Project 1

Image Classification with CNN

Freddy Roldan Rivero

Karthikeyan Karuppusamy (KK)

Alexandre Donciu-Julin

25-27th September, 2024

Table of Contents

Introduction

This project focuses on image classification using Convolutional Neural Networks (CNNs) to classify images from the CIFAR-10 dataset.

Initially, we experimented with custom-built CNN architectures and data augmentation techniques to improve model performance.

Subsequently, we explored transfer learning and fine-tuning using pre-trained models like VGG16 and ResNet50 to leverage their pre-existing feature extraction capabilities.

By comparing different approaches, we aimed to identify the most effective model for classifying low-resolution images in CIFAR-10 efficiently.

1. Datasets

1.1. Dataset Selection

We were presented two datasets: Animals-10 and Cifar10. We spent a couple of hours reviewing Animals10 but decided to go for Cifar10, for the following reasons:

- Animals10 is much heavier. It takes quite a long time to load it and unlike Cifar10, a pre-processed version is not available on tensorflow.
- Animals10 images have different sizes, they are not squared and they have a higher resolution than Cifar10 images. It will make preprocessing and training times much longer.
- Cifar10 images are loaded as numpy arrays directly, making it easy to manipulate and plot them for a quick reviewing.

1.2. Dataset Review

The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. The dataset is balanced, mutually exclusive (only one class subject per picture) and the images are of relatively good quality. The only limitation is the size of 32x32 pixels, which is quite small and could be an issue if running through too many Convolution and MaxPooling layers. *See Figure 1.2.1 below.*

1.3. Dataset Preprocessing

Two pre-processing steps were required to prepare the Cifar10 images for our models:

- **Batch normalization:** Images had color values ranging from 0 to 255, we normalized them from 0 to 1. When working with ResNet50, we used the preprocess_input method from keras that normalizes the data for this model specifically. *For instance: 255, 255, 255 (white) => (1, 1, 1)*.
- **Labels 1-Hot-Encoding**: Labes loaded from Cifar10 are class integers ranging from 0 to 9. To avoid ambiguous prediction values like 8.5, we decided to transform label values to categorical one-hot-encoded ones. *For instance: 5 => [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]*

Additionally we implemented **data augmentation** using a keras ImageDataGenerator, introducing rotation, horizontal flip, width and height shifts to our images. This helped make the model more robust to unseen data, by training it with some variations of the same picture.

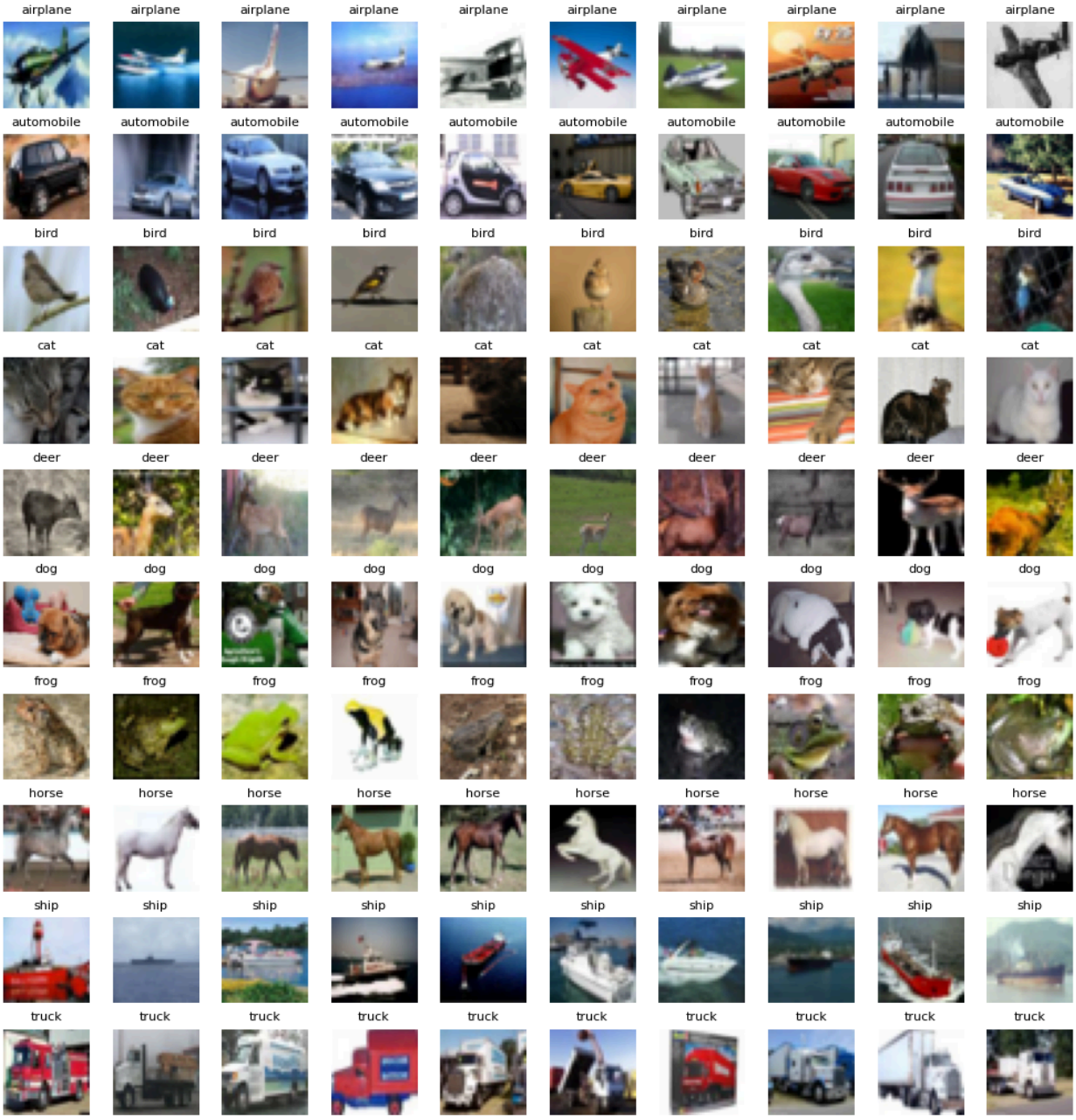


Figure 1.2.1 - Cifar10 classes

2. Models Architectures

The requirements for this project was to build a CNN classifier and then perform transfer learning from an existing one. We therefore discussed different approaches and model architectures and agreed on the following roadmap:

- **Build a home-made classifier:** We had already built a simple CNN from scratch during a lab. We should therefore resume working on it, complexify it, try tuning different hyperparameters and bring it to a point where we think that we are hitting a limit, when the model does not really improves anymore. We can use the VGG16 architecture and a reference.
- **Transfer Learning / Fine Tuning VGG16:** This is the first model we looked into. Already pre-trained on ImageNet dataset which contains millions of labeled images across 1000 classes, making this model mode efficient on extracting useful features from different image datasets including those it wasn't directly trained on such as CIFAR-10. It has a deep architecture with 16 layers.
- **Transfer Learning / Fine Tuning ResNet50:** Finally we decided to run some tests on ResNet50, which is a more complex network, that can cope very good with the vanishing gradient problem by introducing residual connections so the model converge faster. ResNet50 is a versatile model, which can adapt to different image sizes, especially low-resolution datasets like Cifar10. On top of that, its internal architecture of convolution blocks makes it easy to perform transfer learning and fine-tuning on it.

Figure 2.1. Model Training Results is a table summarizing our main results and the models we have trained.

Model Architecture	Dev	Total Layers	Dense	Params	Epochs	DataAug.	BatchNorm.	Dropout	Optimizer	Loss	Test Acc	Test Loss	f1-score	recall	Comments
Home-made CNN (v3)	Alexandre	18	2	2,196,810.00	15		X		Adam	cat_cross	0.81	0.77	0.81	0.81	Overfitting
Home-made CNN (v4)	Alexandre	19	2	2,200,394.00	20	X	X	X	Adam	cat_cross	0.91	0.67	0.91	0.91	No more overfitting
Home-made CNN	Freddy	12	2	4,101,450.00	50	X	X	X	Adam w/Scheduler	cat_cross	0.8195	1.1084	0.81	0.81	Some overfit
Home-made CNN	KK	18	2	3,514,698.00	50	X	X	X	Adam	cat_cross	0.9	0.35	0.9	0.9	Final Model
VGG16 TransferLearn	KK	25	3	41,071,690.00	50		X	X	Adam	cat_cross	0.78	0.69	0.78	0.78	Close to Train accuracy of 0.81 & loss of 0.535
VGG16 Finetuning	KK	25	3	41,071,690.00	30		X	X	Adam	cat_cross	0.84	0.54	0.84	0.84	Training acc 0.94 & loss 0.16, bit of Overfit.
VGG16 TransferLearn- DA	KK	25	3	41,071,690.00	100	X	X	X	Adam	cat_cross	0.68	0.89	0.68	0.68	Bad Training accuracy of 0.64 & loss of 1.00
VGG16 Finetuning - DA	KK	25	3	41,071,690.00	50	X	X	X	Adam	cat_cross	0.78	0.65	0.78	0.78	Close to Training acc 0.85 & loss 0.43. Can improve!
VGG16 TransLearn	Freddy	12	3	4,101,450.00	50	X	X	X	Adam/wscheduler	cat_cross	0.87	0.99	0.87	0.87	Improvement of validation accuracy. Persintent loss spiking
ResNet50 TransLearn	Alexandre	188	4	49,723,082.00	20	X	X	X	RMSProp	cat_cross	0.95	0.17	0.95	0.95	Best one so far
ResNet50 TransLearn	Alexandre	188	4	49,723,082.00	24	X	X	X	RMSProp	cat_cross	0.96	0.15	0.96	0.96	Fine-tuning did not work. Wrong predictions on new images.

Figure 2.1. Model Training Results

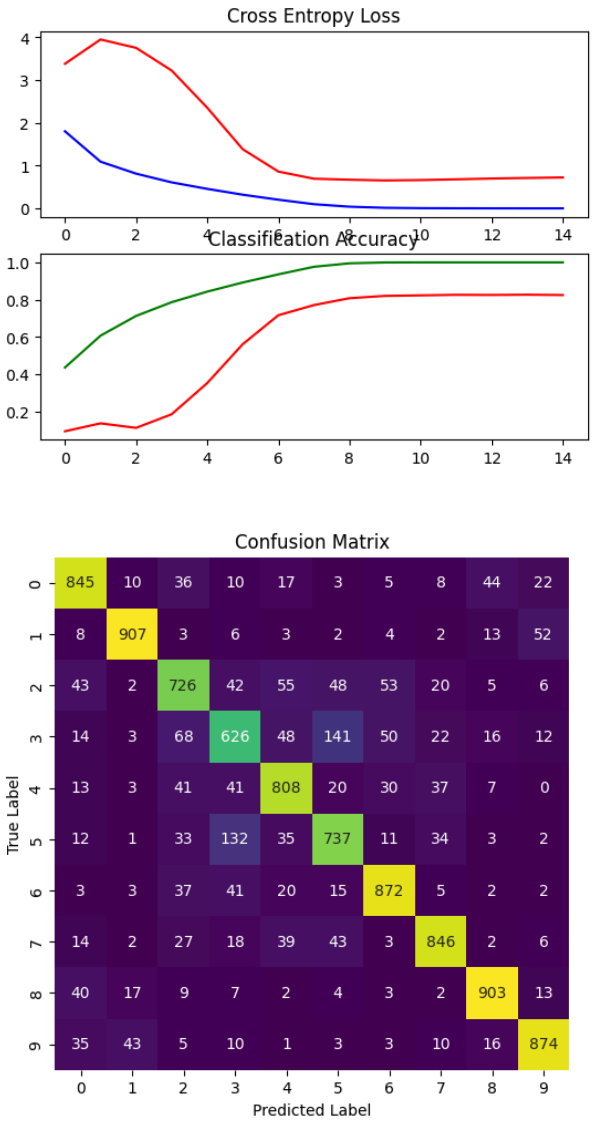
3. Home-Made CNNs

We are presenting here the architecture and the results we get from training CNNs built from scratch. We used the same approach that can be found in more complex models like VGG16: Implementing blocks of Convolution2D/MaxPooling layers with increasing amounts of filters. In the process, we are tweaking the hyperparmeters and trying to reduce overfitting.

3.1. Without data augmentation

This is the best model we could get without data augmentation. It is using 3 blocks of Convolution2D/MaxPooling and 2 dense layers. See results on *Figure 3.1.1*.

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_6 (Conv2D)           (None, 32, 32, 64)        1792

 conv2d_7 (Conv2D)           (None, 32, 32, 64)        36928

 max_pooling2d_3 (MaxPoolin  (None, 16, 16, 64)        0
 g2D)

 conv2d_8 (Conv2D)           (None, 16, 16, 128)       73856

 conv2d_9 (Conv2D)           (None, 16, 16, 128)       147584

 max_pooling2d_4 (MaxPoolin  (None, 8, 8, 128)         0
 g2D)

 conv2d_10 (Conv2D)          (None, 8, 8, 256)         295168

 conv2d_11 (Conv2D)          (None, 8, 8, 256)         590080

 max_pooling2d_5 (MaxPoolin  (None, 4, 4, 256)         0
 g2D)

 flatten_1 (Flatten)         (None, 4096)              0

 dense_2 (Dense)             (None, 256)               1048832

 dense_3 (Dense)             (None, 10)                2570

=================================================================
Total params: 2196810 (8.38 MB)
Trainable params: 2196810 (8.38 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```



- Optimizer/Loss: Adam / Cat. Crossentropy
- Epochs: 15
- Test accuracy: 0.81
- Test loss: 0.77
- F1-score and recall: 0.81

Figure 3.1.1. Home-made CNN without data augmentation

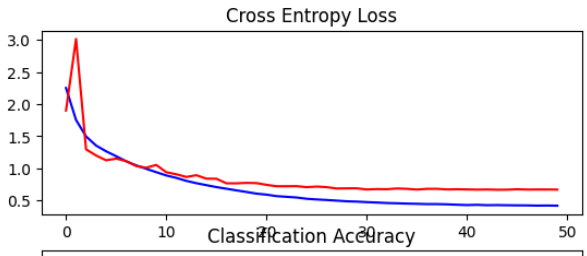
Conclusion: We can already reach a reliable 80% accuracy using just 3 blocks and 2 dense layers. As expected, the model is overfitting and there is a clear gap between training curves (blue and green) and validation curves (red). We can now move on and implement methods to reduce overfitting.

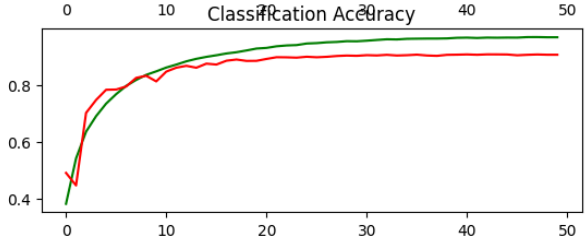
3.2. With data augmentation

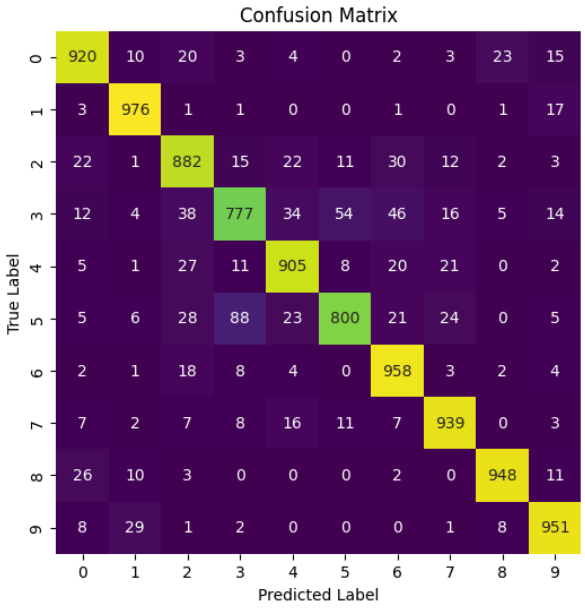
This second model uses the same architecture as the previous one. *See Figure 3.2.1.*

Layer (type)	Output Shape	Param #
conv2d_18 (Conv2D)	(None, 32, 32, 64)	1792
batch_normalization_18 (BatchNormalization)	(None, 32, 32, 64)	256
conv2d_19 (Conv2D)	(None, 32, 32, 64)	36928
batch_normalization_19 (BatchNormalization)	(None, 32, 32, 64)	256
max_pooling2d_9 (MaxPooling2D)	(None, 16, 16, 64)	0
conv2d_20 (Conv2D)	(None, 16, 16, 128)	73856
batch_normalization_20 (BatchNormalization)	(None, 16, 16, 128)	512
conv2d_21 (Conv2D)	(None, 16, 16, 128)	147584
batch_normalization_21 (BatchNormalization)	(None, 16, 16, 128)	512
max_pooling2d_10 (MaxPooling2D)	(None, 8, 8, 128)	0
conv2d_22 (Conv2D)	(None, 8, 8, 256)	295168
batch_normalization_22 (BatchNormalization)	(None, 8, 8, 256)	1024
conv2d_23 (Conv2D)	(None, 8, 8, 256)	590080
batch_normalization_23 (BatchNormalization)	(None, 8, 8, 256)	1024
max_pooling2d_11 (MaxPooling2D)	(None, 4, 4, 256)	0
flatten_3 (Flatten)	(None, 4096)	0
dense_6 (Dense)	(None, 256)	1048832
dropout_6 (Dropout)	(None, 256)	0
dense_7 (Dense)	(None, 10)	2570

Total params: 2200394 (8.39 MB)
Trainable params: 2198602 (8.39 MB)
Non-trainable params: 1792 (7.00 KB)







- Optimizer/Loss: Adam / Cat. Crossentropy
- Epochs: 50
- Test accuracy: 0.91
- Test loss: 0.67
- F1-score and recall: 0.91

Figure 3.2.1. Home-made CNN without data augmentation

To reach this score, we implemented different methods to reduce overfitting:

- BatchNormalisation layers between layers
- Dropout layers between the Dense layers
- Data augmentation (see section 1.3. on data preprocessing)
- EasyStopping to stop training when the accuracy score does not improving anymore

The confusion matrix clearly shows a good distribution along the diagonal, with some misclassifications, mainly between cats (3) and dogs (5), due to visual similarities between the two species, leading to confusion. The model architecture is not complex enough to capture the fine details that differentiates them.

Conclusion: Implementing the BatchNormalisation and Dropout layers really helped with reducing overfitting, especially training with augmented data to make the model more robust to unseen data. With 90% test accuracy, this is a very good base model that we can use to assess future transfer learning results.

4. Transfer Learning and Fine-tuning with VGG16

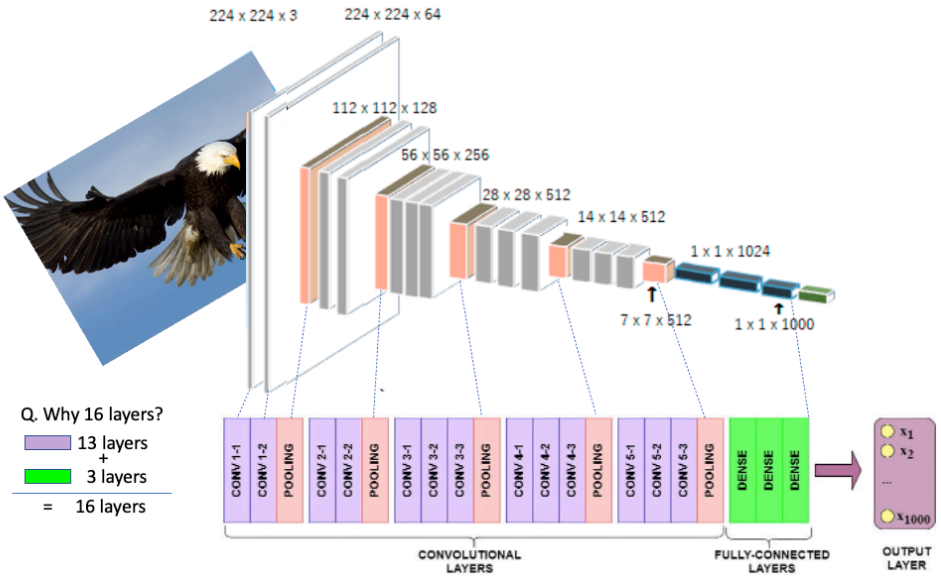
As part of Transfer Learning, we are doing 'Feature Extraction' of a Pre-Trained model and adding our Home-made CNN customizations to train the modified Pre-trained Model. The activities involve the following steps.

- Pick a Pre-trained model suitable for the input CiFAR dataset
- Replace the classification layer with our home-made CNN classification layer.
- Freeze all other pre-trained layers to prevent re-training.
- Modify hyperparameters and improve the model's performance.
- Evaluate the model and come up with observations.

We have evaluated different models like VGG16 , Inception & Resnet, we came to the conclusion that for a Dataset like CIFAR which is not very huge & complex to process, VGG16 can provide better performance than other models. VGG16 is also a pre-trained model using ImageNet (complex image DS) so it can handle CiFAR better. Some consideration to be given is that VGG16 handles images of size 224*224*3, whereas CIFAR images are 32*32*3.

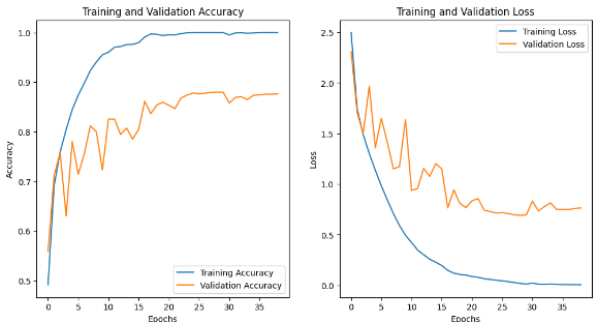
4.1. Transfer Learning

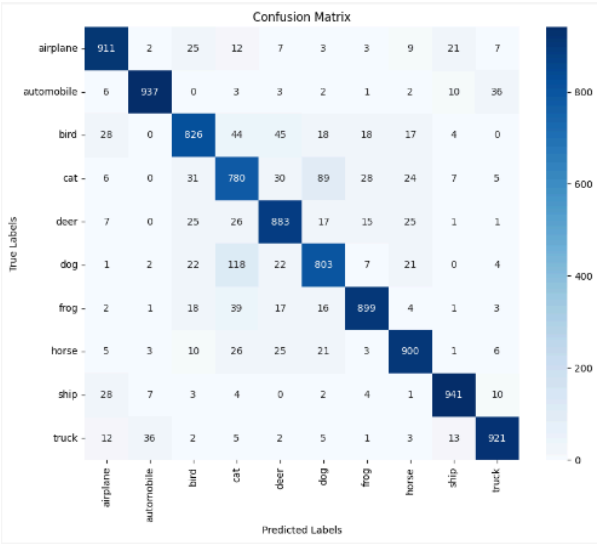
VGG16 Model has 16 layers of CONV and Dense layers and in addition pooling layers. As part of Transfer Learning we will be removing the Dense layer and include our own Dense and classification Output layers.



The following FC and Output layer is added to the VGG16 layer, in which the top layer was removed while copying.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 vgg16 (Functional)          (None, 7, 7, 512)         14714688

 conv2d (Conv2D)             (None, 7, 7, 256)         1179904

 batch_normalization (Batch  (None, 7, 7, 256)         1024
 Normalization)

 max_pooling2d (MaxPooling2  (None, 3, 3, 256)         0
 D)

 conv2d_1 (Conv2D)           (None, 3, 3, 512)         1180160

 batch_normalization_1 (Bat  (None, 3, 3, 512)         2048
 chNormalization)

 max_pooling2d_1 (MaxPoolin  (None, 1, 1, 512)         0
 g2D)

 global_average_pooling2d (  (None, 512)               0
 GlobalAveragePooling2D)

---
Total params: 17477450 (66.67 MB)
Trainable params: 9839114 (37.53 MB)
Non-trainable params: 7638336 (29.14 MB)
```





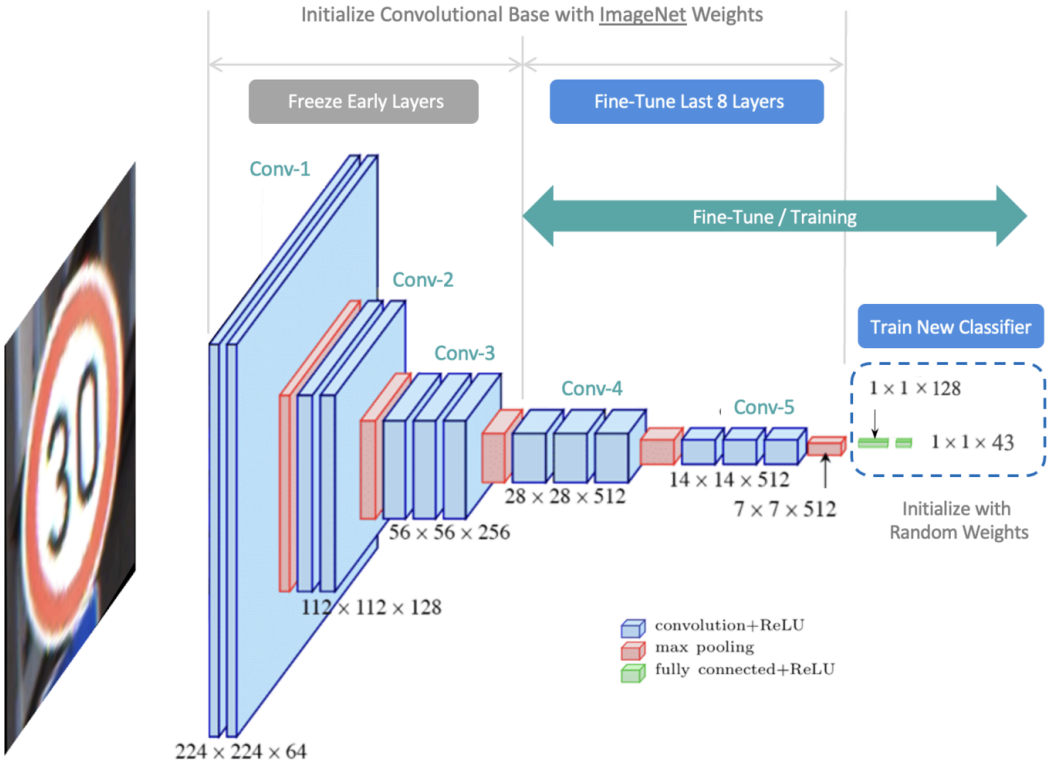
- Optimizer/Loss: Adam / Cat. Cross Entropy
- Epochs: 50
- Test accuracy: 0.87
- Test loss: 0.99
- F1-score and recall: 0.87

4.2. Fine-Tuning

Fine-Tuning is a method in which a pre-trained neural network model is retrained on a specific dataset to customize the model to meet particular requirements. Fine-tuning in VGG16 involves adapting a pre-trained model, which was originally trained on a large dataset like ImageNet, to a new more specific dataset, CIFAR-10 in our case.

In this model, we unfreeze some of the layers, 4 layers in our case, and retrain these layers with our dataset. This allows the model to adjust the weight of these unfreezed layers.

Example of a Fine-tuning architecture diagram is included below. Here we are fine tuning 8 layers, in our case we re-trained 4 layers. It is important to keep the learning rate to Low to allow for smaller adjustments of the pre-trained weights.



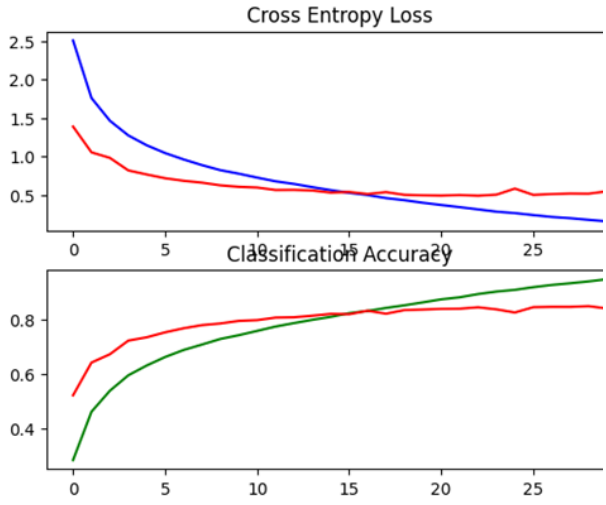
4.2. Model 2

The following pic shows the ‘Unfreezing’ of the last 4 Fully Connected layers of the VGG16 model and the model is compiled with a learning rate of 1e-5 to allow for smaller changes.

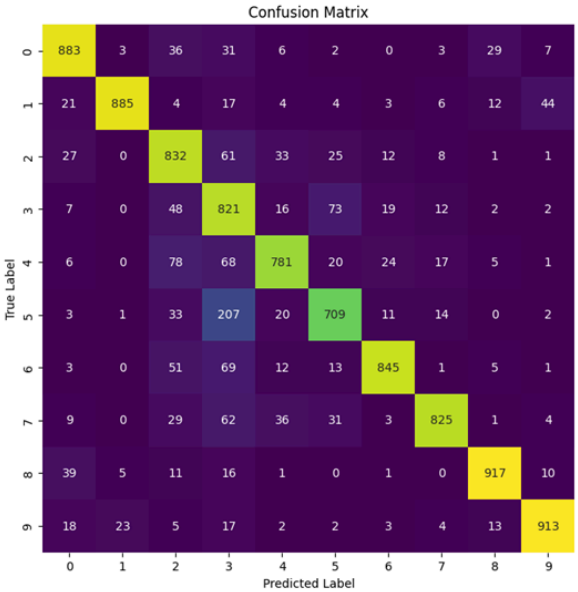
The Result of running the CIFAR-10 dataset through this model is also shown below.

```
# FINE-TUNING of the VGG-16 Model Integration


# Unfreeze the top layers for fine-tuning
for layer in vgg_model.layers[-4:]:  # Unfreeze the last 4 layers
    layer.trainable = True

✓ 0.0s
```

```
# COMPILE the Model

from tensorflow.keras.optimizers import Adam

# Compile with a lower learning rate
model_vgg.compile(
    loss='categorical_crossentropy',
    optimizer=Adam(learning_rate=1e-5),  # Lower learning rate
    metrics=['accuracy']
)

✓ 0.0s
```



- Optimizer/Loss: Adam / Cat. Crossentropy
- Epochs: 30
- Test accuracy: 0.84
- Test loss: 0.54
- F1-score and recall: 0.84



5. Transfer Learning with ResNet50

We wanted to compare the performances of VGG16 with a more complex and recent model and decided to go for ResNet50. As said in section 2, ResNet50 is well-knows for its approach to solve the vanishing gradient problem by introducing residual connections so the model converge faster. The ResNet architecture is widely used for classification models and transfer learning, making it the perfect candidate for our project.

5.1. ResNet50 architecture

ResNet-50 consists of 50 layers organized into a series of residual blocks, each containing convolutional layers and skip connections (shortcuts) that allow the network to learn residual mappings, enabling efficient training of deep models by facilitating gradient flow and preventing the vanishing gradient problem. See *Figure 5.1.1*.

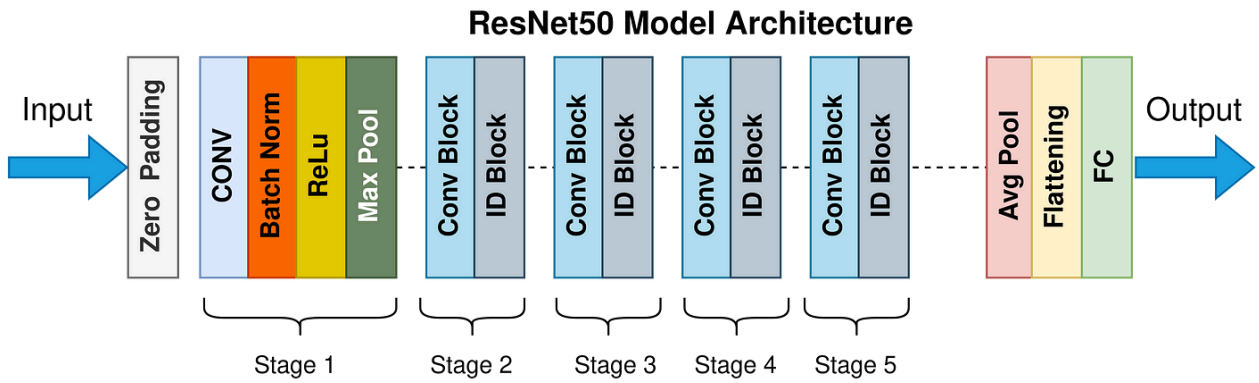


Figure 5.1.1 - ResNet50 internal structure

The main difficulties we had to face working with ResNet50 were the following:

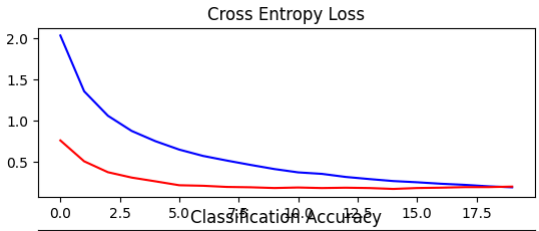
- ResNet50 has been trained on the imagenet dataset, which consists of 1000 classes. The model is therefore tuned to recognize differences between very different classes (a squirrel and a toothbrush for instance) but also very similar ones (water snake and vine snake). This level of complexity could be an obstacle when doing transfer learning.
- Like VGG16, ResNet50 requires images of shape 224x224 as input. An image upscaling was therefore required to use it with our 32x32 images dataset. We tested different approaches to resize the dataset pictures, but due to the large amount of images, most of these methods were very time-consuming. In the end we implemented the resizing operation using a Lambda layer at the input of the model. It gave us the best results performance-wise.

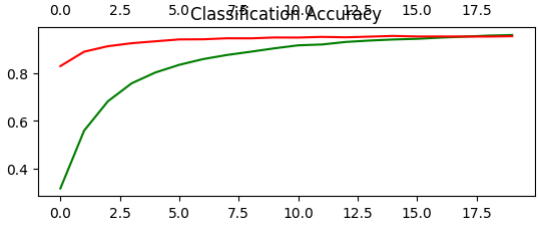
5.2. Transfer Learning

The convolution blocks structure of ResNet50 makes it easy to manipulate to do transfer learning. We went through the following steps:

- Load ResNet50 model from keras without the deep learning layers
- Freeze the weights on all the layers except the last convolution block (142 layers untrainable, the last convolution block trainable). This way the model can use the last block to capture details related to our 10 dataset classes.
- Add Dense 3 layers to extract high-level features into the final predictions, plus the softmax classification dense layer at the end.
- Add a Lambda layer as model input, to perform image resizing on the fly.

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lambda (Lambda)             (None, 224, 224, None)    0

 resnet50 (Functional)       (None, None, None, 2048   2358771
                             )

 flatten (Flatten)           (None, 100352)            0

 batch_normalization (Batch  (None, 100352)            401408
 Normalization)

 dense (Dense)               (None, 256)               2569036

 dropout (Dropout)           (None, 256)               0

 batch_normalization_1 (Bat  (None, 256)               1024
 chNormalization)

 dense_1 (Dense)             (None, 128)               32896

 dropout_1 (Dropout)         (None, 128)               0

 batch_normalization_2 (Bat  (None, 128)               512
 chNormalization)

 dense_2 (Dense)             (None, 64)                8256

 dropout_2 (Dropout)         (None, 64)                0

 batch_normalization_3 (Bat  (None, 64)                256
 chNormalization)

 dense_3 (Dense)             (None, 10)                650

=================================================================
Total params: 49723082 (189.68 MB)
Trainable params: 40909770 (156.06 MB)
Non-trainable params: 8813312 (33.62 MB)
_________________________________________________________________
```





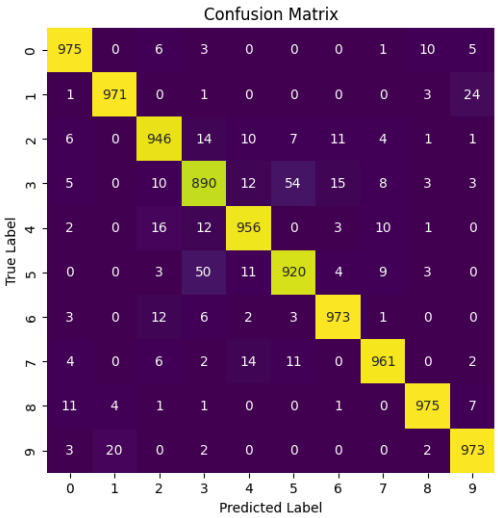


Figure 5.2.1. Transfer Learning on ResNet50

- Optimizer/Loss: RMSProp (LR: 1e-5) / Cat. Crossentropy
- Batch size: 32
- Epochs: 20
- Test accuracy: 0.95
- Test loss: 0.17
- F1-score and recall: 0.95

To train this model, we changed the Optimizer to RMSProp, which is a little easier to tune. We set a low learning rate of 1e-5, as we are only training the last convolution block and the dense layer. We also implemented all the previous techniques to reduce overfitting (Batch normalisation, dropout, data augmentation, etc.)

It's interesting to note the gap between green/blue curves and red ones. The test accuracy is better at the beginning, while the training accuracy catches up over the epochs. This is due to the Dropping layers, active only during the training, that decrease the model's accuracy at first to make it more robust to unseen data.

The final result is very good, with an accuracy, f1 score and recall over 0.95. The confusion matrix clearly shows how well the model is performing at differentiating images from the different classes, with still some misclassifications between cats and dogs. See *Figure 5.2.2*.

Figure 5.2.2. Classifying unseen images

5.3. Fine-Tuning

Finally, we tried to fine-tune the previous model by unfreezing all layers and re-training it with a low learning rate. See results on *Figure 5.3.1*.

Layer (type)	Output Shape	Param #
lambda (Lambda)	(None, 224, 224, None)	0
resnet50 (Functional)	(None, None, None, 2048)	2358771
flatten (Flatten)	(None, 100352)	0
batch_normalization (BatchNormalization)	(None, 100352)	401408
dense (Dense)	(None, 256)	2569036
dropout (Dropout)	(None, 256)	0
batch_normalization_1 (BatchNormalization)	(None, 256)	1024
dense_1 (Dense)	(None, 128)	32896
dropout_1 (Dropout)	(None, 128)	0
batch_normalization_2 (BatchNormalization)	(None, 128)	512
dense_2 (Dense)	(None, 64)	8256
dropout_2 (Dropout)	(None, 64)	0
batch_normalization_3 (BatchNormalization)	(None, 64)	256
dense_3 (Dense)	(None, 10)	650

Total params: 49723082 (189.68 MB)
Trainable params: 40909770 (156.06 MB)
Non-trainable params: 8813312 (33.62 MB)

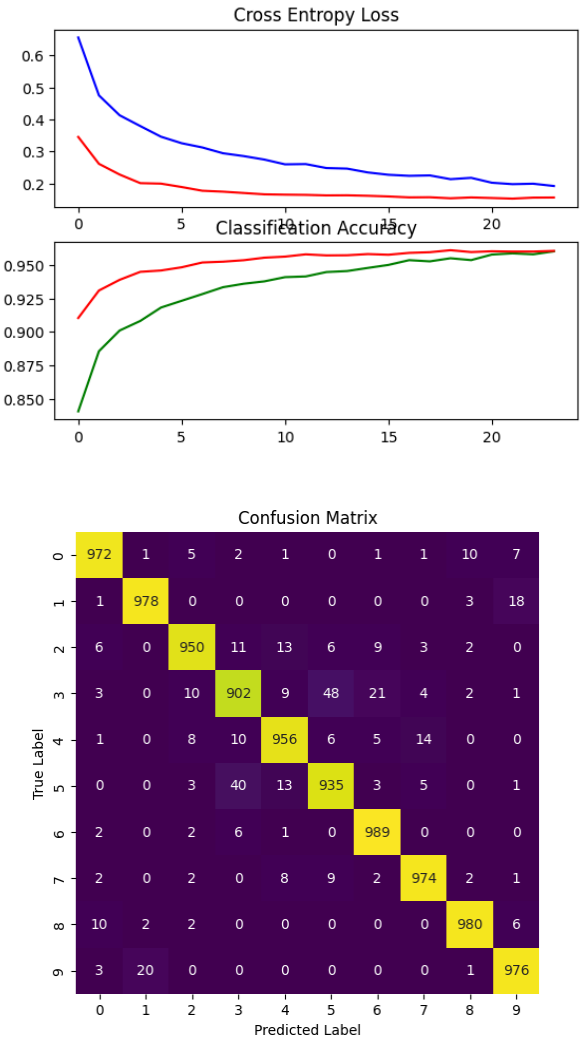


Figure 5.3.1. Transfer Learning on ResNet50

Unfortunately, while we do gain 1% accuracy, this process seems to mess up the weights and the model, while showing similar results, is not able to classify images properly anymore. See *Figure 5.3.2*.

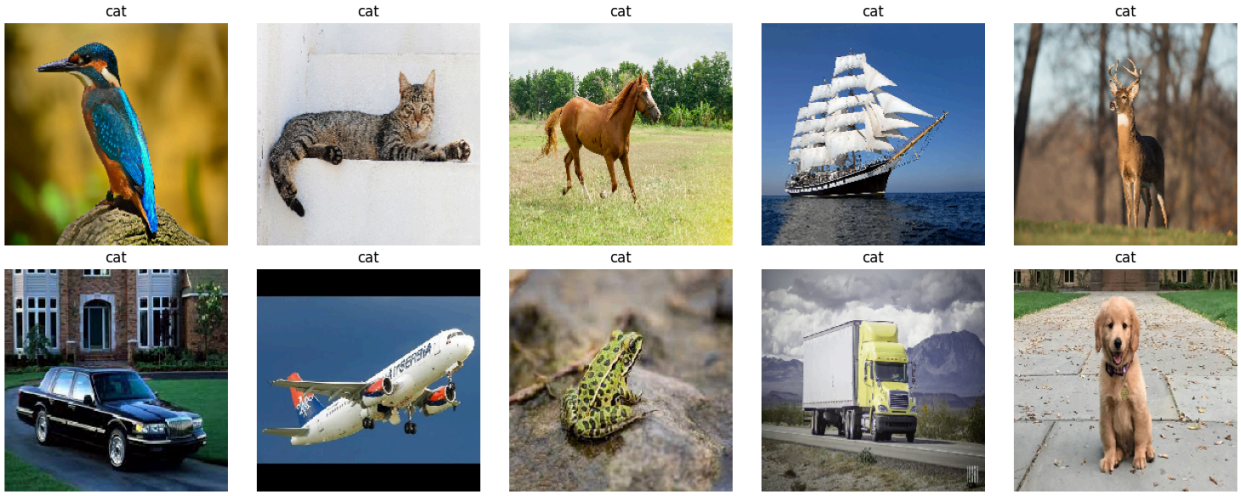


Figure 5.3.2. Misclassification after fine-tuning

6. Deployment

We did not have time to build a flask application, nor deploy the model on the cloud but we could create a script to demo our best model (ResNet50 with transfer learning) on Gradio. See *Figure 6.1*.

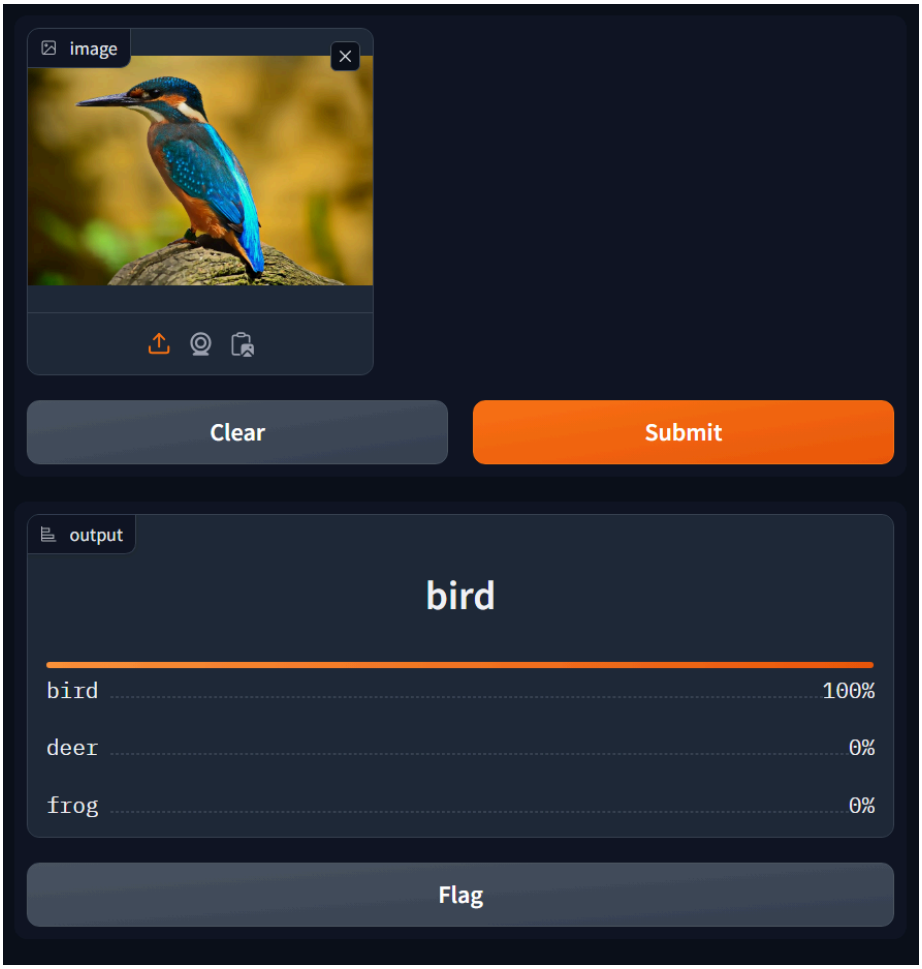


Figure 6.1. Model deployment on Gradio

Conclusion

Our experimentation demonstrated that custom-built CNNs can achieve reasonable performance with adequate data augmentation and regularization techniques.

However, leveraging pre-trained models such as VGG16 and ResNet50 significantly enhanced classification accuracy and model robustness.

ResNet50, in particular, showed excellent results, reaching 95% accuracy through transfer learning and careful fine-tuning.

Despite some challenges with overfitting and image resizing, this model emerged as the best performer, showcasing the power of advanced architectures in handling complex image classification tasks.

Future work could involve deploying the model in a practical application and further refining fine-tuning strategies for even better performance.