

- Input Size: 12288 (flattened image).
- 2 Hidden Layers:
- First hidden layer with 512.
- Second hidden layer with 256.
- Activation Function: ReLU.
- Dropout: Dropout with a rate of 0.5 applied after each hidden layer for regularisation.
- Output Layer: Fully connected layer with 15 output neurons representing the 15 classes.

LinearNNV2: Add Layers

- Input Size: Same as V1 (12288).
- 4 Hidden Layers: [1024, 512,256,128]
- Changed dropout to 0.3

LinearNNV3: Tried to get better features

- 6 Hidden Layers: [2048, 1024, 512, 256, 128, 64]
- I tried a new dropout with initial rate of 0.5, gradually decreasing at each layer (0.45 → 0.4 → 0.35 → 0.3 → 0.25 → 0.2).
- I hoped this would help generalisation while still allowing good features.

Results from the first run can be seen below

Model (architecture)	Epochs	Batch_Size	Optimiser	Data Augmentation	Early Stopping	Hyperparameter Tuning	Training Acc	Validation Acc	Test Acc	Test Loss
V1	40	32	Adam(lr=0.001)	No	No	No	27.73%	19.20%	22.60%	2.4754
V2	40	32	Adam(lr=0.001)	No	No	No	30.18%	18.07%	18.20%	2.6662
V3	40	32	Adam(lr=0.001)	No	No	No	33.93%	23.93%	25.73%	2.6344
V1	50	64	SGD(lr=0.001 Dropout=0.3)	No	Yes	Yes	48.31%	34.26%	33.47%	2.1243
V2	60	128	SGD(lr=0.01 Dropout=0.3)	No	Yes	Yes	39.73%	32.47%	31.53%	2.1900
V3	40	32	SGD(lr=0.01) Dropout=0.3	No	Yes	Yes	38.76%	28.13%	26.54%	2.4587

Fig 6. *Results of Linear Neural Networks first run*

Overall, these metrics were quite disappointing. While V1 returned the highest test score there was severe overfitting and most of the results were from guesswork whereas V2 when hyperparameter tuned showed promise of a stable learning rate and less overfitting, so I decided to take this further and try some more tuning and data augmentation on V2.

The results of these extra runs can be seen below:

Model (architecture)	Epochs	Batch_Size	Optimiser	Data Augmentation	Early Stopping	Hyperparameter Tuning	Training Acc	Validation Acc	Test Acc	Test Loss
V2	60	128	SGD(0.001, momentum=0.9) Dropout Rate=0.3	No	Yes (Patience=5)	Yes	45.42%	32.47%	33.00%	2.1522
V2	60	128	SGD(0.001, momentum=0.9) Dropout Rate=0.5	No	Yes (Patience=5)	Yes	32.09%	29.73%	30.00%	2.1893
V2	100	128	SGD(0.001, momentum=0.9) Dropout Rate=0.3	Yes	Yes (Patience=5)	Yes	44.20%	28.93%	30.47%	2.2116
V2	100	128	SGD(0.001, momentum=0.9) Dropout Rate=0.5	Yes	Yes (Patience=5)	Yes	35.00%	27.33%	27.67%	2.2888

Fig 7. *Results of V2 refined Linear Neural Network*

The use of techniques such as changing the learning rate, batch normalisation and data augmentation had little to no effect on performance. My key findings are discussed in the discussion section below.

3.3 Conducting image classification using the convolutional neural network approach

The next approach I took was using a Convolutional Neural Network approach to tackle the image classification problem. My hopes were that the more complex architecture and use for convolutional layers and pooling would help improve on my previous approaches. For training I connected to a A100 GPU provided by Google Colab.

I used the same initial preprocessing and batching as the linear neural network approach.

3.3.1 Model Architectures

To improve the performance of my image classification task, I iteratively refined my Convolutional Neural Network (CNN) models from ECS8051modelV1 to ECS8051modelV5. Each iteration introduced changes to feature extraction, regularisation, and overall classification accuracy. Below is an explanation of how and why I evolved my models across these five versions.

ECS8051modelV1: Baseline Model

The first model, ECS8051modelV1, served as my simple baseline CNN to establish initial performance metrics.

- It consisted of two convolutional layers with 32 filters each, both using a kernel size of 3x3.
- Activation function - ReLU,
- Dropout (0.5) after linear layer to regularise
- One max-pooling layer - Reduce number of features and reduce dimension.
- A fully connected layer with 512 neurons was introduced
- Final classification layer with 15 output classes.

Reasoning: I kept this model simple to use as a baseline for performance.

ECS8051modelV2:

Difference - Reducing Complexity for Generalisation

- The dense layer was reduced from 512 to 256 neurons, to try to reduce overfitting.
- Rest was the same as V1

ECS8051modelV3: Increasing Feature Extraction Capacity

In ECS8051modelV3 I tried to enhance feature extraction as V1 and V2 were struggling:

- I increased filter sizes to 128 and 256 filters in the two convolutional layers.
- Added a second fully connected layer 512 → 256 was introduced, making my model deeper.
- Added an extra dropout layer (0.5)

Reasoning: I wanted the model to learn more complex features to improve accuracy.

ECS8051modelV4: Deepening the Network for Improved Representation

Building on V3, ECS8051modelV4 further increased the depth of the network:

- I added a fourth convolutional layer (32 → 64 → 128 → 256 filters).
- Added three max-pooling layers, progressively reducing spatial dimensions while preserving important features.

Reasoning: I structured it like successful CNN models such as VGG to compare performance.

ECS8051modelV5: Tried to reduce overfitting in V4 while retaining features

The final model, ECS8051modelV5, I refined the design by making the network more computationally efficient.

- I kept the fourth convolutional layer as it was performing well but reduced the initial filter size.
- Had three fully connected layers
- Higher dropout rates (0.4) in fully connected layers for better generalisation.

The first few runs involved running models using their base setting of Stochastic Gradient descent and the learning rate set to 0.01% and no hyperparameter tuning or data augmentation. I also implemented early stopping with a patience of 5 on some of the runs which can be seen below

Model (architecture)	Epochs	Batch_Size	Optimiser	Early Stopping	Hyperparameter Tuning	Training Acc	Validation Acc	Test Acc	Test Loss
V1	40	64	SGD(lr=0.01) momentum=0.9	No	No	96.38%	39.40%	40.60%	2.4998
V1	10	64	SGD	Yes	No	85.42%	39.73	42.67%	2.2538
V2	40	64	SGD	No	No	97.84%	41.80%	42.00%	3.3799
V2	12	64	SGD	Yes	No	80.82%	42.20%	41.07%	2.1182
V3	40	64	SGD	No	No	97.29%	44.80%	43.07%	2.5002
V3	14	64	SGD	Yes	No	83.31%	45.13%	44.47%	2.1804
V4	40	64	SGD	No	No	92.31%	46.93%	46.87%	1.8266
V4	17	64	SGD	Yes	No	82.44%	45.47%	48.07%	1.8029
V5	40	64	SGD	No	No	78.20%	43.80%	42.20%	2.2047
V5	22	64	SGD	Yes	No	57.73%	40.33%	44.13%	1.8658

Fig 8. *Results of V1-V5 Convolutional Neural Networks*

V1, V2, V3 all showed extreme overfitting to training data and lacked the ability to generalise well. V4 showed the ability to recognise important features but was overfitting whereas V5 showed a balance between both. I decided to use these two models for further testing. It can also be seen that early stopping was effective at improving results on test set.

3.3.2 Hyperparameter Tuning and Data Augmentation

The next step I took was to start tuning the hyperparameters of the best models from above; V4 and V5. During the hyperparameter tuning I used a grid search method and experimented with the following parameters:

```
param_grid = {
    'learning_rate': [0.001, 0.01],
    'batch_size': [32,64,128,256],
    'dropout_rate': [0.3,0.5],
    'num_epochs': [40,50,60],
    'optimizer': ['adam', 'sgd']  # I wanted to test both Adam and SGD optimizers
}
```

Fig 9. *Parameter Grid for Grid Search Hyperparmeter Tuning*

The code below is a segment of the output of one of the hyperparameter testing loops. By doing it this way I was able to get the output of all the different models being tested which was a major benefit.

```
Early stopping triggered after epoch 15
Training with params: {'batch_size': 32, 'dropout_rate': 0.3, 'learning_rate': 0.001, 'num_epochs': 40, 'optimizer': 'sgd'}
Epoch 1/40 - Train Accuracy: 14.56% | Train Loss: 2.5931 | Val Accuracy: 23.47% | Val Loss: 2.4200
Epoch 2/40 - Train Accuracy: 23.33% | Train Loss: 2.4049 | Val Accuracy: 25.13% | Val Loss: 2.3331
Epoch 3/40 - Train Accuracy: 26.47% | Train Loss: 2.3068 | Val Accuracy: 28.27% | Val Loss: 2.2701
Epoch 4/40 - Train Accuracy: 29.58% | Train Loss: 2.2259 | Val Accuracy: 30.20% | Val Loss: 2.2298
Epoch 5/40 - Train Accuracy: 31.40% | Train Loss: 2.1584 | Val Accuracy: 30.40% | Val Loss: 2.1934
Epoch 6/40 - Train Accuracy: 33.47% | Train Loss: 2.0982 | Val Accuracy: 30.87% | Val Loss: 2.1891
Epoch 7/40 - Train Accuracy: 35.60% | Train Loss: 2.0374 | Val Accuracy: 31.53% | Val Loss: 2.1789
Epoch 8/40 - Train Accuracy: 38.02% | Train Loss: 1.9668 | Val Accuracy: 30.67% | Val Loss: 2.1668
Epoch 9/40 - Train Accuracy: 38.40% | Train Loss: 1.9321 | Val Accuracy: 32.33% | Val Loss: 2.1523
Epoch 10/40 - Train Accuracy: 40.84% | Train Loss: 1.8617 | Val Accuracy: 32.47% | Val Loss: 2.1545
Epoch 11/40 - Train Accuracy: 42.84% | Train Loss: 1.8174 | Val Accuracy: 33.00% | Val Loss: 2.1569
Epoch 12/40 - Train Accuracy: 45.71% | Train Loss: 1.7549 | Val Accuracy: 32.87% | Val Loss: 2.1523
Epoch 13/40 - Train Accuracy: 45.18% | Train Loss: 1.7179 | Val Accuracy: 33.00% | Val Loss: 2.1751
Early stopping triggered after epoch 14
Training with params: {'batch_size': 32, 'dropout_rate': 0.3, 'learning_rate': 0.001, 'num_epochs': 50, 'optimizer': 'adam'}
Epoch 1/50 - Train Accuracy: 11.76% | Train Loss: 3.1682 | Val Accuracy: 15.13% | Val Loss: 2.6058
Epoch 2/50 - Train Accuracy: 15.71% | Train Loss: 2.6329 | Val Accuracy: 19.47% | Val Loss: 2.5227
Epoch 3/50 - Train Accuracy: 17.60% | Train Loss: 2.5758 | Val Accuracy: 19.47% | Val Loss: 2.5353
Epoch 4/50 - Train Accuracy: 19.27% | Train Loss: 2.5466 | Val Accuracy: 20.93% | Val Loss: 2.4841
Epoch 5/50 - Train Accuracy: 20.56% | Train Loss: 2.5049 | Val Accuracy: 20.47% | Val Loss: 2.4777
Epoch 6/50 - Train Accuracy: 21.36% | Train Loss: 2.4741 | Val Accuracy: 22.27% | Val Loss: 2.4530
Epoch 7/50 - Train Accuracy: 20.73% | Train Loss: 2.4629 | Val Accuracy: 23.07% | Val Loss: 2.4311
Epoch 8/50 - Train Accuracy: 21.89% | Train Loss: 2.4566 | Val Accuracy: 21.67% | Val Loss: 2.4551
Epoch 9/50 - Train Accuracy: 22.29% | Train Loss: 2.4329 | Val Accuracy: 20.47% | Val Loss: 2.4413
Epoch 10/50 - Train Accuracy: 21.89% | Train Loss: 2.4435 | Val Accuracy: 22.27% | Val Loss: 2.4055
Epoch 11/50 - Train Accuracy: 22.02% | Train Loss: 2.4201 | Val Accuracy: 22.67% | Val Loss: 2.4278
Epoch 12/50 - Train Accuracy: 22.27% | Train Loss: 2.4202 | Val Accuracy: 20.73% | Val Loss: 2.4654
Epoch 13/50 - Train Accuracy: 22.36% | Train Loss: 2.4121 | Val Accuracy: 22.13% | Val Loss: 2.4325
Epoch 14/50 - Train Accuracy: 22.62% | Train Loss: 2.4100 | Val Accuracy: 20.93% | Val Loss: 2.4315
Epoch 15/50 - Train Accuracy: 24.40% | Train Loss: 2.3619 | Val Accuracy: 22.13% | Val Loss: 2.4030
Epoch 16/50 - Train Accuracy: 23.33% | Train Loss: 2.3577 | Val Accuracy: 20.40% | Val Loss: 2.4700
Epoch 17/50 - Train Accuracy: 25.11% | Train Loss: 2.3494 | Val Accuracy: 21.60% | Val Loss: 2.4067
Epoch 18/50 - Train Accuracy: 23.82% | Train Loss: 2.3690 | Val Accuracy: 17.67% | Val Loss: 2.4846
Epoch 19/50 - Train Accuracy: 24.78% | Train Loss: 2.3368 | Val Accuracy: 18.33% | Val Loss: 2.4929
Early stopping triggered after epoch 20
Training with params: {'batch_size': 32, 'dropout_rate': 0.3, 'learning_rate': 0.001, 'num_epochs': 50, 'optimizer': 'sgd'}
Epoch 1/50 - Train Accuracy: 14.60% | Train Loss: 2.6030 | Val Accuracy: 23.93% | Val Loss: 2.4317
Epoch 2/50 - Train Accuracy: 22.58% | Train Loss: 2.4098 | Val Accuracy: 27.67% | Val Loss: 2.3272
Epoch 3/50 - Train Accuracy: 25.73% | Train Loss: 2.3208 | Val Accuracy: 29.93% | Val Loss: 2.2639
Epoch 4/50 - Train Accuracy: 29.71% | Train Loss: 2.2232 | Val Accuracy: 30.40% | Val Loss: 2.2212
Epoch 5/50 - Train Accuracy: 31.62% | Train Loss: 2.1615 | Val Accuracy: 30.67% | Val Loss: 2.2023
```

Fig 10. *Example of subsection of hyperparameter tuning process*

I also explored the use of data augmentation. My hope was that this would expose the model to a few new features using CNNs powerful ability to use spatial features and improve the model's ability to generalise. The augmentation performed can be seen below and the result table

```
train_aug = transforms.Compose([
    transforms.Resize((64, 64)),  # Resize to 64x64 (just in case)
    transforms.RandomHorizontalFlip(),  # To expose new spatial features
    transforms.RandomRotation(30),  # Here I will do a random rotationof anywhere between 0,30 degrees
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),  # To try expose new colours
    transforms.ToTensor()  # Here I convert to Tensor between [0,1]
])
```

Fig 11. *Transformations for Data Augmentation*

Note: The augmentation transformations were only performed on the training data. The normal transformations were carried out on test and validation set.

Model (architecture)	Epochs	Batch_Size	Optimiser	Data Augmentation	Early Stopping	Hyperparameter Tuning	Training Acc	Validation Acc	Test Acc	Test Loss
V4	40	128	Adam(lr=0.001) Dropout Rate = 0.3	No	Yes	Yes	85.62%	55.94%	52.80%	1.5647
V4	50	128	Adam(lr=0.001) Dropout Rate = 0.3	Yes	Yes	Yes	74.44%	52.06%	49.27%	1.7153
V5	40	128	Adam(lr=0.01) Dropout Rate = 0.3	No	Yes	Yes	68.36%	50.00%	47.20%	1.9967
V5	50	256	Adam(lr=0.01) Dropout Rate = 0.3	Yes	Yes	Yes	57.18%	43.53%	43.53%	1.8921

Fig 12. *Results of best models V4 and V5 with Data Augmentation and Hyperparameter Tuning*

Clearly there were improvements in the results using parameter optimisation and augmentation which was reducing the gap between training and validation accuracy. However, I wasn't happy with this performance fully, so I decided to explore a different architecture.

3.3.4. Improved Architecture

I then created three new models – V6, V7, V8. There were two main changes in these models' architecture compared to previous

- Use of Global Averaging Polling instead of fully connected Layers
- Batch Normalisation
- Use of a step learning scheduler in hyperparameter tuning to stabilise learning

During my research, I discovered a new technique called Global Average Pooling, which is used in advanced architectures like ResNet. Instead of relying on large, parameter-heavy fully connected layers, this technique uses a simple spatial averaging step. This reduced the number of trainable parameters while still retaining important spatial information. My hope was that this would help with my overfitting problem but maintain results by dealing with the vanishing gradient problem.

ECS8051modelV6: Improving Efficiency with Batch Normalisation and Global Pooling

With ECS8051modelV6, I used the new approaches mentioned above.

- Introduced Batch Normalisation after each convolutional layer to improve convergence and deal with the instability of the learning.
- Went back to three convolutional layers (16, 32, and 64 filters) with ReLU activations.
- Implemented Global Average Pooling instead of fully connected layers
- A final fully connected layer directly mapped the feature outputs to 15 classes.

ECS8051modelV7: Enhancing Feature Extraction and Regularisation

Building on V6, ECS8051modelV7 introduced:

- Returned the fourth convolutional layer (128 filters) to extract deeper features.
- Increased dropout rate (0.2) in the fully connected layer to prevent overfitting.

ECS8051modelV8: Deepening and Refining the Network

To further enhance performance, I revisited ECS8051modelV4 and made the following changes:

- Increased filter sizes
- Added dropout layers (0.3–0.4) to avoid overfitting.
- Replaced Global Average Pooling with Global Max Pooling
- Fully connected layers were reduced to one 512-unit layer

Model (architecture)	Epochs	Batch_Size	Optimiser	Data Augmentation	Early Stopping	Hyperparameter Tuning	Training Acc	Validation Acc	Test Acc	Test Loss
V6	50	32	Adam(lr=0.01, Dropout Rate=0.5)	Yes	Yes Patience=5	Yes	55.22%	51.4%	47.20%	1.7214
V6	40	32	Adam(lr=0.01, Dropout Rtae=0.5)	No	Yes Patience=5	No	66.87%	~50%	49.80	1.6588
V7	40	32	Adam(lr=0.001) Dropout Rate = 0.3	Yes	Yes Patience=3	No	56.00%	51.80%	52.00%	1.6271
V7	100	32	Adam(lr=0.001) Step Scheduler(size-10)	Yes	Yes Patience=3	Yes	60.51%	58.80%	58.67%	1.3467
V7	100	32	Adam(lr=0.001) Step Sceduler(size-10)	No	Yes Patience=3	Yes	66.98%	63.53%	62.27%	1.2560
V8 (Combination of V7 and V4)	80	32	Adam(lr=0.01)	Yes	Yes Patience=3	Yes	77.44%	51.84%	49.93%	1.5542

Fig 13. *Results of improved models using Batch Normalisation and Global Average Pooling*

The results will be discussed more in discussion section below, but these changes had a positive effect on results. The fourth convolutional layer was required to improve results, but new changes had positive effects, with the best model achieving 62% accuracy on the test set while maintaining a steady learning rate and little overfitting.

4. Discussion

4.1 Significant Findings Handcrafted Features

- ORB was significant more efficient than SIFT
- ORB was more compatible with fisher vector than SIFT which struggled a lot more compared to its Kmeans scores
- The resizing for ORB made a big difference for feature extraction
- The image enhancement was critical for SIFT otherwise it couldn't access any features.
- SIFTs robustness to size and scale wasn’t maximised by this dataset due to smaller images
- Both performed well with number of words between 50-100 for BoW model but after that there appeared to be overfitting as test scores declined.
- GMM needed a much smaller number of clusters
- The ability for GMM to calculate extra statistical information provided a better overall metric when paired for ORB than just using counts in the BoW model

4.2 Significant Findings NNs

- More complex models struggled with this data
- Less layers and parameters yielded better results
- Hyperparameter tuning and early stopping increased result significantly
- SGD was far better optimiser than Adam
- Data augmentation was not successful
- Lacked effective feature extraction
- There was overfitting as shown in plot of training vs validation loss below

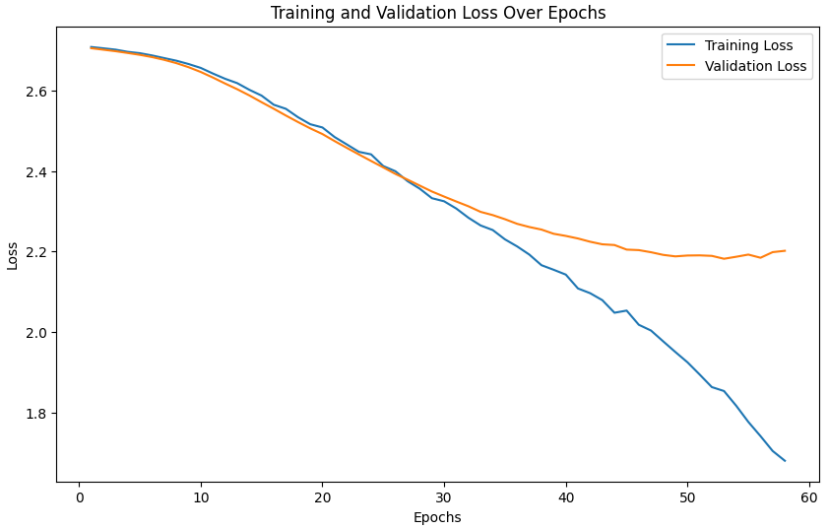

Fig 14 . *Training vs Validation Loss V2 Best Model*

4.3 Significant Findings CNNs

Some of the key takeaway from the CNN analysis were as follows:

- Some of my deeper more complex model architectures were not well suited to this specific dataset given its size and image clarity.
- Simpler, less dense models outperformed some of the others
- Early stopping was effective in preventing overfitting throughout.
- Validation scores reflected test set scores which showed a representative and effective validation set.
- The use of data augmentation and dropout paired with global max pooling helped prevent overfitting and created models that could generalise well, and I found that it outperformed dense fully connected layers.
- Adam optimiser's ability to converge quickly and use of adaptive learning rates was better suited to this low resource dataset.
- Originally, before the addition of a scheduled learning rate, learning rates were too high and scores were fluctuating which signified oscillation and unstable training process.
- The final model showed a steady decrease in both test and training loss and avoided this unstable learning and oscillation whilst also not overfitting.

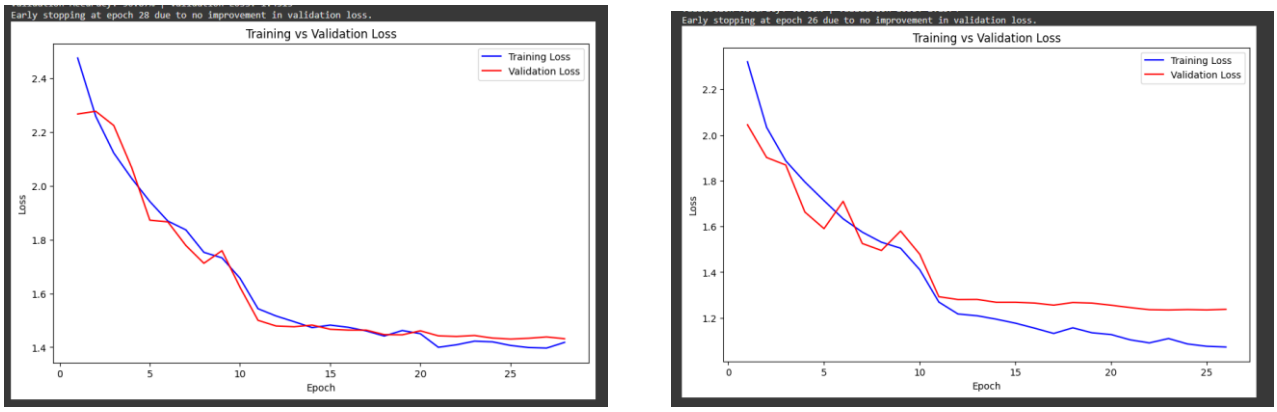

Fig 15 & 16. *Training vs Validation Loss for V7* ***Augmented (Left) and*** *Normal* ***(Right)***

- While the model V7 trained on normal data showed better test scores the plot of the validation and training loss showed slight overfitting whereas the augmented version showed a much better fit thanks to the addition of new data and therefore would hint at being the better model for an increased and future dataset.
- Shows the importance of analysing a model thoroughly rather than selecting just best accuracy scores.
- The improved architecture in V7 allowed me to deal with the vanishing gradient problem and led to a smoother learning process which improved results.

5. Comparison of Approaches

To compare the three approaches, I took the best model from each (and two from CNN)

Model	Epochs/Number of Clusters	Architecture	Hyperparameters	Test Accuracy
ORB Fisher Vector	50	Fitted Gaussian Mixture Model Classified using LinearSVC() Output (15 classes)	Max_iter=1000, C=0.1	24.09%
LinearNNV2	60	Input Size - 64 x 64 x 3 Flattening Layer - (12288) Four Hidden Layers = [1024, 512,256,128] ReLU Activation Dropout - 30% Output(15 classes)	Batch_size = 128 Optimiser = SGD Learning Rate=0.001 Momentum = 0.9 Loss = CrossEntropy Dropout Rate=0.5 Early Stopping Patience = 5	33.00%
CNNV7	100	Input (64×64×3) → Conv1 (16 filters, 3×3, ReLU, BN, MaxPool 2×2) → Conv2 (32 filters, 3×3, ReLU, BN, MaxPool 2×2) → Conv3 (64 filters, 3×3, ReLU, BN, MaxPool 2×2) → Conv4 (128 filters, 3×3, ReLU, BN, MaxPool 2×2) → Global Avg Pool → Fully Connected Layer (128 → 15) → Dropout (20%) → Output (15 classes). BN (Batch Normalisation)	Batch_size= 64 Optimiser = Adam Learning Rate=0.001 Scheduler = StepLR(10,0.1) Loss=CrossEntropy Dropout Rate = 0.2 Early Stopping Patience=3 Data Augmentation=No	62.27%
CNNV7 Data Augmentation	100	Same as above	Same as above except Data Augmentation=True	58.67% (Neater Train/Val loss plot)

Fig 17. *Results of Best Model from Each Approach*

I noticed a clear improvement in accuracy on the test set as the complexity of the models increased. The handcrafted features in the simpler models struggled to capture important patterns, whereas the CNN's ability to learn spatial relationships made a significant difference.

What surprised me most was the jump in accuracy from the fully connected NN to the CNN. The fully connected layers introduced a huge number of parameters, making the model prone to overfitting and picking up noise. On the other hand, the convolutional layers with smaller kernel sizes allowed the model to learn more informative hierarchical features. My V7 model struck a balance between having enough layers to get this information but not suffering from the curse of having to many parameters.

A big change I noticed was batch normalisation helped keep training stable by making sure activations stayed within a good range. Discovering the global average pooling was a massive benefit to performance because it reduced the number of parameters a lot compared to fully connected layers while keeping important spatial information. This helped prevent overfitting and made the model more reliable than the standard NN and handcrafted features

To analyse these results in further detail I looked at the confusion matrix for each:

5.1 Error analysis

SVC

Test Classification Report:

	precision	recall	f1-score	support
0	0.26	0.21	0.23	100
1	0.18	0.20	0.19	100
2	0.22	0.27	0.24	100
3	0.07	0.01	0.02	100
4	0.19	0.18	0.19	100
5	0.24	0.36	0.28	100
6	0.09	0.05	0.06	100
7	0.26	0.51	0.35	100
8	0.15	0.10	0.12	100
9	0.27	0.38	0.31	100
10	0.16	0.15	0.16	100
11	0.27	0.35	0.30	100
12	0.21	0.20	0.21	100
13	0.25	0.16	0.20	100
14	0.22	0.18	0.20	100
accuracy			0.22	1500
macro avg	0.20	0.22	0.20	1500
weighted avg	0.20	0.22	0.20	1500

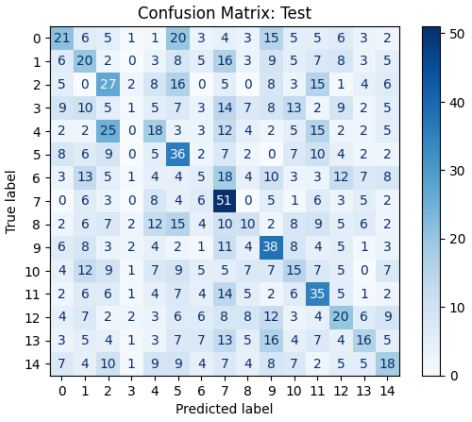

Fig 18&19. *Classification Report and Confusion Matrix for Best Handcrafted Model*

Linear Neural Network

Classification Report:

	precision	recall	f1-score	support
0	0.25	0.20	0.22	100
1	0.27	0.16	0.20	100
2	0.49	0.72	0.58	100
3	0.08	0.03	0.04	100
4	0.31	0.38	0.34	100
5	0.23	0.23	0.23	100
6	0.09	0.02	0.03	100
7	0.32	0.36	0.34	100
8	0.37	0.30	0.33	100
9	0.40	0.44	0.42	100
10	0.25	0.29	0.27	100
11	0.39	0.46	0.42	100
12	0.29	0.37	0.33	100
13	0.48	0.62	0.54	100
14	0.30	0.42	0.35	100
accuracy			0.33	1500
macro avg	0.30	0.33	0.31	1500
weighted avg	0.30	0.33	0.31	1500

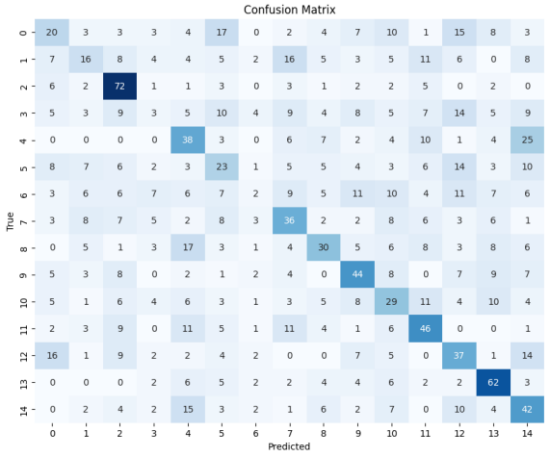

Fig 20&21. *Classification Report and Confusion Matrix for Best Linear Neural Network*

CNN trained on normal data

Classification Report:

	precision	recall	f1-score	support
0	0.54	0.58	0.56	100
1	0.55	0.59	0.57	100
2	0.81	0.87	0.84	100
3	0.44	0.21	0.28	100
4	0.59	0.73	0.65	100
5	0.62	0.62	0.62	100
6	0.47	0.45	0.46	100
7	0.61	0.66	0.63	100
8	0.73	0.69	0.71	100
9	0.71	0.61	0.66	100
10	0.63	0.74	0.68	100
11	0.79	0.80	0.80	100
12	0.61	0.66	0.63	100
13	0.67	0.62	0.64	100
14	0.50	0.51	0.51	100
accuracy			0.62	1500
macro avg	0.62	0.62	0.62	1500
weighted avg	0.62	0.62	0.62	1500

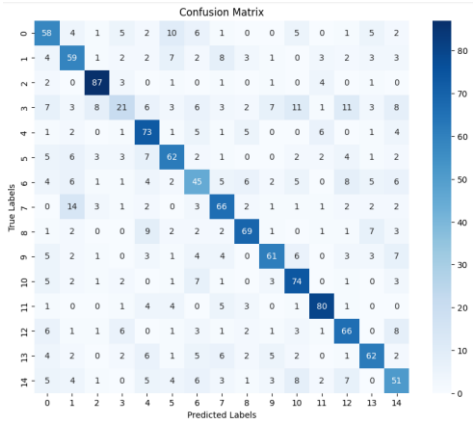

Fig 21&22 . *Classification Report and Confusion Matrix for Best Convolutional Neural Network*

CNN trained on augmented data

Classification Report:

	precision	recall	f1-score	support
0	0.45	0.49	0.47	100
1	0.46	0.46	0.46	100
2	0.75	0.85	0.80	100
3	0.35	0.07	0.12	100
4	0.56	0.76	0.64	100
5	0.47	0.66	0.55	100
6	0.34	0.22	0.27	100
7	0.59	0.60	0.60	100
8	0.62	0.72	0.66	100
9	0.67	0.62	0.65	100
10	0.61	0.59	0.60	100
11	0.66	0.77	0.71	100
12	0.58	0.51	0.54	100
13	0.59	0.65	0.62	100
14	0.47	0.44	0.45	100
accuracy			0.56	1500
macro avg	0.54	0.56	0.54	1500
weighted avg	0.54	0.56	0.54	1500

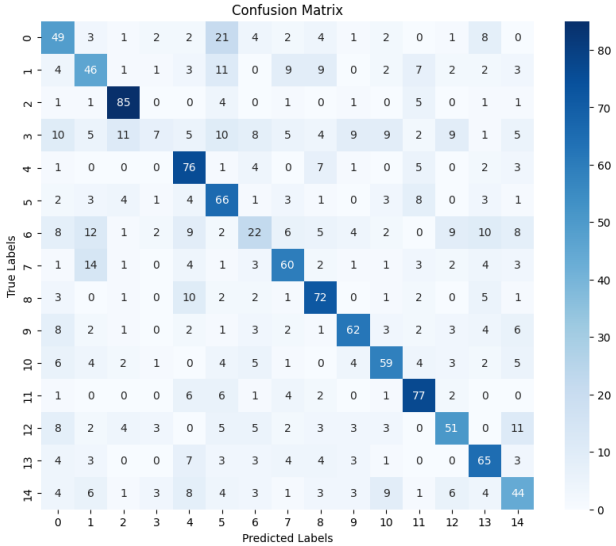

Fig 22&23. *Classification Report and Confusion Matrix for Best Convolutional Neural Network with Data Augmentation*

From simply looking at the confusion matrices above, one can see very clearly that the diagonal through the middle is far darker for the CNN models than the linear and handcrafted features showing a superior predictive power. To analyse further however, I want to take specific pattern that I notice.

Liner NN vs Handcrafted

In the handcrafted model, I noticed that labels 5 and 7 had significantly higher recall than precision, indicating that the model defaulted to these labels when uncertain. This suggests it had identified and relied heavily on specific features associated with guinea pigs and spiny lobsters.

In contrast, the Linear NN produced a more balanced distribution of predictions, with similar precision and recall scores, showing less bias toward labels. It also performed significantly better on certain labels, such as 2 and 13, demonstrating that it had learned useful features.

Clearly the handcrafted model relied on a few dominant features, while the Linear NN's learned features at least led to more consistent predictions, showing an improvement as it was learning patterns rather than merely guessing.

Linear NN vs CNNs

Beyond the obvious improvement in accuracy, the CNN outperformed the Linear NN in other ways. One way to see this was by looking at the mistakes rather than just the correct predictions. The Linear NN often made errors like predicting chimpanzee (4) as backpack (14) and vice versa, showing that the features it used weren't relevant or informative as these two items do not resemble each other.

The V7 CNN, however, avoided these major mistakes. Its main error was predicting boa constrictor (1) as spiny lobster/crawfish (7). While not ideal, the images below show that these classes share some visual similarities. This suggested the CNN was picking up on more useful features and making better-informed predictions, highlighting the advantages of the convolutional layer and global average pooling for feature extraction.

Fig 24. *Crawfish Training Image*

Fig 25. *Boa Constrictor training image*

Test Case to show improvement in all

One test case I used to highlight the benefits of my CNN model was its performance on label 3 (Bucket and Pail). This label consistently produced the worst metrics across all models. After examining the images, it became clear why—many contained distrctions. Handcrafted features, for example, tended to focus on elements like eyes rather than identifying the bucket itself.

Fig.26 *Bucket training image*

My CNN model performed significantly better than the other approaches, making far more correct predictions. This demonstrated the advantages of the convolutional architecture in extracting useful features and how spatial data gives it a clear edge over the alternative methods I used.

6. Conclusion

This assignment was a particularly interesting one. As shown above, I started with handcrafted features, which performed the worst, and progressively refined my approach to a CNN that achieved 62% accuracy—a strong result given the dataset's size and quality. The report and process highlight the improvement of the Linear NN over handcrafted features, but more importantly, the significant advantage of convolutional spatial data over a 1D input. Techniques such as early stopping, batch normalisation, dropout, and global average pooling were crucial in preventing overfitting and building my effective V7 model. As a next step, I would fine-tune a large pre-trained model on this dataset to evaluate its performance. Given the limited data available, this approach could yield even better results.

7. References

1. Brownlee, J. (2019). A Gentle Introduction to Pooling Layers for Convolutional Neural Networks. [online] Machine Learning Mastery. Available at: https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/

2. Doshi. K. (2021). Batch Norm Explained Visually — How it works, and why neural networks need it. [online] Medium. Available at: https://towardsdatascience.com/batch-norm-explained-visually-how-it-works-and-why-neural-networks-need-it-b18919692739.GeeksforGeeks (2024).

3. ResNet18 from Scratch Using PyTorch. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/resnet18-from-scratch-using-pytorch/.Google (2019).

4. Google Colaboratory. [online] Google.com. Available at: https://colab.research.google.com/.PyTorch (2023).

5. PyTorch. [online] Pytorch.org. Available at: https://pytorch.org/.Sarin, S. (2020).

6. VGGNet vs ResNet (The Vanishing Gradient Problem). [online] Medium. Available at: https://towardsdatascience.com/vggnet-vs-resnet-924e9573ca5c.Scikit-image.org. (2025).

7. Fisher vector feature encoding — skimage 0.25.1 documentation. [online] Available at: http://scikit-image.org/docs/stable/auto_examples/features_detection/plot_fisher_vector.html#sphx-glr-auto-examples-features-detection-plot-fisher-vector-py.scikit-image.org. (n.d.).

8. scikit-image: Image processing in Python — scikit-image. [online] Available at: https://scikit-image.org/.www.run.ai. (n.d.). PyTorch ResNet. [online] Available at: https://www.run.ai/guides/deep-learning-for-computer-vision/pytorch-resnet.