Transfer Learning Via Default Project 1

Olof Lilliestierna **Nils Hamrin** **Mostafa Mohammadi**
Group 60 - KTH, Stockholm, Sweden
oloflil@kth.se nilham@kth.se mosmoh@kth.se

Abstract

This report investigates the application of transfer learning for image classification using the pre-trained convolutional neural networks ResNet18 and ResNet50 on the Oxford-IIIT Pet Dataset. The work is divided into two parts: binary classification with cats and dogs, and multi-class classification across 37 pet breeds. For the binary task, only the final layer is trained. In the multi-class setting, we compare two fine-tuning strategies, simultaneous tuning of the last l layers and gradual unfreezing, evaluating their performance in terms of test accuracy and compute time. We further study the effects of data augmentation, L2 regularization, batch normalization updates and learning rate schedules. Additional experiments examine the impact of class imbalance and the use of deeper networks. Our findings highlight the importance of strategy choice and regularization techniques in achieving robust performance in transfer learning, achieving a maximum accuracy of 98.39% for binary classification and 91.99% for multi-class classification.

1 Introduction

The goal of this project is to explore the application of transfer learning by adapting and fine-tuning pre-trained convolutional neural networks (ConvNets), specifically ResNet18, to solve image classification problems using the Oxford-IIIT Pet Dataset. This dataset consists of 7,349 images of 37 distinct breeds of cats and dogs, and provides a useful benchmark for both binary and multi-class classification tasks.

The project is structured in two main stages. In the first stage, we perform binary classification to distinguish between images of cats and dogs. This is achieved by replacing the final fully connected layer of a pre-trained ConvNet with a new layer that outputs two class scores. Only this final layer is fine-tuned using the training data, while the rest of the network remains frozen. In the second stage, we turn to a more challenging task: recognizing the specific breed of the animal in the image. This is a multi-class classification problem with 37 classes. Here, we replace the final layer to output 37 class scores and adopt deeper fine-tuning strategies.

2 Related Work

Fine-tuning pre-trained convolutional networks has become a widely used approach in image classification, particularly when working with smaller or specialized datasets. Recent studies have explored how different fine-tuning strategies impact performance across varying levels of task complexity.

Kornblith et al. examined how well models pre-trained on ImageNet transfer to a range of downstream image classification tasks [2]. They found that simply replacing the final layer often yields suboptimal results on more challenging datasets. Instead, performance improved significantly when fine-tuning deeper parts of the network, especially when accompanied by appropriate regularization and learning rate adjustments. Their results support the use of progressive fine-tuning in cases where the target task differs substantially from the pre-training objective.

Khosla et al., in the context of supervised contrastive learning, also applied staged fine-tuning of ResNet architectures [1]. Although their main contribution was in designing a contrastive loss function, they showed that performance on classification tasks improved when multiple layers were fine-tuned after pre-training. Their work further reinforces the practical value of adjusting the fine-tuning depth based on the complexity of the new task and the size of the available dataset.

These studies provide empirical support for the strategies used in this project, including gradually increasing the number of trainable layers and applying data augmentation and regularization to improve generalization.

3 Data

3.1 The IIIT pet dataset

For this study we use the IIIT pet dataset. This dataset consists of images of 37 different pet breeds, either cats or dogs [3]. There are approximately 200 images of each class, and the images are of different sizes and proportions. The standard transformation was done by using the default weights for the respective networks, imported from Torchvision.

3.2 Dataset splits

When downloading the dataset, we used a PyTorch function that had already split the test from the train/validation data [4]. After that we used the random_split function from torch in order to split our validation and training data. We used 80% of the data for training and 20% for validation.

When performing experiments with imbalanced data, another split was used. Here we split the data such that 20% of the images of each class still are put in the validation set, but instead of putting all the remaining images in the training set, only 25% of the remaining cat images are put into the training set, and the rest remain unused. This entails that the final training dataset has 80% of the dog images in the train/validation data but only 20% of the cat images. A cat majority set was also constructed where 80% of non-test cat images and 20 % of the non-test dog images was used. The following table describes the distribution of data points in the test, train and validation dataset for each of the splits:

Dataset	Train size	Val size	Test size
Normal	2944	736	3669
Dog majority	2227	712	3669
Cat majority	1450	726	3669

4 Method

For the following experiments, the evaluation is done once for every epoch using the validation data and then one final time after the last epoch using the testing data to calculate the model's accuracy. All training runs were made using a total of 10 epochs. The models were trained locally using CUDA on a NVIDIA GTX 1080 GPU. This will be the standard for all following experiments. Note, due to the extensive compute times, there was no opportunity to conduct any proper hyper-parameter tuning using, for instance, a grid or randomised search. The used hyper-parameters were found to be adequately good for the experiments using trivial trial and error.

Strategy 1: Fine-tune l layers simultaneously

In this strategy, we investigate the impact of fine-tuning different numbers of layers in the network. Specifically, we fine-tune the last l layers of the ConvNet, starting with $l = 1$ and incrementally increasing to deeper layers, alongside the newly initialized classification layer.

Strategy 2: Gradual unfreezing

Gradual unfreezing is a staged fine-tuning process where training begins with the final layers of the network and earlier layers are progressively unfrozen as training progresses. This strategy is intended to provide more stable convergence by allowing higher-level task-specific features to be learned first, before adapting lower-level features.

Fine-tuning with imbalanced classes

We simulate class imbalance by retaining only 20% of the training data for each cat breed, while keeping all dog breed data. We evaluate the performance impact and explore compensatory techniques such as weighted cross-entropy loss and over-sampling of minority classes.

Extensions

Beyond the three main strategies, we explore additional scenarios to broaden our understanding of fine-tuning dynamics. These extensions provide further insight into how architectural choices, training strategies, and dataset characteristics interact in the context of transfer learning with pre-trained ConvNets.

1. Using deeper networks. We substitute ResNet18 with the deeper ResNet50 model to assess whether increased model capacity leads to improved performance. We experiment with different optimizers different layer depths ($l = 0$, $l = 1$, $l = 3$).

2. Batch normalization behaviour. We analyse how freezing or updating batch normalization parameters affects learning. Since batch norm layers maintain running estimates of mean and variance, is is possible that controlling these could influence model convergence and generalisation.

5 Experiments

5.1 Binary classification

First off, the binary classification of the data set. For this initial experiment, the BCELoss criterion was used in combination with Adam due to its binary classification strengths. The learning rate was set to 0.001, which will be used as the default learning rate for this project.

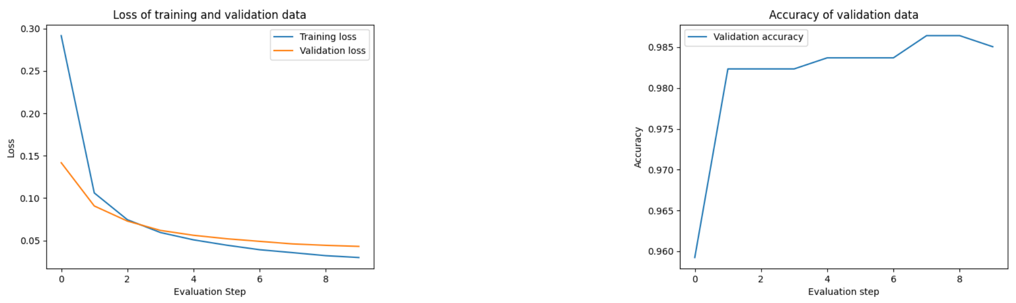

Figure 1: Binary classification loss and accuracy

While the experiment did not result in the suggested $\geq$99% accuracy, the testing accuracy was rather close at a final 98.39%. Since the experiment was not done using any type of hyper-parameter tuning, the result was deemed to be satisfactory close to the suggested accuracy. As can be seen in Figure 1, the sharp loss decline and flattening validation accuracy curve also suggests that this might, at least, be close to the optimum accuracy for these specific hyper-parameters.

5.2 Multi-class classification

5.2.1 Strategy 1

The compute, and time, ceiling was deemed to be drawn at $L = 5$ with the set epoch amount. At this point, as is showcased in Figure 2, there was also not really any obvious benefit to adding more layers. For this experiment, SGD was used as the optimizer with the learning rate set to 0.001, momentum to 0.9 and *nestrov* to True. This was done in hope of faster convergence to reduce compute time.

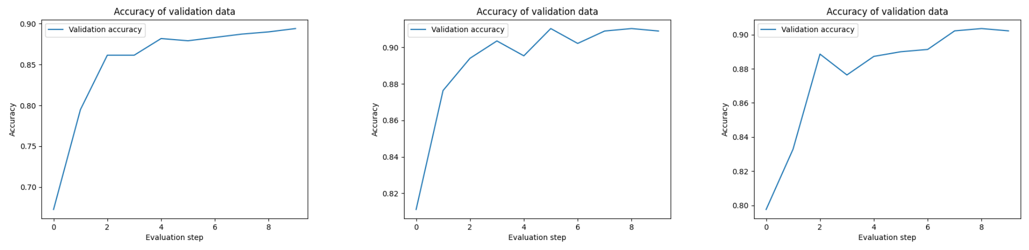

Figure 2: Strategy 1 accuracy using one, three and five layers (left to right)

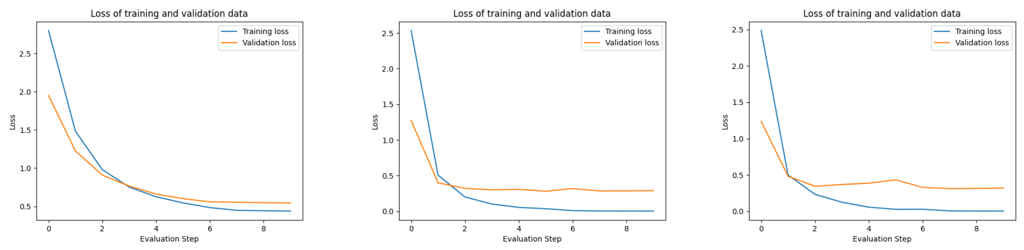

Figure 3: Strategy 1 loss using one, three and five layers (left to right)

All five versions, using $l = 1, 2, 3, 4$ and 5, had quite similar outcomes. Although, as can be seen in the figures 2 and 3, it did require some more epochs to get a satisfactory validation accuracy on the models training more layers. Interestingly, these models had also a larger divide between the training and validation loss, which could possibly be the result of overfitting when greater parts of the network tries to learn and adapt to the training data. Also to note, the reason why the training loss starts out as greater compared to the validation loss is most likely due to dropout. The code uses the evaluate mode when calculating loss and accuracy for validation and testing, but not training.

Finally, some other configurations using: A higher learning rate at 0.01, L2 regularization with weight decay set to 0.01 and training data augmentation were tested. For the data augmentation a random horizontal flip was used in combination with random rotation of 20 degrees and a colour jitter for the data transform. Except for this differentiating factors, all hyper-parameters remained the same from the previous experiments. For the sake of compute time, all of these were only computed using $l = 3$.

Active layers l	**Test accuracy**
1	86.56%
2	88.33%
3	88.91%
4	89.32%
5	89.07%

Factor ($l = 3$)	**Test accuracy**
Baseline	88.91%
Higher learning rate	88.25%
L2 regularization	88.58%
Data augmentation	89.72%

Compared to the original test accuracy of 88.91%, these changes are rather minor, but alas not insignificant. The lower accuracy, along with a similar loss profile, using L2 regularization could suggest that there is not any overfitting issue. Also, using more advanced data augmentation in the transform, as compared to the standard weights, seems to have a improvement on final accuracy, at least using these specific hyper-parameters.

5.2.2 Strategy 2

Using gradual unfreezing were in comparison to strategy 1 in some ways better, and in some ways worse. It used about as much compute time, 208 seconds, as when strategy 1 trained on three layers. Training on fewer layers than this was quicker and more layers slower. The new strategy had also quite similar performance, 88.77% as compared to 88.91%. As can be read in the tables, strategy 1 with fewer layers had worse accuracy compared to this, while three or more layers performed better. During this experiment, other factors were also tested with the same method as during strategy one.

Factor	**Test accuracy**	**Compute time**
Baseline	88.77%	208 seconds
Higher learning rate	88.85%	207 seconds
L2 regularization	88.49%	208 seconds
Data augmentation	85.34%	232 seconds
Different learning rate on different layers	88.69%	208 seconds

Interestingly, here the higher learning rate was the best performer, while being the worst during strategy one. Seemingly, the ramp up of layer unfreezing requires the network to learn faster. The opposite can be said when testing the data augmentation factor. The new factor, *Different learning rate on different layers*, used $lr = 0.01$ on layer one, dividing this by ten for each layer. While the difference in learning rate was quite large, it seemed to have a negligible effect on the network's performance and compute time.

5.2.3 Fine-tuning with imbalanced classes

For this experiment, we used the imbalanced dataset as described in Chapter 3.2 with only a select amount of the images. In one of the tests, the dataset using only 20% of the dog images was used.

A total of seven tests were conducted. The network was trained on the imbalanced data with one to five layers unfrozen, and finally twice more with $l = 3$. Once using a weighted cross-entropy loss function that makes each class contribute the same total weight, and once with the cat majority dataset.

Active layers l	**Test accuracy**
1	75.85%
2	83.37%
3	85.93%
4	84.98%
5	85.17%

Factor ($l = 3$)	**Test accuracy**
Baseline	85.93%
Weighted loss	85.42%
Cat majority dataset	83.13%

Here we can observe that the networks result with an imbalanced training dataset perform notably worse than the previous two strategies. This is not surprising since the uneven data split likely is a worse representation of the test dataset. When $l = 1$ this difference is especially significant. We also observe that no significant improvement was obtained when using our weighted loss, and when using the cat majority dataset we get even worse results, which could be partially because that training dataset is smaller and partially because there are fewer cats in the actual dataset, and thus probably also the test dataset.

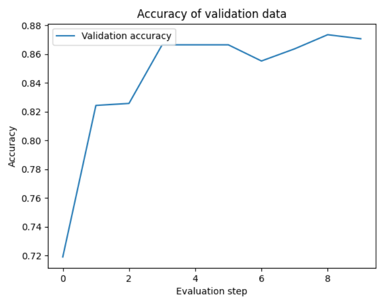

Figure 4: Accuracy of the network with $l = 3$ and weighted loss trained on the imbalanced data

5.3 Extensions

5.3.1 Deeper networks

This first extension was completed using ResNet50 instead of the previous ResNet18. As one might suspect, using a deeper network did indeed correlate better accuracies, although with some important issues. This particular experiment was done using the SGD and AdamW optimizers to compare the two, and as a added bonus also SGD with L2 regularization. The weight decay was set to 0.01 for the SGD optimizer and 0.001 for the one with AdamW. All optimizers were trained on zero, that being only the added classification layer, one and three layers.

Optimizer ($l = 0$)	**Test accuracy**
SGD	90.49%
SGD with L2	90.71%
AdamW	90.95%

Optimizer ($l = 3$)	**Test accuracy**
SGD	91.99%
SGD with L2	91.74%
AdamW	48.71%

Focusing on the pair of SGD optimizers, here it is clear that training more layers than just the classifying layer does indeed improve performance, although not quite as much as when using a much smaller network. The obvious outlier here being the AdamW optimizer training on three layers. The issue here was that, as can be seen in Figure 5, the accuracy crashes after the first epoch and never recovers. During the experiment no obvious combination of hyper-parameters was found to fix this issue. A suggested cause for this issue could be a vanishing or exploding gradient causing a inaccurate update step. Apart from this outlier, the difference between the two optimizers were not all that great.

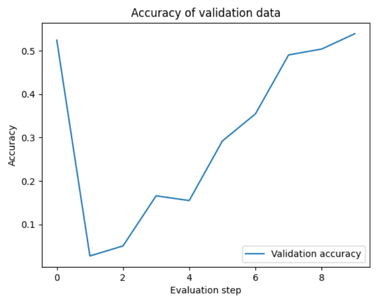

Figure 5: Accuracy of the deep network using the AdamW optimizer on three layers

Optimizer	**Average test accuracy**	**Average compute time**
SGD with L2	90.64%	243.77 seconds
AdamW	90.94%	246.04 seconds

As showcased, there is not much between the two when comparing the representative results. Although, do note that the compute time is not very scientific since other programs were running at the computer at the same time, which could have affected the relative difference. Also, as stated in the introduction to the method in Chapter 4, better results might be possible when searching for the optimal hyper-parameters.

5.3.2 Batch normalisation

The second extension consisted of testing the impact of only fine-tuning the batch normalization layers in the network. This would be beneficial as we would save training time by not training any of the convolutional layers. In order to test this, an experiment was conducted where the network was trained once with all layers except the final layer frozen, and once where both the final layer and all the batch normalization layers were unfrozen.

Unfrozen layers	**Test accuracy**
Final layer	88.31%
Final + batch norm	88.72%

We can observe a slight improvement when the batch normalization layers are allowed to train alone compared to when only the final layer is trained. Furthermore, the final test accuracy is quite similar to most of the results where $l = 3$ obtained in some of our previous tests. When comparing it to the network where three layers are trained with the same hyperparameters the performance is only 0.2% worse. This implies that you could do a bit more tests with only training batch normalization and see if we can obtain better scores when implementing the same methods as was done with three unfrozen layers.

6 Conclusion

To conclude, we can observe that we were able to apply transfer learning to the IIIT pet dataset in order to obtain quite good results. We were able to achieve a 98.39% test accuracy on the binary classification task of differentiating dog pictures and cat pictures without too many tests, and a test accuracy of approximately 90 to 92% for the multi-class classification task of correctly identifying pet species. Using a deeper network, unfreezing more layers and training using the SGD optimizer all seemed to lead to an improved performance, but often at the cost of increased compute time. Given more time and computing resources, it would be interesting to train some of the good hyper-paramter settings for longer or try combining multiple strategies.

References

[1] Prannay Khosla et al. *Supervised Contrastive Learning*. 2021. arXiv: `2004.11362 [cs.LG]`. URL: `https://arxiv.org/abs/2004.11362`.

[2] Simon Kornblith, Jonathon Shlens, and Quoc V. Le. *Do Better ImageNet Models Transfer Better?* 2019. arXiv: `1805.08974`. URL: `https://arxiv.org/abs/1805.08974`.

[3] Omkar M Parkhi et al. "Cats and dogs". In: *2012 IEEE Conference on Computer Vision and Pattern Recognition*. 2012, pp. 3498–3505. DOI: `10.1109/CVPR.2012.6248092`.

[4] PyTorch. 2025. URL: `https://pytorch.org/`.

Appendix

The code for this project can be found at: https://github.com/nhamrin/DeepLearning-Project