CV1 Lab Final- Part 2
Group 30

Pradyut Nair	Christina Isaicu	Akshay Sardjoe Missier	Benjamin Shaffrey
Student ID: 15558169	Student ID: 15050025	Student ID: 12708178	Student ID: 13853163

October 18, 2024

1 Section 1-7: Image Classification on CIFAR-100

In this section we compare the classification performance between a Convolutional Neural Network (CNN) and a fully connected Multi Layer Perceptron (MLP). The networks are trained and tested on the CIFAR-100 dataset, which contains 32x32 pixel RGB images, with 100 different classes. For a visualization of samples from the dataset, see the Appendix 1.10.

1.1 MLP Architecture

Prior to any tuning, the initial architecture of the MLP is instantiated as follows: Two fully connected layers with ReLU activation function in-between. The input has 3 channels, one for each RGB colour channel, of size 32x32 corresponding to the pixel dimensions of the image. The hidden layer is of size 1024, and the network outputs 100 values corresponding to the number of classes.

1.2 CNN Architecture

Prior to any tuning, the initial CNN architecture we use follows the LeNet-5 model architecture created by Lecun et al. (1998) with some modifications to ensure compatibility with the CIFAR-100 dataset.

The network architecture is comprised of three convolutional layers with a kernel size of 5, and two fully connected layers. The input to the network is the same as for the MLP above. Figure 1 illustrates the architecture used. For a full breakdown, see the description by Lecun et al. (1998).

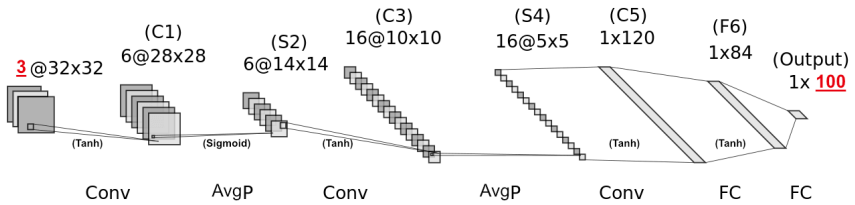


Figure 1: Initial CNN architecture adapted from Lenet-5 Lecun et al. (1998) (image adapted from DL-Visuals repository by @dvgodoy Godoy). Underlined values in red indicate our modifications.

Layer F6 has 10 164 trainable parameters, calculated by multiplying the number of input by output features, added to the number of bias values (which corresponds to the size output features). 120 x 84 + 84 = 10164.

1.3 Initial hyperparameter search

To direct our hyperparameter search efforts, we used a systematic pointwise approach where all parameters are kept fixed to the baseline values, and one parameter is adjusted at a time. This method was chosen as a more computationally feasible alternative to grid-search, and a time-sensitive option in comparison to more sophisticated hyperparameter search methods. The parameters we searched were chosen according to common values used in standard deep learning practice. Where applicable, we tested values that are higher and lower than the baseline. The following table lists the baseline hyperparameters, and those included in the search. Initial results of the sweep were used to draw conclusions about optimal combinations for further tuning.

Table 1: Hyperparameters searched for MLP and CNN Models

Hyperparameter	**Baseline Value**	**Tuning Values**
Learning rate	1.0E-03	1.0E-02, 1.0E-04
Batch size	512	32, 64, 128, 256, 1024
Number of epochs	100	10, 50, 200, 500
Optimizer	AdamW	SGD, RMSProp
Weight decay	1.0	0.8, 0.6, 0.4, 0.2, 0.01
Transform function	Normalize	Normalize (CNN only), Random horizontal flip, Random rotation (10°), Color Jitter
Activation function	Sigmoid & Tanh (LeNet-5)	ReLU

Table 2: Original values and additional hyperparameter values searched for pointwise tuning
Adam params: β=(0.9, 0.999) – **SGD params:** learning rate=1e-3, momentum=0.9,weight decay=1e-2 – **RMSProp params:** learning rate=1e-3, α=0.99, ϵ=1e-8, weight decay=1e-2

1.4 Hyperparameter Evaluation

The metrics measured are as follows: Accuracy, accuracy per epoch, loss per epoch, precision, recall, F1 score. Figures 2 and 3 show accuracy comparisons for different hyperparameter values compared to the baseline accuracy.

Hyperparameter analysis for both models revealed the following insights: Higher performance compared to the baseline was observed in lower weight decay values, inclusion of additional transformations, and additional layers (extra_layers).

In the MLP, a lower learning rate, more or fewer batch sizes, as well as more or fewer epochs, improved performance. In the CNN, the baseline learning rate outperforms lower or higher values. Changes to the batch size and number of epochs don't offer significant improvements. Improvements resulting from changes to the optimizers will be discussed in section 1.5 and to the architecture in section 1.7.

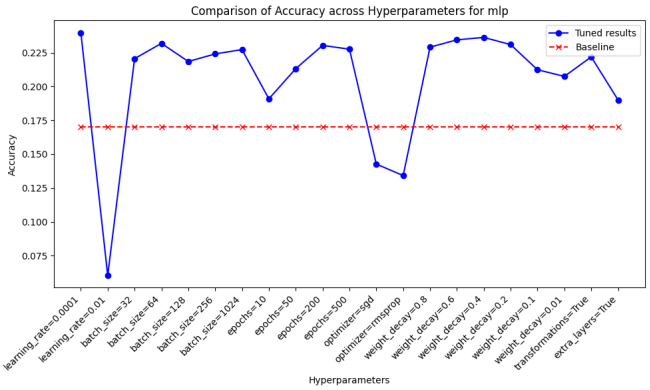


Figure 2: Tuning MLP

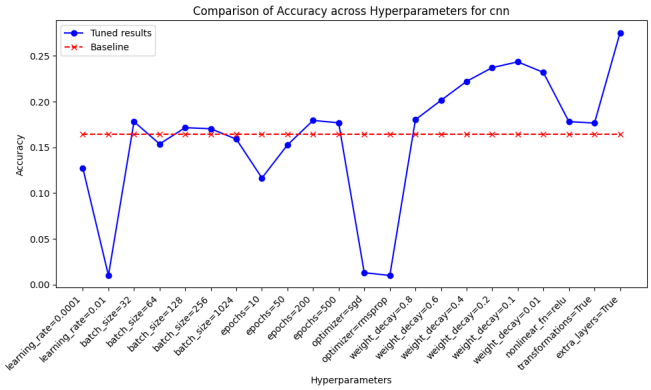
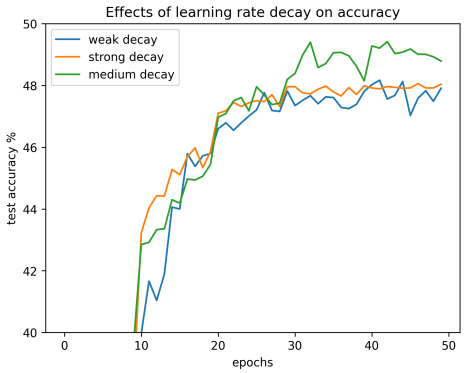


Figure 3: Tuning CNN

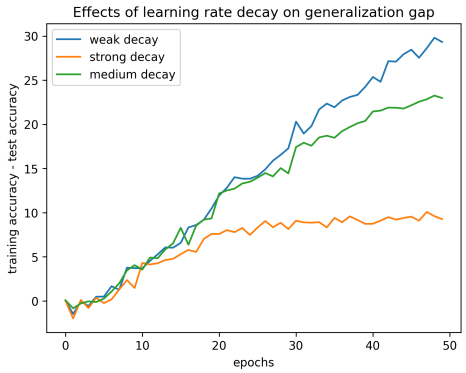
1.5 Tuning the CNN: Comparing optimizers SGD and ADAM

An important aspect of finding the optimal CNN architecture is deciding on an optimizer for the learning process. ADAM and stochastic gradient descent (SGD) are both commonly used in deep learning tasks, and therefor we tested both. Previous research has shown that ADAM typically achieves faster convergence, whereas SGD can offer better generalization with careful parameter tuning. Kingma (2014) Low performance of different optimizers in the initial hyperparameter sweep (compared to the baseline) prompted further investigation into the specific hyperparameter adjustments required for optimal performance.

To get the most out of SGD, several decisions have to be made about its parameters. These include learning rate, momentum and weight decay. The learning rate is also dependent on the batch size chosen for the data loader. This is because the batch size affects the accuracy of the calculated gradients, and more accurate gradients can allow larger steps without become too unstable. Additionally, the batch size can also improve the generalization of the model, this is described in more detail later on. We have found that a learning rate of 0.003 is relatively stable for batch sizes of at least 64. However, a consistent learning rate throughout the learning process can still become unstable when approaching local minima. In order to more improve the stability in this scenario, it can help to have a decaying learning rate. This allows the step size to decrease gradually throughout the learning process and this can prevent overshooting the local minima. The effects of different strengths of decaying learning rates can be seen in figure 18.



(a) Effect of learning rate decay on test accuracy

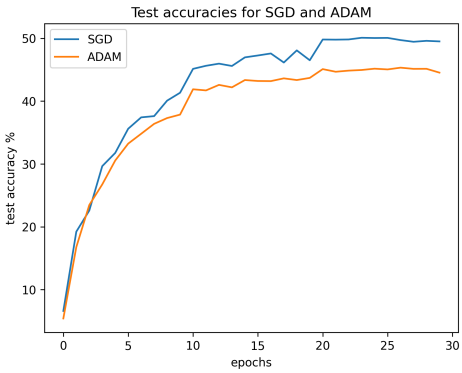


(b) Effect of learning rate decay on generalizablity. A lower value indicates better generalization.

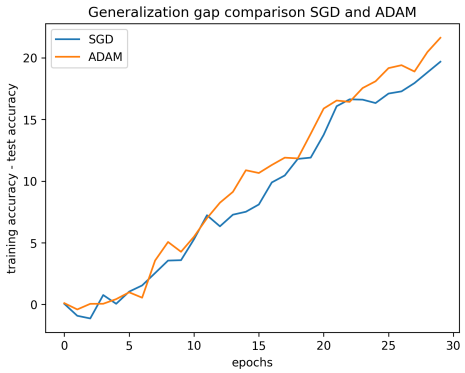
Figure 4: Comparison of the effects of learning rate decay

It can be observed that a strong decay in learning rate can help the model not to overfit to the training data, as seen in the small generalization gap. However, the aggressive decay also stops the model from learning in the later epochs. On the other end of the spectrum, a weaker decay causes a large generalization gap, indicating that the model is overfitting to the training data. The medium decay offers a compromise, with the best accuracy, despite a sizeable generalization gap. SGD in its standard form only uses the currently calculated gradient to determine the next step direction. Momentum is used incorporate past gradients to decide the next step. This can accelerate learning if the gradients are consistent, but it can also stabilize learning when the gradients are inconsistent. Typically, momentum allows for faster convergence compared to SGD without momentum. For our model, we decided on a momentum value of 0.9, which means that 90% of the previous velocity is carried over. This is value is commonly chosen as it strikes a good balance between accelerating learning and preventing overshooting.

Lastly, weight decay is a form of L2 regularization. This penalizes large weights and pushes the weights to remain small. This is turn prevents overfitting and improves generalization. Combining all the previously mentioned parameters results in a finely tuned implementation of SGD. Figure 5 shows a comparison between SGD and ADAM under the same learning rates and weight decay trained on 30 epochs. From this figure, it can be seen that our implementation of SGD outperforms ADAM both in accuracy as well as generalizability.



(a) Test accuracy for SGD and ADAM



(b) Generalization gap for SGD and ADAM

Figure 5: Comparison of SGD and ADAM

1.6 Tuning the CNN: Covariate shift and Overfitting

CNNs are highly sensitive to changes in the input data distribution between training and testing stages. An example of this would be having training images taken in indoor lighting, whilst having testing images with natural outdoor lighting. Such a distribution shift of the input data is known as covariate shift, and it can limit the model's ability to generalize well. This happens because the model will overfit on the features of the training data, that do not actually represent the underlying relationship of the data, such as the indoor lighting in the previous example. Covariate shift is more likely to occur when training models on small datasets, because they will generally only cover a small section of the real input distribution.

CIFAR-100 is a relatively small dataset, hence the risk of covariate shift is negligible. Covariate shift is typically managed with data augmentation and batch normalization. Data augmentation is the process of adding transformations to the training data in order to make the data more diverse, and thus cover a larger portion of the real-world distribution of the data. Figure 6 depicts the results of the transformations we

applied to the training data. The following transformations were used:

- **Random horizontal flip**. This introduces more variability for object orientations in the dataset.
- **Color jitter**. This randomly changes the the color characteristics of images and can account for lighting conditions and camera specifics in the training data.
- **Random rotation**. By randomly rotating images in the training dataset objects can appear in more orientations than originally present in the dataset. A maximum rotation of 10° was used.
- **Normalize**. By normalizing the images, the mean intensity is set to zero, and the intensity values are normalized to the range $[-1, 1]$. This can prevent large gradients and therefor stabilize learning. It can also allow for faster convergence.

Figure 6: Examples of transformed images.

Aside from data augmentation, the model itself can also be made robust to covariate shift. This is done by introducing batch normalization layers in between the convolution layers and their activation functions. Similarly to the normalization transform described in the previous section, batch normalization normalizes the values that resulted from the convolution layer to have a mean of zero and a standard deviation of one. This ensures that the distribution of the activations are consistent throughout the convolution layers of the network, which reduces the risk of covariate shift. Additionally, this normalization vastly reduces the risk of vanishing or exploding gradients, which will also benefit the learning process.

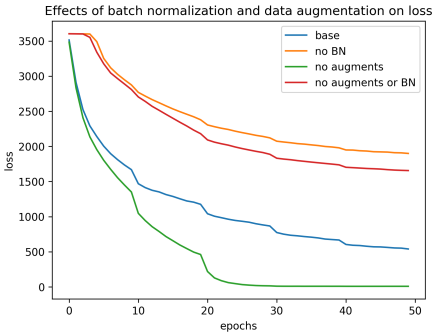


Figure 7: Loss per epoch with and without covariate shift minimizing techniques

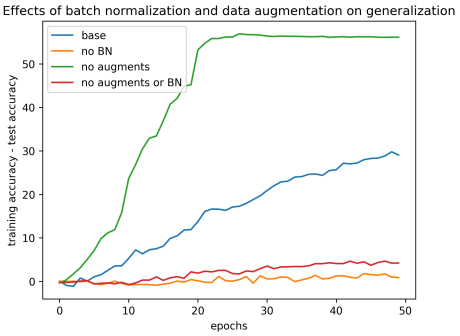


Figure 8: Generalization gap with and without covariate shift minimizing techniques

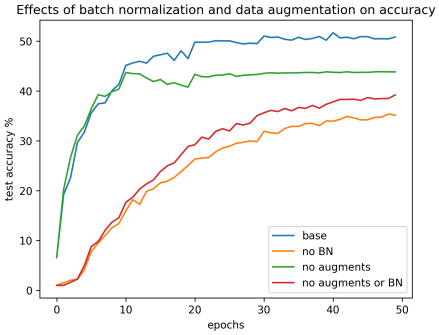


Figure 9: Test accuracy of models with and without covariate shift minimizing techniques.

From figures 7, 8 and 9 it can be observed that a model without data augmentations will overfit severely to the training data. This is indicating by the combination of minimal loss and a large generalization gap. These plots also show that the models without batch normalization preform far worse than those with, in terms of accuracy. This is an expected result because of the learning benefits of batch normalization described in the previous section. Interestingly, the models without batch normalization show the smallest generalization gaps. For our network, we considered test accuracy to be the most important metric. And as such we concluded that the model with both data augmentation and batch normalization preforms best.

As a final effort to combat our model from overfitting to the training data, we added dropout to the first two fully connected layers of our network. Dropout works by randomly turning off a subset of neurons at every training iteration. This forces neurons to independently learn richer representations of the data, because they can rely less on other neurons. This means that every training iteration, a different set of neurons are trained. When the model is evaluated, all neurons are turned on. This effectively preforms model averaging of sorts for the fully connected layers and can help reduce overfitting. For our model we have chosen a dropout probabilty of 55%, meaning that on average 45% of the neurons in the first two fully connected layers are trained per training iteration. Figure 10 shows the test results for generalizability with and without dropout. It can be seen that dropout does help to shrink the generalization gap, albeit only slightly.

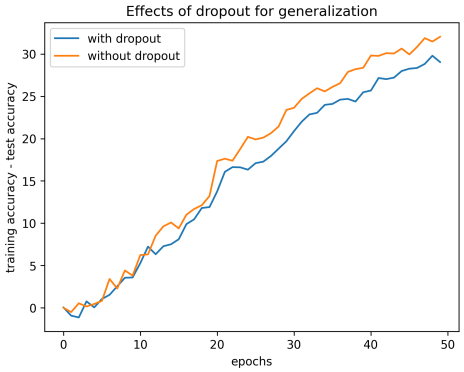


Figure 10: Comparison of the generalization gap with and without dropout (55%)

1.7 Tuning the models: Modified architecture

In addition to the batch normalization layers and dropout layers discussed in the section above, we added 2 convolutional layers and one fully connected layer to the CNN architecture, in order to promote better

generalization and learning of abstract features.

The original architecture of LeNet5 was was designed for the MNIST dataset, which is not optimal for our purposes. For example, the last hidden linear layer outputs 84 channels which is designed to represent "a stylized image of the corresponding character class drawn on a 7x12 bitmap" Lecun et al. (1998).

Additionally, standards have changed in the years since Lenet5 was released, so we updated the depth of the layers, and size of the convolutional kernels in accordance with modern ConvNet standars. For similar reasons, and to promote improved performance, we replaced the Sigmoid and Tahh activation functions with ReLU. Figure 11 illustrates the modified architecture of the CNN.

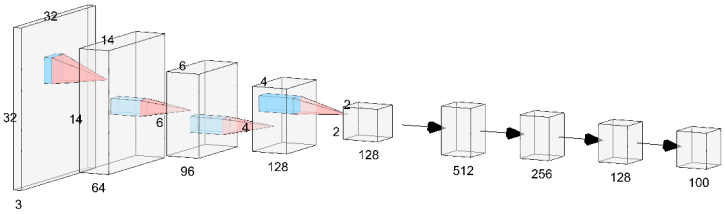


Figure 11: Architecture of our tuned CNN.

For the MLP, two additional linear layers with the same number of input and output features, with ReLU activation in between, were added to increase the depth of the network, and therefore performance. The feature sizes were kept large to allow for learning of complex relationships.

1.8 Results: Comparison of performance between the baseline and tuned models

The following table lists the hyperparameters for both baseline models compared to the tuned models, for both CNN and MLP.

- **Learning rate**: Baseline: 1.0E-03 – CNN Tuned: 3.0E-03 with decay (step=10, γ=0.5) – MLP Tuned: 1.0E-04
- **Batch size**: Baseline: 512 – CNN Tuned: 64 –MLP Tuned: 64
- **Number of epochs**: Baseline: 100 – CNN Tuned: 50 – Tuned: 100
- **Optimizer**: Baseline: AdamW – CNN Tuned: SGD (momentum = 0.9) – MLP Tuned: AdamW
- **Weight decay**: Baseline: 1 – CNN Tuned: 1.0E-03 – MLP Tuned: 0.8
- **Transform function**: Baseline: Normalize – CNN Tuned: Normalize Random horizontal flip, Color Jitter, Random rotation (10°) – MLP Tuned: Random horizontal flip, Random rotation (10°), Random resized crop
- **Activation function**: CNN Baseline: Sigmoid & Tanh – MLP Baseline: ReLU – CNN Tuned: ReLU – MP Tuned: ReLU
- **Additional layers**: Baseline: False – CNN Tuned: True – MLP Tuned: True

Evaluation metrics for the tuned and baseline models are listed below. To keep the analysis concise, our report focuses on accuracy as the latter metrics do not provide significantly different insights.

Table 3: Evaluation Metrics for MLP and CNN (Baseline vs. Tuned)

Model	Accuracy	Precision	Recall	F1 Score
MLP Baseline	0.2284	0.2283	0.2284	0.2190
MLP Tuned	0.2563	0.2637	0.2563	0.2506
CNN Baseline	0.2016	0.1837	0.2016	0.1793
CNN Tuned	0.5169	0.5154	0.5169	0.5135

Table 3, Fig. 12, and Fig. 14 show that the accuracy for both tuned models increased significantly, which is in line with our expectations given our reasoning for the hyperparameter selection provided in the above sections. Figures 13 and 15 show that the loss is lower for the tuned models than the baseline models.

Figure 12: Accuracy MLP

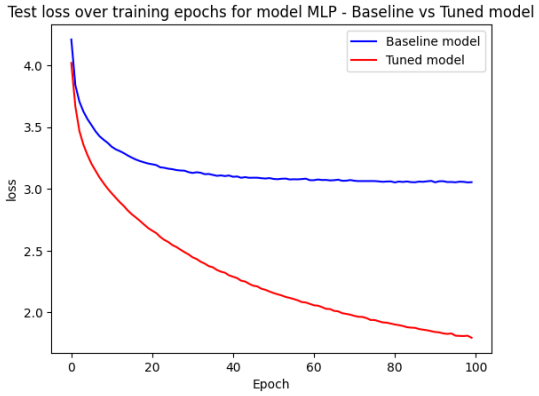


Figure 13: Loss MLP

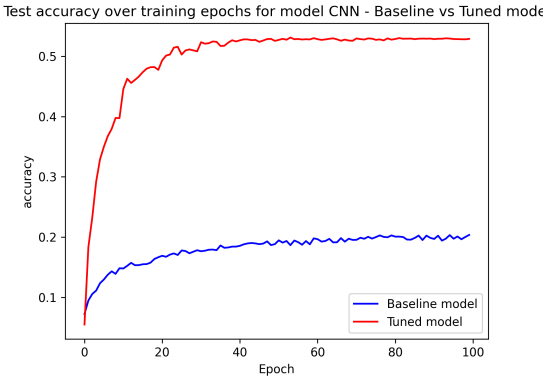


Figure 14: Accuracy CNN

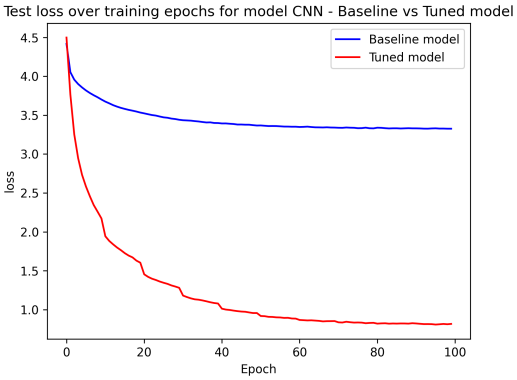


Figure 15: Loss CNN

Figures 16 and 17 below display the accuracy for the top 5 and bottom 5 accuracies per class for each model, showing that there is variation in model performance between different classes. The classes Plain and Road are in the top 5 for both classes, while Otter and Mouse are in the bottom five. Comparing the top performances, it seems like the MLP can learn simple colours and shapes, such as round orange oranges. The CNN can learn slightly more complex categories that still have distinctive silhouettes, such as Motorcycle (see Appendix 1.10). Plain seems like one of the easiest classes to categorize because it often features a sharp contrasting horizontal line between sky and land. Both models perform poorly on mammals that often have in-distinctive shapes and colourings.

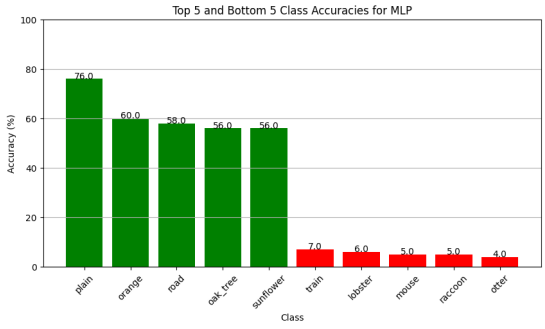


Figure 16: Class accuracies for MLP

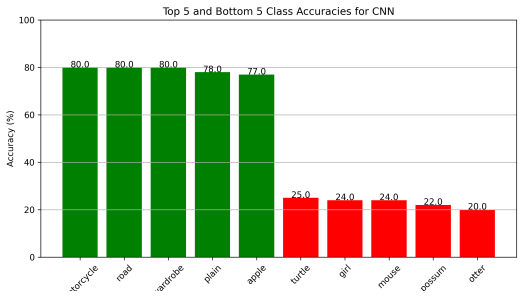


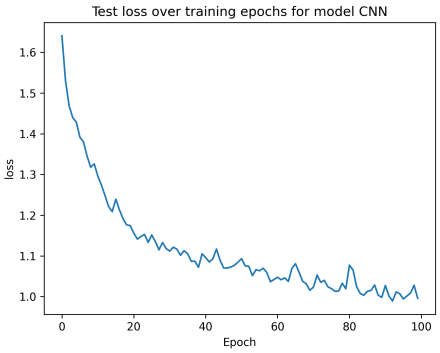
Figure 17: Class accuracies for CNN

Section 8-10: Fine-tuning ConvNet on STL-10

1.9 Section 8 and Section 9

We then wished to fine-tuned the model trained on the CIFAR-100 dataset with data from a subset of the STL-10 dataset. As before we visualised samples from the dataset, but now just for a subset containing only 5 classes. Similarly we wrote a custom dataset loader class, namely STL10_loader that allowed us to filter the dataset to only given 5 classes and to parameterise the transforms that we would then apply to the data. We then proceeded to iterate over the parameters in the trained model, obtaining a dictionary of the parameters in each layer that excluded the final layer. We initialised a new CNN model with an output size of 5 rather than 100 and we loaded the parameter values from our dictionary into the state dictionary of the new model and trained it with the same hyperparameter values as in the case of the tuned CNN from before. The resulting accuracy and loss curves can be seen in Figures 18a and 18b respectively.

(a) Accuracy for fine-tuned model



(b) Loss for fine-tuned model

Figure 18: Learning curves for fine-tuned model

1.10 Achieving maximum performance for STL10

For the final part of this project we attempted to achieve maximum accuracy on a subset of the STL10 dataset, with five classes. For architecture we turned to our previous CNN designed for CIFAR-100 (). The images in STL10 are of size 96×96, whereas those in CIFAR-100 are 32×32. To account for this additional transforms are added to the data augmentation step. In addition to the transforms described in section 1.6, the following transforms are used:

- **Resize**. This transform downsamples the images to 32×32. This ensures that all the convolutions in our previous CNN will have the correct shape.
- **Sharpen**. The images are sharpened to appear more like the images in CIFAR-100

These additional transformations are applied to both the training and test data. Figure 19 depicts the resulting images after these transforms.

Figure 19: Examples of transformed images in STL10

Now that the images in STL10 are the same size as those in CIFAR-100, the CNN architecture can be modified to predict 5 classes. This is done by adding a fully connected layer at the end of the network that maps to 5 outputs. The resulting architecture is shown in figure 20.

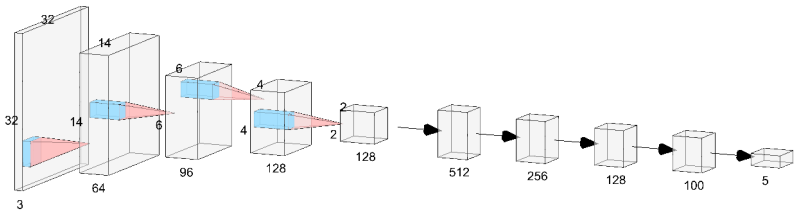


Figure 20: Learning statistics for the retrained model

Because the architecture is mostly the same as our previous CNN, we can load in our pre-trained model for CIFAR-100 and retrain it for the new dataset. This approach works well because the STL10 dataset is transformed to be very similar to the CIFAR-100 dataset. The maximum achieved accuracy was 58.25%, and the loss and accuracy per epoch are shown in figure 21.

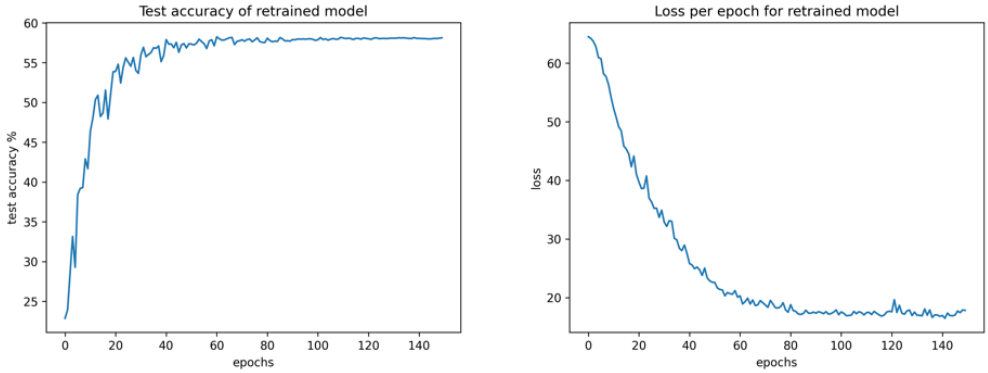


Figure 21: Learning statistics for the retrained model

Appendix

Visualization of CIFAR-100 dataset

Including all superclasses and 5 examples from each of their corresponding subclasses that demonstrate the diversity of the dataset.

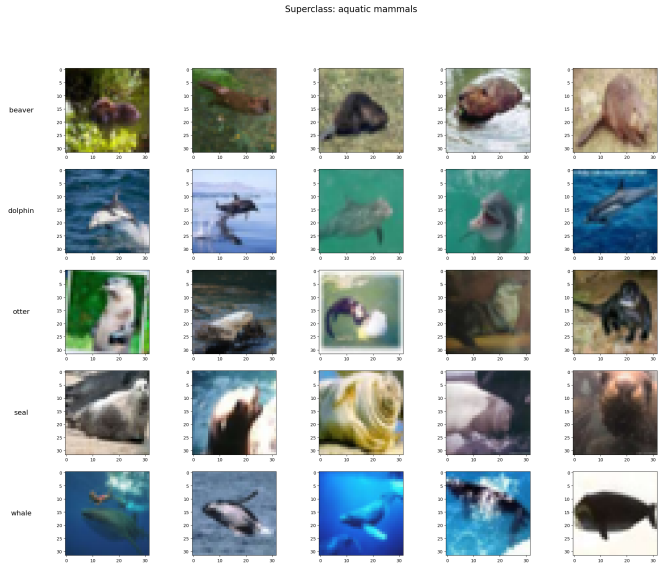


Figure 22: Superclass: Aquatic mammals

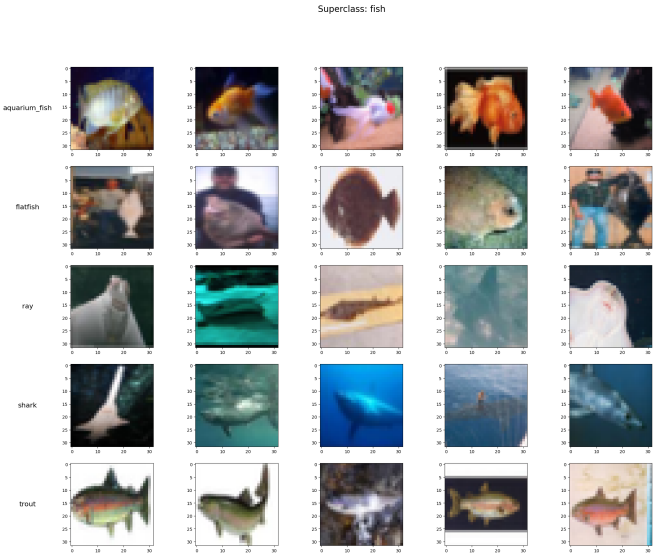


Figure 23: Superclass: Fish

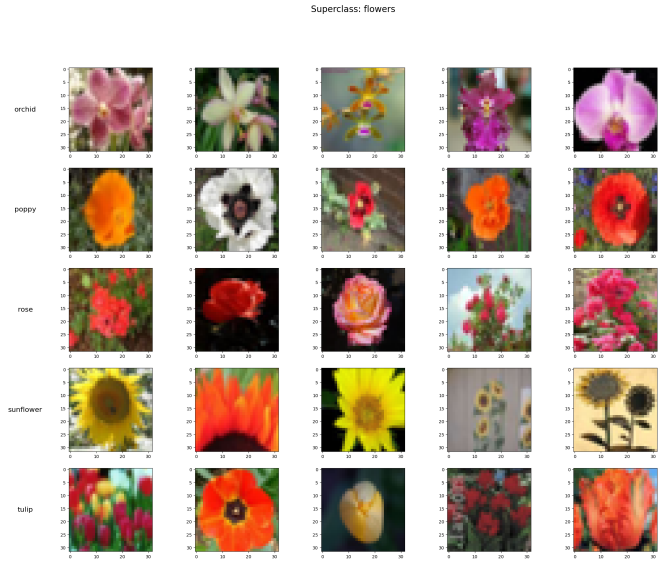


Figure 24: Superclass: Flowers

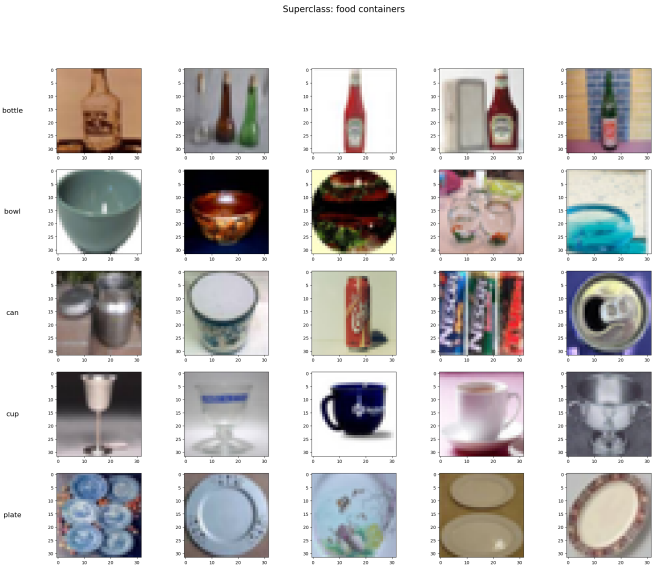


Figure 25: Superclass: Food containers

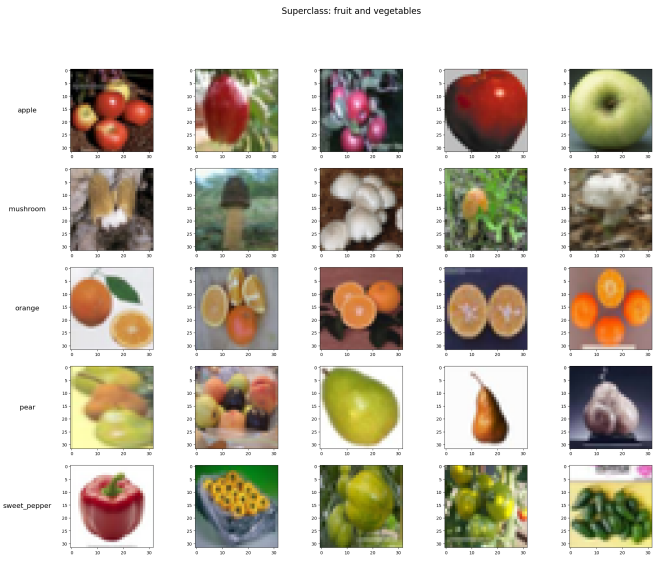


Figure 26: Superclass: Fruit and vegetables

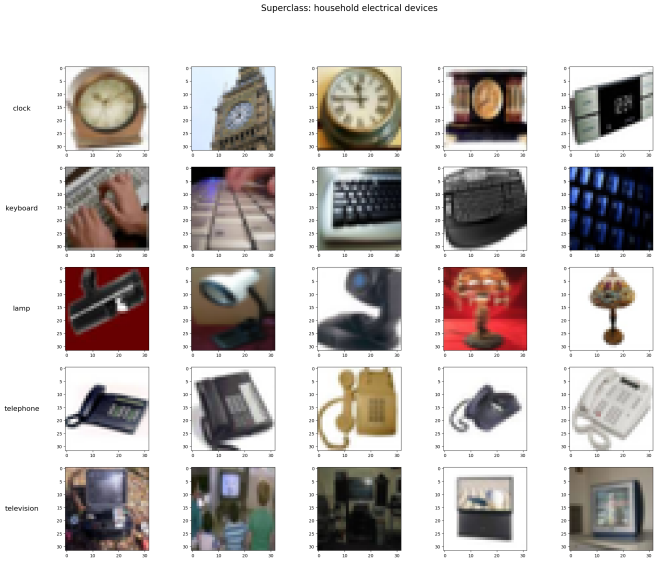


Figure 27: Superclass: Household electrical devices

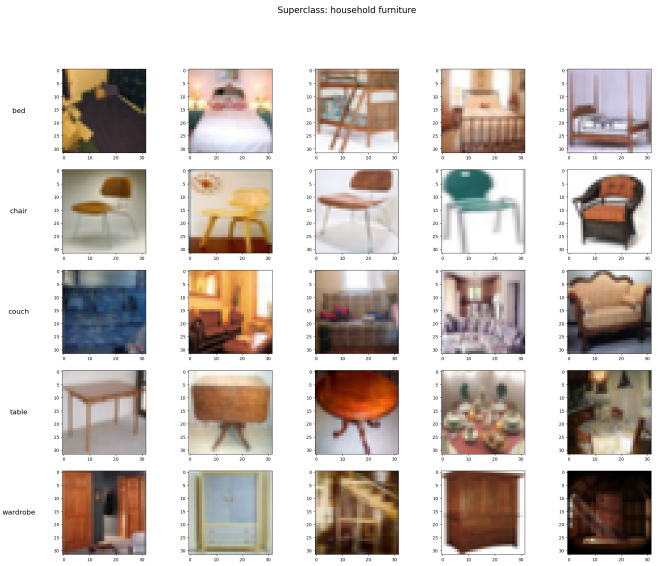


Figure 28: Superclass: Household furniture

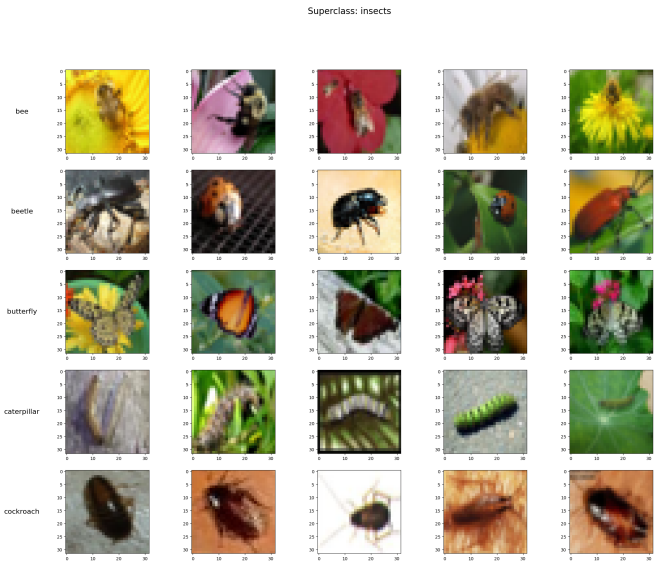


Figure 29: Superclass: Insects

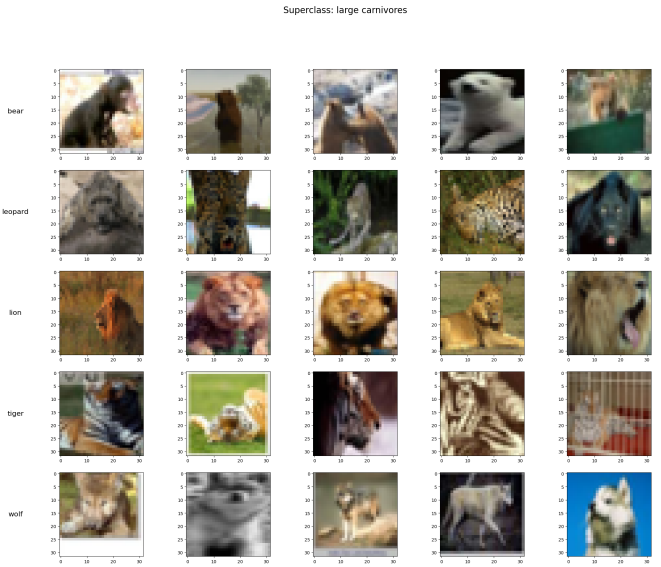


Figure 30: Superclass: Large carnivores

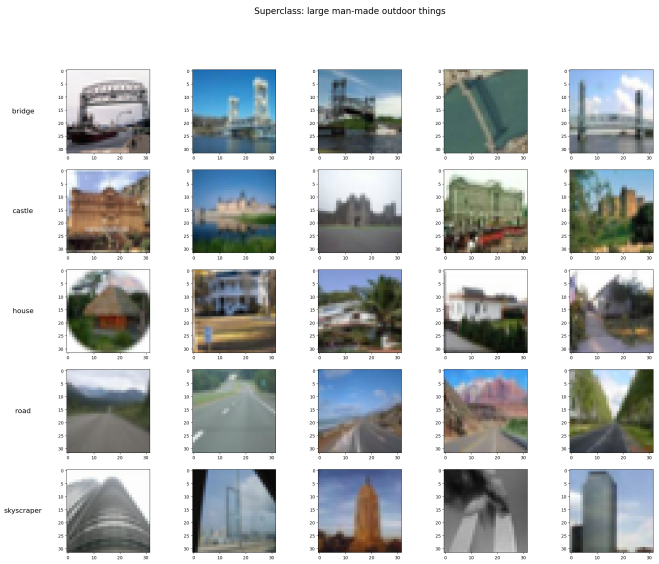


Figure 31: Superclass: Large man-made outdoor things

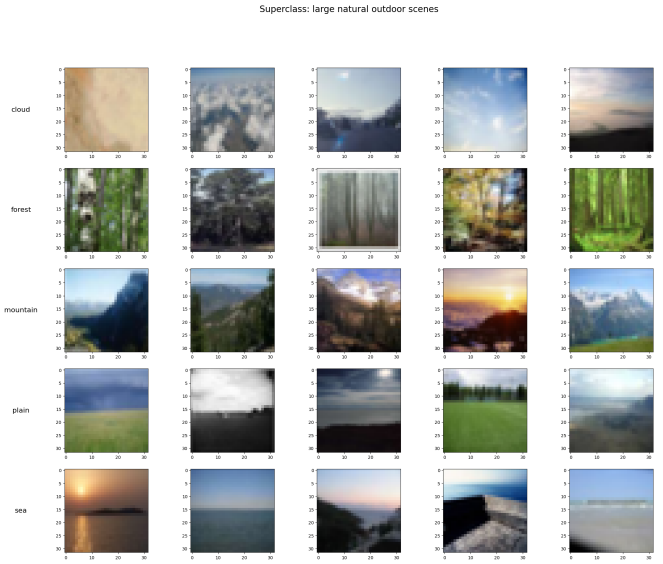


Figure 32: Large natural outdoor scenes

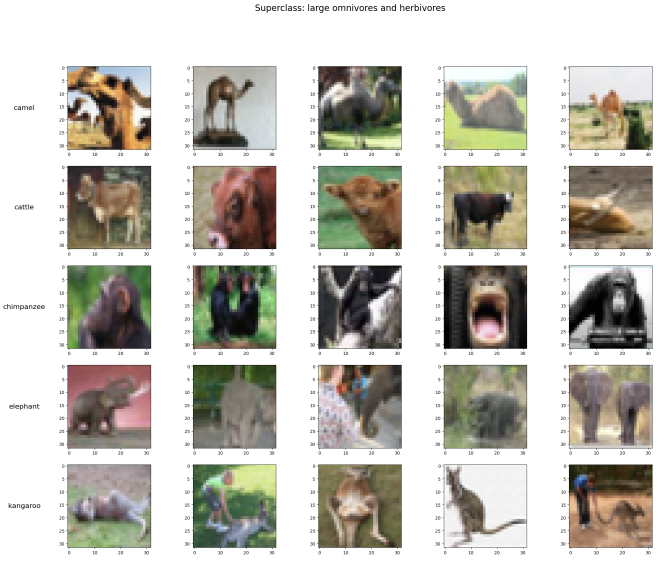


Figure 33: Superclass: Large omnivores and herbivores

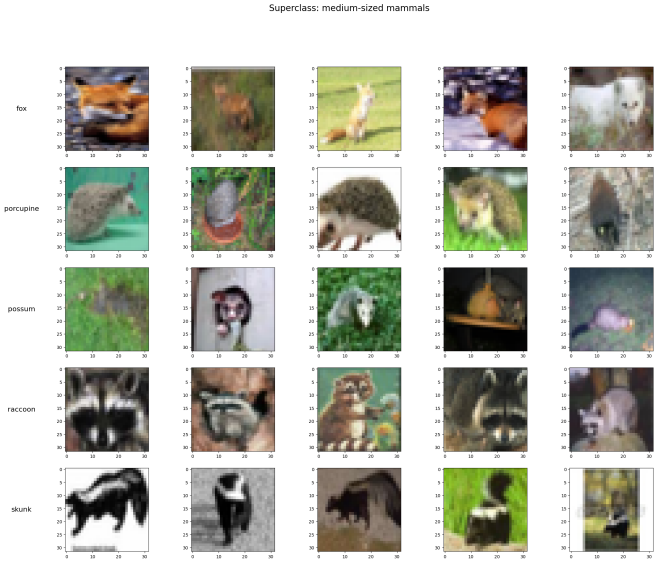


Figure 34: Superclass: Medium-sized mammals

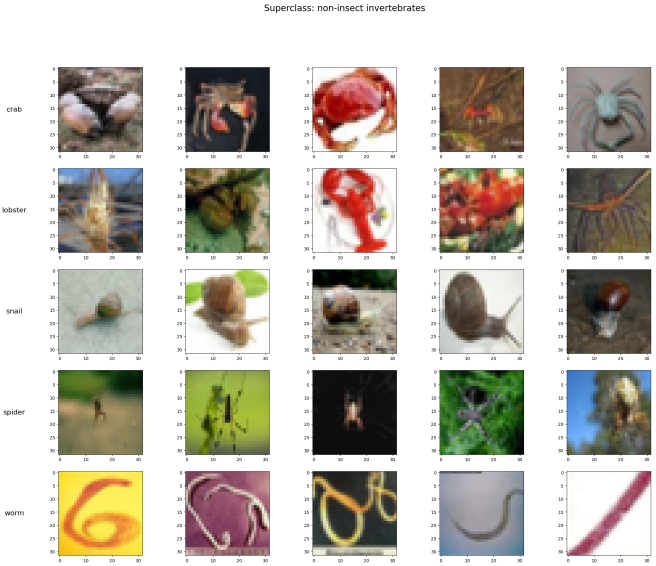


Figure 35: Superclass: Non-insect invertebrates

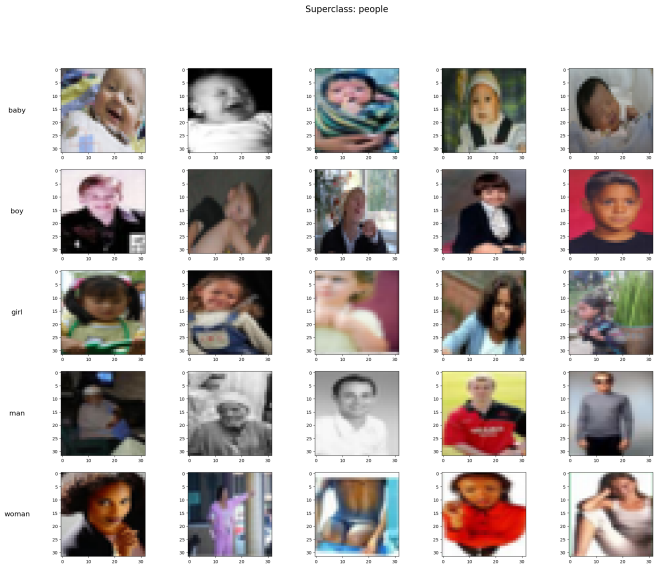


Figure 36: Superclass: People

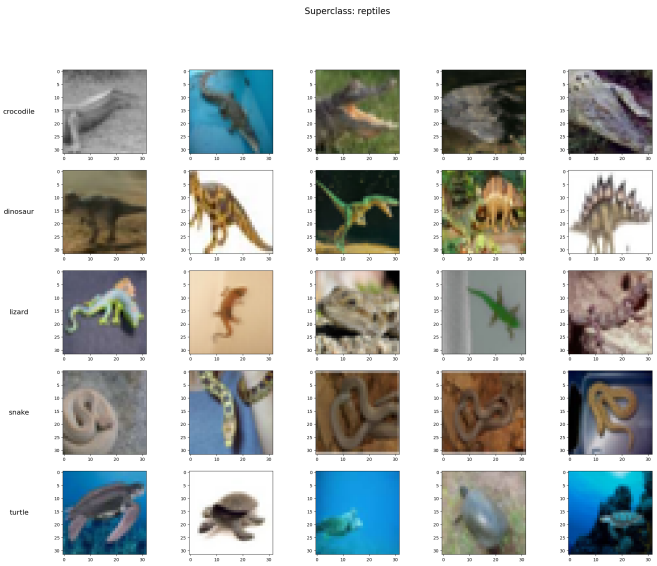


Figure 37: Superclass: Reptiles

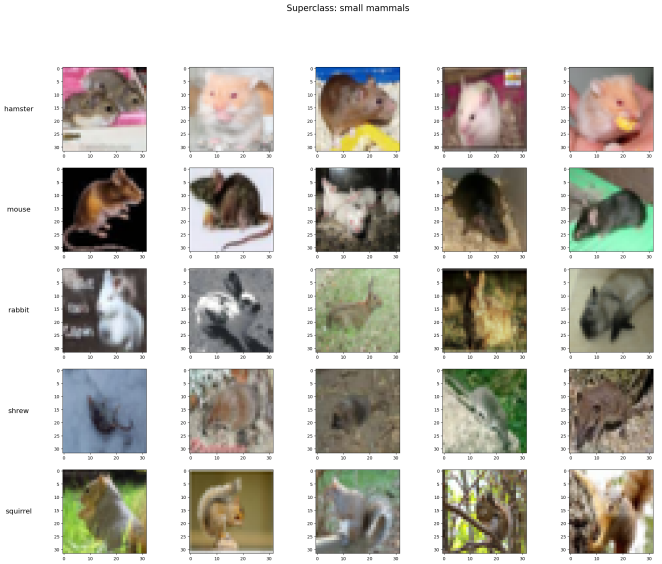


Figure 38: Superclass: Small mammals

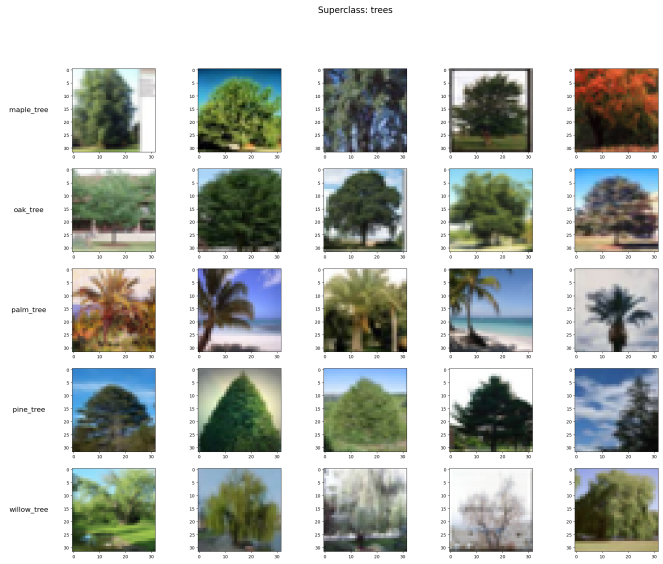


Figure 39: Superclass: Trees

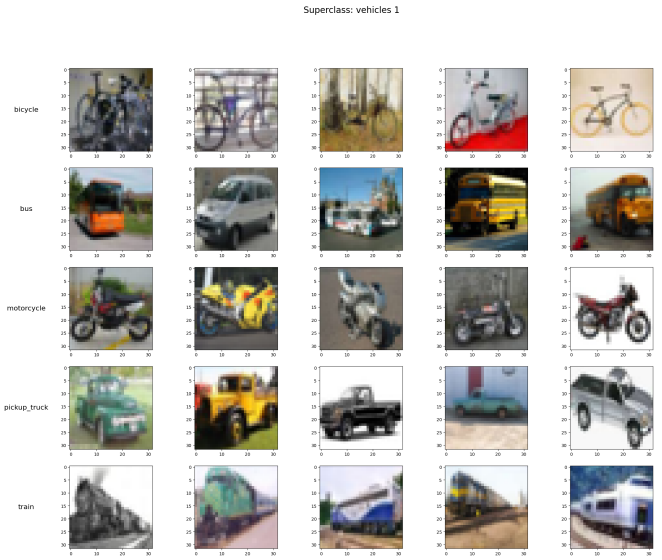


Figure 40: Superclass: Vehicles 1

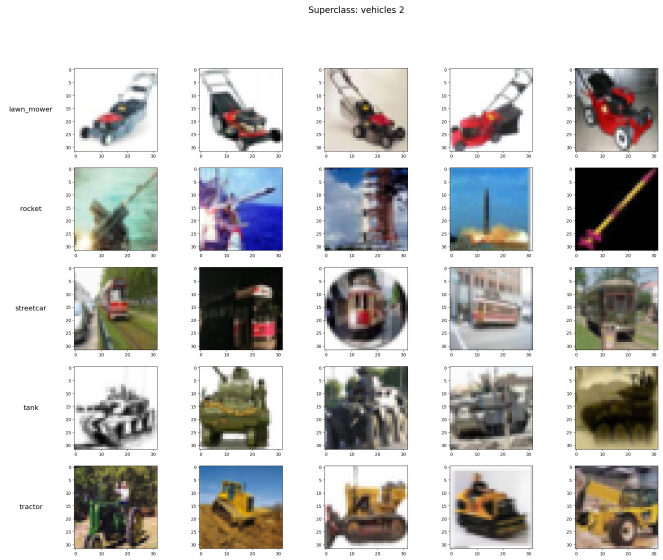


Figure 41: Superclass: Vehicles 2

References

D. V. Godoy. Dl-visuals [github repository]. URL `https://github.com/dvgodoy/dl-visuals`.

Diederik P Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, pp. 2278–2324, 1998. doi: 10.1109/5.726791.