Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Deep Learning

Final Report

Authors:

José Antonio Ruiz Heredia

Héctor Carlos Flores Reynoso

Álvaro Honrubia Genilloud

Teachers:

Luis Baumel

Daniel Manrique

Martin Molina

Roberto Valle

Date:
April 11, 2025

Contents

1 The Problem

This final assignment focuses on applying *Deep Learning (DL)* techniques to solve image classification problems using the *xView* dataset. The objectives of the experiments include developing practical experience with *Keras*, designing and optimizing both *Feed-Forward Neural Networks (ffNNs)* and *Convolutional Neural Networks (CNNs)*, applying regularization techniques to reduce overfitting, and experimenting with transfer learning using pre-trained models such as *Xception*, *VGG*, and *ResNet*.

2 Dataset

The dataset used is a subset of the *xView* [1] dataset, one of the largest publicly available collections of satellite images. It contains approximately 1 million manually annotated objects in 60 categories, derived from complex global scenes and captured by the *WorldView-3* satellite at high spatial resolution. Originally released for the *DIUx xView 2018 Detection Challenge*, the dataset was designed to push the boundaries of computer vision by improving detection at lower resolutions, increasing learning efficiency, expanding object class diversity, and enabling classification in real-world satellite data.

- **Training set**: 21377 object images from 761 scenes
- **Test set**: 2635 object images from 85 scenes

2.1 Dataset Imbalance

The selected dataset of *xView 2018* has a noticeable class imbalance as discovered during the *Exploratory Data Analysis (EDA)*. Classes like *Small Car* and *Building* tend to have a significantly higher number of instances compared to others such as *Helicopter* or *Cargo Plane*.

This uneven distribution can bias object detection models toward the most frequent classes. For example, while a model may quickly learn to detect buildings and small cars due to abundant training data, it may struggle with recognizing *Helicopter* or *Dump Truck* because of their limited presence.

3 The Process

This project followed a step-by-step approach to training *Deep Learning (DL)* models for image recognition, increasing model complexity at each phase. All experiments were conducted on *Kaggle* using an ***NVIDIA Tesla P100 GPU*** accelerator [2].

We started with basic *Feedforward Neural Networks (ffNNs)*, exploring different optimizers such as *SGD*, *Momentum*, *RMSprop*, and *Adam*. After establishing a baseline, regularization techniques including normalization, *L1* and *L2* regularization, and dropout were applied to improve generalization.

In addition, we transitioned to *Convolutional Neural Networks (CNNs)*, fine-tuning convolutional parameters and introducing data augmentation. Finally, *Transfer Learning (TL)* was implemented using pretrained models, with additional layer tuning and improved data augmentation strategies.

4 Feedforward Neural Networks (ffNNs)

In these first experiments, we discuss the process and results of architectures using *Feedforward Neural Networks (ffNNs)* [3] for multilabel classification on the satellite dataset. The main objective was to optimize the performance of the model and the stability of the training. Over the course of the week, we implemented multiple patterns that allow us to decide the number of fully deep connected layers, each one with different shape. Once the basic architecture was decided, a variety of activations functions where tested including *ReLU*, *PReLU* and *LeakyReLU*. Being *ReLU* the best one. Finally, falling best practices *softmax* was used as an output layer to handle the multilabel classification task.

4.1 Base Model

The base model was trained with the default `batch_size` of 16, `early_stopping` with a patience of 40 epochs, and the *Adam* optimizer with its default settings. The model achieved a **Mean Accuracy** of 41.628%, a **Mean Recall** of 22.989%, and a **Mean Precision** of 26.909%. The classification performance can be visualized in Figure 1.

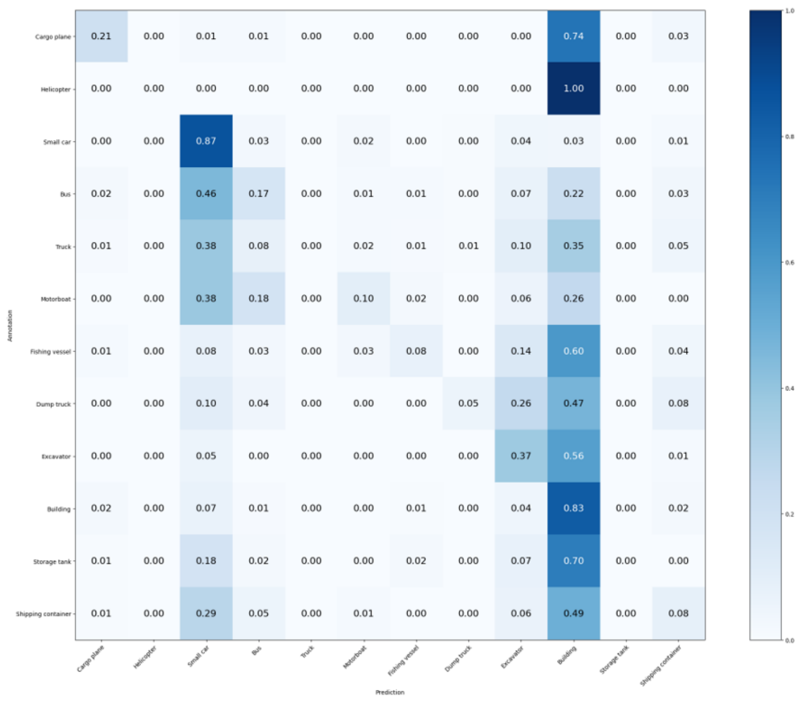


Figure 1: Confusion Matrix after training the base model.

Although the model performed well on certain classes like *Small car* (recall: 86.667%) and *Building* (recall: 83.06%), other classes like *Helicopter* and *Truck* showed no recognition, with recall values close to 0%.

4.2 Training & Optimizer Parameters

During training, multiple model integrations where tested with different parameter combinations.

After training our first modification, by increased the `batch_size` from 16 to 32, reduced `early_stopping` patience from 40 to 10 epochs, and tightened the *ReduceLROnPlateau* settings by setting the factor to 0.5 and patience to 3. We quickly realized that manual adjustments can become a problem, especially when dealing with multiple hyperparameters.

Although our model improved in performance, with a significant increase in **Mean Accuracy**, reaching 45.18%. It was hard to continuously continue improving and constantly understanding the relationship between accuracy (while avoiding overfitting) and our dataset.

We started to focus on automatizing hyperparameter search. Using *Keras Tunner* we were able to set a choice of options or range of values for different parameters using:

```
// Definition of range of parameters
hp.Float(’example_range’, min_value=0.7, max_value=0.9, step=0.1)

// Definition of choice parameters
hp.Choice(’example_choice’, [’adam’, ’rmsprop’, ’momentum’, ’sgd’])
```

Depending on the tuner algorithm, multiple in-between values may be added. We focused our research into 3 different sections:

- **Number of fully connected layers**: We set the ceiling to 5 fully connected layers with $2^(11 - x - n)$ output units where **n** is the current layer number and **x** is the modifier between 0 and 4 so different sizes could be tested.
- **Activation**: We decided to stick with *ReLU* and it's possible alternatives ***’ReLU’, ’LeakyReLU’, ’PReLU’***. However we assigned different *negative_slops* or *alpha_initializer* depending on the use case to try different alternatives.
- **Optimizer**: ***’adam’, ’rmsprop’, ’momentum’, ’sgd’*** where our selected choices since they were the ones we discussed in class, tweaking different `leaning_rate` (0.01, 0.005, 0.001, 0.0005, 0.0001), `beta1` and `beta2` parameters where appropriate.

Each trial was trained for 20 epochs and iterated using a *HyperBand* algorithm [4]. The best model with the greated value accuracy then would be fully trained 100 epochs.

This approach improved drastically the work efficiency. However, we quickly realized that this type of automatization can be memory exhaustive, even when using *Garbage Collectors* [5]. So we decided to split each experiment into the 3 phases above instead of single experiment.

- **Phase 1** took 50 minutes yielding 5 layers to be the best result
- **Phase 2** took 4 hours and 21 minutes yielding **ReLU** with a **0.005** `negative_slope`
- **Phase 3** took 1 hour and 55 minutes yielding *Adam* as the best optimizer with `learning_rate` of **0.0005**, `beta1` of **0.8999999999999999** and `beta2` of **0.9770000000000001**.

4.3 Best Model

In this final model, we incorporated five *Fully Connected (FC)* layers, using the same training and optimization settings discussed earlier. The number of units in the *FC* layers follows a decreasing pattern: 1024, 512, 256, 128 and 64.

Leaving the final model as described in Figure 2.

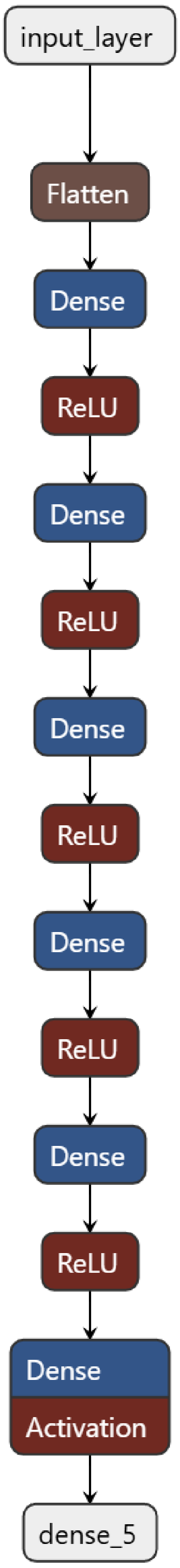


Figure 2: FFNN final Model.

Based on this architecture, this final model obtained the following metrics:

- **Mean Accuracy**: 58.746%
- **Mean Recall**: 47.808%
- **Mean Precision**: 51.183%

In addition, as shown in Figure 3, the training process presents a gradual improvement in **Mean Accuracy**, with noticeable fluctuations during the first 10 epochs. The model eventually reaches its peak **Mean Accuracy** of 58.746% near epoch 60.

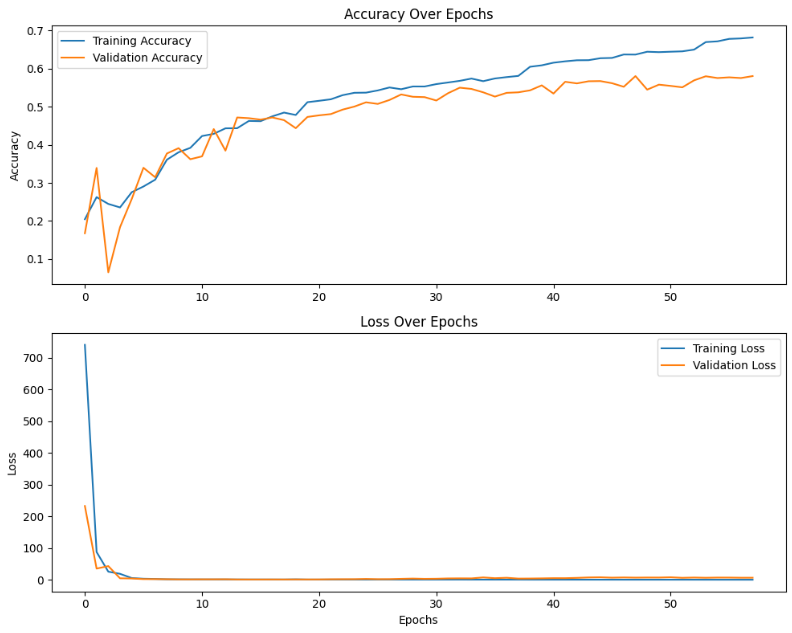


Figure 3: Training & Validation of *ffNN* architecture.

The addition of these layers enhanced the model's ability to classify certain categories such as *Building*, *Cargo plane*, *Excavator*, and *Small car* more effectively than previous, simpler models, as illustrated in Figure 4.

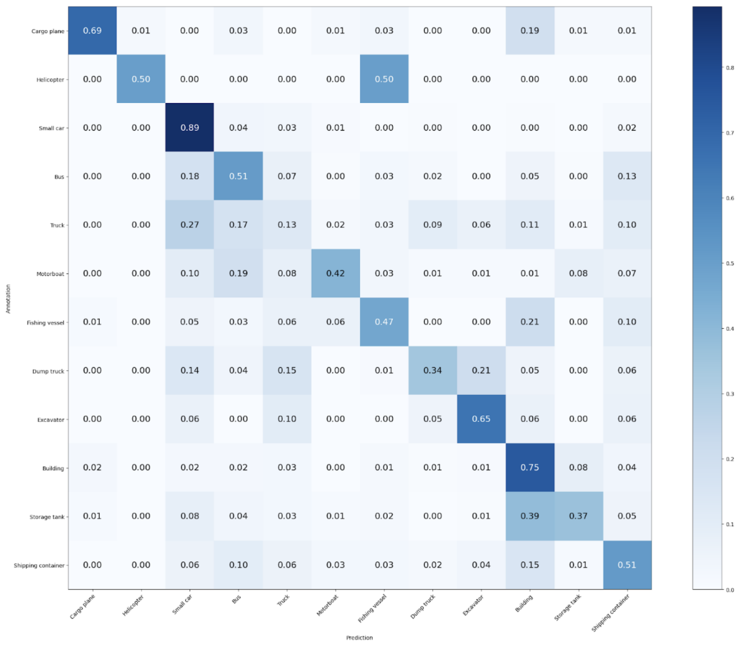


Figure 4: Confusion Matrix of best *ffNN* architecture.

4.4 Conclusion

From the experiments, it is evident that fine-tuning the optimizer and training parameters can significantly improve model performance, as it can be summarized in Table 1. The modifications such as increasing `batch_size`, adjusting `early_stopping` criteria, and optimizing the *Adam* parameters resulted in improvements in **accuracy**, **recall** and **precision**, particularly for classes like *Small car* and *Excavator*. Despite these results, challenges remain with certain underrepresented categories.

In Table 1, we show the top best 5 models iterations presented by *Keras Tunner*.

Model	**Mean Accuracy (%)**	**Mean Recall (%)**	**Mean Precision (%)**
Base Model	41.628	22.989	26.909
Model 1: Training Parameters	45.18	30.53	36.77
Model 2: Optimizer (*RMSprop*)	47.03	33.80	37.97
Model 3: Optimizer (*Adam*)	47.90	35.32	39.01
Model 4: Optimizer (*SGD*)	41.63	22.99	26.91
Model 5: Best Model	58.74	47.81	51.18

Table 1: Comparison of performance metrics on different *ffNN* experiments.

5 Regularization (Reg)

In the previous experiment, we can see validation loss is noisy, since it doesn't follow training loss, which potentially means the model is overfitting. To avoid this, in new experiments, we explore the effects of various *regularization* techniques [6] on model performance. The experiments include normalization after activation functions, *L1* and *L2* regularizations, dropout variations, and batch normalization strategies. We compare the results based on **mean accuracy**, **recall**, and **precision**. Each model has been tested under different configurations with a focus on achieving a balance between overfitting and underfitting.

As the previous iteration, we tried to fine tune our hyper-parameter search using *Keras Tuner*. Based on the previous knowledge we divided each phase into different experiments.

5.1 Normalization

For this experiment, we normalized the model's output after applying the activation function. Model 1 showed an improvement in performance on multiple metrics such as **Mean Accuracy** of 63.704%, **Mean Recall** of 61.372% and **Mean Precision** of 62.228% compared to *ffNN* model.

In contrast, we also experimented in Model 2 with the normalization applied before the activation function. The model showed a slight improvement in **Mean Accuracy** with 64.219% and **Mean Recall** with 60.595% compared to Model 1.

5.2 Regularization

The second experiment section was regularization, we used *Lasso Regularization (L1)* in Model 3, which led to a noticeable decrease in performance, particularly in **Mean Recall** with 24.560% and **Mean Precision** with 26.501%. This indicates that regularization may have been too aggressive.

Secondly, in Model 4 we tested *Ridge Regularization (L2)*, which resulted in an improved model performance compared to *L1*. It obtained a higher **Mean Accuracy** of 64.920% and better **Mean Recall** of 56.140%. The experiment was conducted with a 100-epoch training session.

5.3 Dropout

Dropout regularization was tested with `dropout_rate` of 0.1 and 0.5, as well as a descending rate starting from 0.4 to 0.1. The goal was to understand how varying dropout rates affected the model's ability to generalize.

In Model 5, with a `dropout_rate` of 0.1, the model achieved a **Mean Accuracy** of 62.254%, **Mean Recall** of 52.819%, and **Mean Precision** of 53.801%. This configuration provided a slight performance improvement compared to models with no dropout rates, suggesting a better balance between overfitting and underfitting with this regularization technique.

Similarly, in Model 6, with a higher `dropout_rate` of 0.5, the model achieved a **Mean Accuracy** of 64.219%, **Mean Recall** of 55.354%, and **Mean Precision** of 56.098%. This led to a noticeable performance improvement compared to the previous smaller `dropout_rate`.

5.4 Memory Problems

As each phase was completed, we started to have memory allocation problems. Some iterations were running out of memory and resources in *Kaggle*.

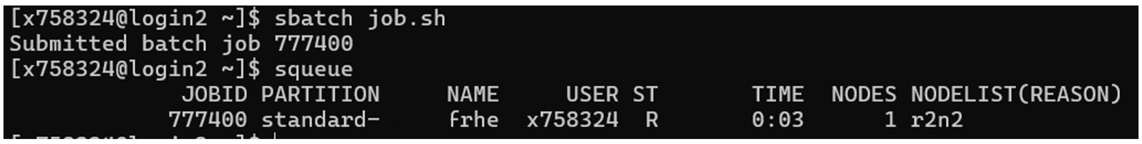
...

```
tensorflow.python.framework.errors_impl.ResourceExhaustedError: {{function_node
__wrapped__AddV2_device_/job:localhost/replica:0/task:0/device:GPU:0}}
failed to allocate memory [Op:AddV2] name:
...
```

This happen after 3 to 4 hours of work. Even when other iteration algorithms such as `RandomSearch` and `GridSearch` where used. Shortly after we noticed that *Keras Tuner* saved every trial, which made it possible to continue training after restarting the *Kernel*.

This continue to happen multiple times. So we decided to migrate to a robust machine.

5.5 CeSViMa Integration



```
[x758324@login2 ~]$ sbatch job.sh
Submitted batch job 777400
[x758324@login2 ~]$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
            777400 standard-     frhe  x758324  R       0:03      1 r2n2
```

Figure 5: *CeSViMa* integration.

After gaining access to *CeSViMa* [7], we encountered multiple errors along 4 failed trials.

1. ***The first trial*** took almost 2 days to initialize, queued on a Fri and processed until Sun with 0:05 runtime (Runtime error).
2. ***The second trial*** initialized the same day, ran for 46 minutes and exit after being unable to render images.
3. ***The third trial*** initialized the same day, ran for 33 seconds before existing because of a Python and Tensorflow library version incompatibility
4. ***The fourth trial*** was run with the full *Keras Tuner* hyperparameter code. It processed for 24 hours and *EXIT* automatically right after.

Since we were unable to continue training with *Keras Tuner*. Phase 3 (Dropout) was done manually over a series of manual hyperparameter tuning. In our opinion *CeSViMa* is lacking a lot of control over model development. Time is wasted trying to understand the configured environment inside, many available packages are out of date (ej: *TensorFlow* 2.15) which makes it difficult to find information and documentation.

Also logging is crucial. *CeSViMa* doesn't output any print logs with makes it a terrible solution when training multiple iterations. We couldn't complete any experiment successfully.

5.6 Best Model

Finally, Model 7 implements a descendant `dropout_rate`, starting from 0.4 up to 0.1 in its deep layers.

Leaving the final model as described in Figure 6.

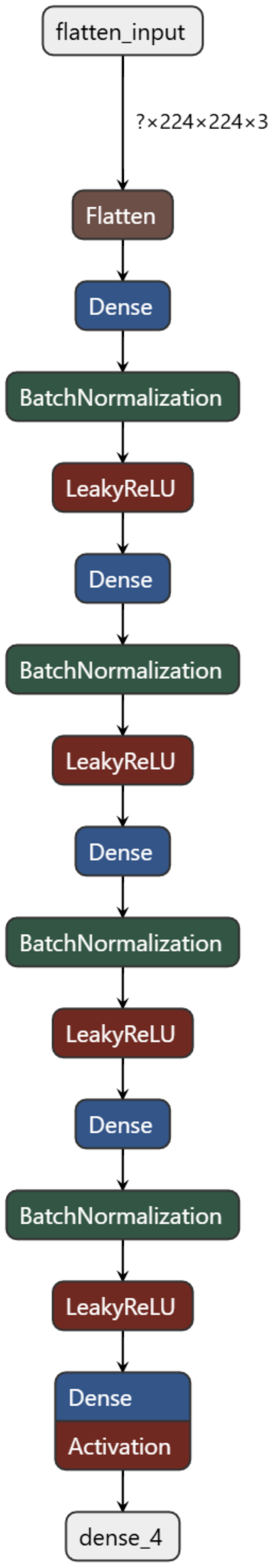


Figure 6: *Regularization* final Model.

With this approach we obtained the following metrics:

- **Mean Accuracy**: 65.389 %
- **Mean Recall**: 55.578%
- **Mean Precision**: 57.930%

As seen in 7, it shows a significantly more stable and generalizable learning process compared to the previous run without regularization techniques. Training and validation accuracy improve steadily, showing that the model is not biased. In this scenario, overfitting is not noticeable compare to the previous version since accuracy kept rising while validation accuracy plateaued.

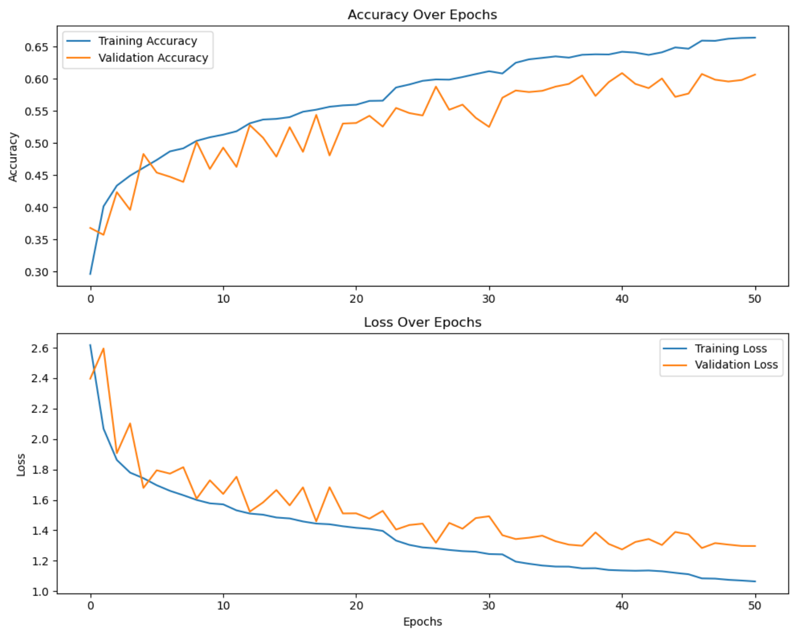


Figure 7: Training & Validation of *Regularization* architecture.

This results are noticed by the progressive `dropout_rate` that particularly enhanced **Mean Recall** and **Mean Precision**, with notable improvements in the *Cargo plane*, *Small car*, and *Fishing vessel* categories as it can be visualized in Figure 8.

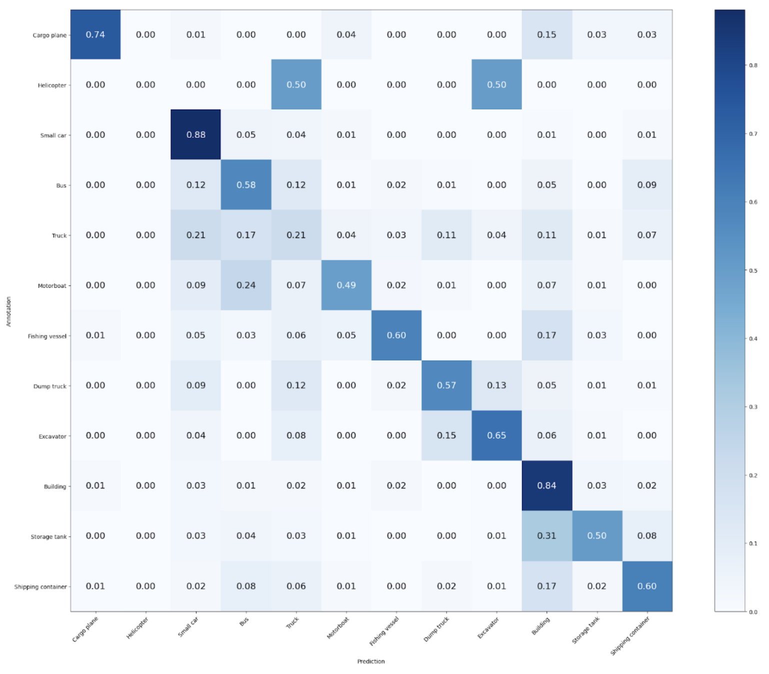


Figure 8: Confusion Matrix of best *Regularization* architecture.

5.7 Conclusion

Regularization experiments gave mixed results as shown in Table 2. While normalization (both before and after activation functions) and dropout methods provided clear improvements in the metrics, *Lasso regularization (L1)* appeared too aggressive, leading to underfitting. *Ridge regularization (L2)* provided improvements in generalization. Dropout with descendant rate showed the most consistent results.

In Table 2, we show the top best 5 models iterations.

Model	**Mean Accuracy (%)**	**Mean Recall (%)**	**Mean Precision (%)**
Base Model	57.58	45.14	49.33
Model 1: After Activation	63.70	61.37	62.23
Model 2: Before Activation	64.22	60.59	59.08
Model 3: *L1*	42.24	24.56	26.50
Model 4: *L2*	64.92	56.14	57.66
Model 5: Dropout at 0.1	62.25	52.82	53.80
Model 6: Dropout at 0.6	64.22	55.35	56.09
Model 7: Best Model	65.39	55.58	57.93

Table 2: Comparison of performance metrics on different *Regularization* experiments

.

6 Convolutional Neural Networks (CNNs)

Following the regularization experiments, this section explores the application of *Convolutional Neural Networks (CNNs)* to the image classification task. The regularization study revealed that techniques such normalization and dropout could substantially improve model performance, with the best regularization model achieving a **Mean Accuracy** of 65.389%.

While these improvements are notable, *CNNs* offer a distinct advantage by exploiting the spatial structure of image data. By learning hierarchical feature representations through convolutional layers, *CNN* has the potential to capture intricate patterns and achieve superior classification accuracy compared to *FC* networks. Furthermore, a custom-designed *CNN* allows for greater flexibility in architecture design, enabling the incorporation of specific regularization techniques and optimization strategies tailored to the unique characteristics of the image dataset. This section explores various *CNN* architectures and parameters, with the goal of optimizing performance for the classification problem [8].

6.1 Convolutional Parameters

The base convolutional model was implemented adding a *Conv2D* layer [9] with the following specific parameters: `filters` (32), `stride` (1,1), `bias_initializer` (”zeros”), and `kernel_size` (1,1). Subsequent experiments involved changing parameters to observe their effects. Specifically, the `stride` parameter was modified to (2,2) and (3,3) to analyze the impact of larger strides on feature extraction and spatial reduction. The `kernel_size` was adjusted to (3,3) and (5,5) to evaluate how larger receptive fields influence the model's ability to capture spatial hierarchies. *BatchNormalization* was incorporated to stabilize training and potentially allow for the use of higher `learning_rate`.

6.2 Convolutional Layers

We also experimented varying the number of convolutional layers in the network architecture. In this study, we implemented models with two, three, four, and six convolutional layers to observe how depth influences the classification accuracy. The architecture with three convolutional layers included a reduction in `batch_size` to 32 and the addition of a *GlobalAveragePooling2D* layer, which aimed to reduce the number of parameters and computational load. The four-convolutional layer model implemented *LeakyReLU* activation to address the vanishing gradient problem and *SpatialDropout2D* for regularization. The six convolutional layer model was configured with `filters` of 64, 128, 256, 256, 512, and 512.

6.3 Data Augmentation

We implemented data augmentation techniques to improve the generalization and reduce overfitting using *ImageDataGenerator* [10]. Specific augmentation parameters included a `rotation_range` of 30 degrees to introduce rotational invariance, `width_shift` and `height_shift` ranges of 0.2 to provide translational invariance, and a `zoom_range` of 0.1 to handle scale variations. These transformations were applied to the training data to artificially increase its diversity and robustness.

6.4 Best Model

This model incorporates a convolutional feature extractor followed by *Fully Connected (FC)* layers. The convolutional part begins with a *Conv2D* layer with 32 filters and `kernel_size` (1,1), followed by *LeakyReLU* activation and *MaxPooling*. Subsequent convolutional layers use filters [64, 128, 256, 256, 512, 512] with `kernel_size` (3,3), *LeakyReLU* activation, and *MaxPooling*. *GlobalAveragePooling2D* is used to reduce the spatial dimensions before the *FC* layers. These layers consist of *FC* layers with decreasing units and dropout for regularization, and *BatchNormalization*. The output layer uses softmax activation for classification.

Leaving the final model as described in Figure 9.

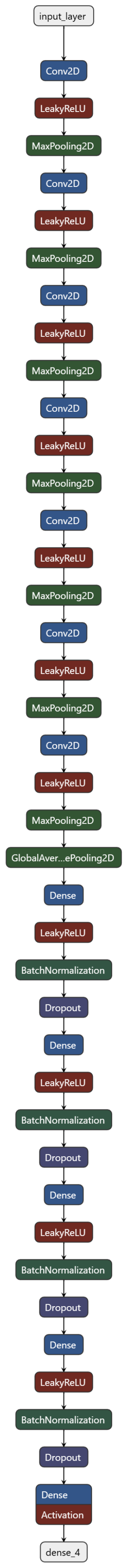


Figure 9: Convolutional Neural Network final Model.

With this approach we got the following metrics:

- **Mean Accuracy**: 79.233%
- **Mean Recall**: 75.668%
- **Mean Precision**: 77.523%

Figure 10 shows training accuracy rapidly increasing and plateaus around 0.8 after 50 epochs, while validation accuracy plateaus lower, around 0.75, after 40 epochs. Furthermore, training loss decreases steadily, but validation loss plateaus and slightly increases after 40 epochs.

Given this clear indication of overfitting occurring relatively early in training (around epochs 40-50), a more strict early stopping strategy is crucial. Instead of allowing the model to continue training for the full 150 epochs, it should be halted much earlier, close to epochs 40-50, to prevent the model from memorizing noise in the training data.

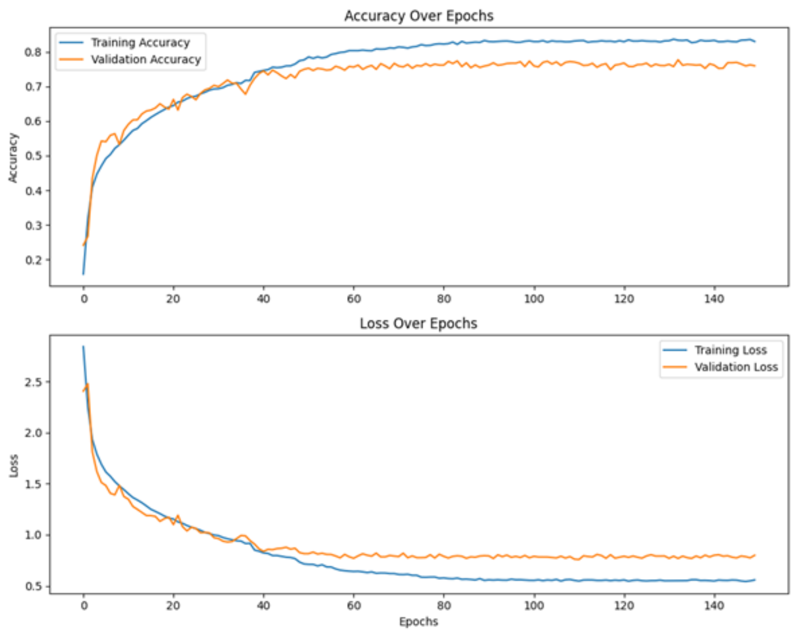


Figure 10: Training & Validation of *Convolutional Neural Network* architecture.

Moreover, Figure 11 reveals that most classes are being correctly predicted with high accuracy, such as Cargo plane (90%), Small car (91%), Excavator (85%), Building (91%) and Storage tank (90%).

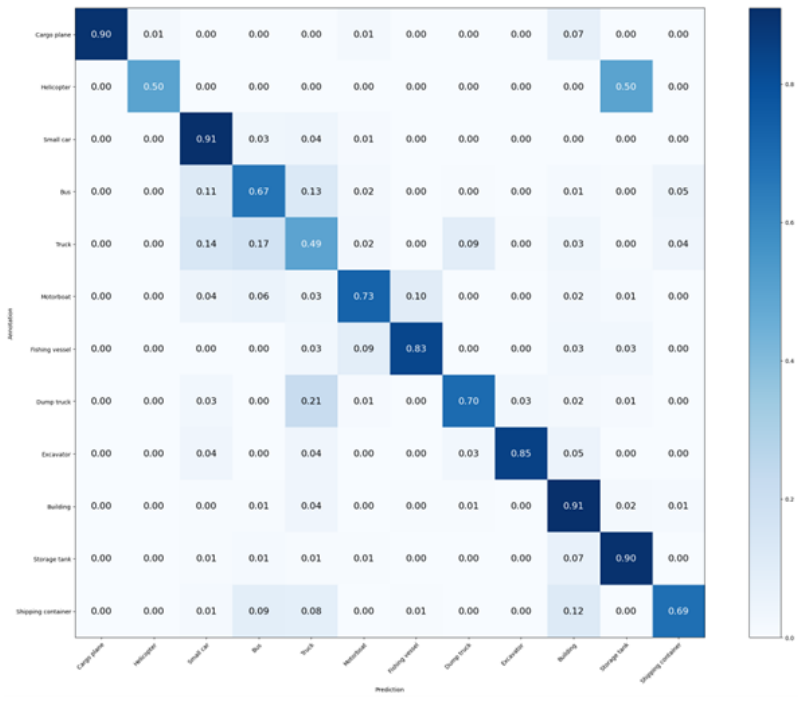


Figure 11: Confusion Matrix of best *Convolutional Neural Network* architecture.

6.5 Conclusion

The exploration of *CNNs* involved a systematic evaluation of various architectural parameters, including `stride`, `kernel_size`, *BatchNormalization*, and the number of convolutional layers. Our experiments revealed that increasing the depth of the network, as seen from Model 5 (3 Layers) with a **Mean Accuracy** of 74.32% to Best Model (6 Layers) achieving 79.23%, generally led to improved performance compared to simpler base models as seen on Table 3. This underscores the capacity of deeper *CNNs* to learn more complex features from the image data.

The "Best Model" identified within this *CNN* exploration achieved a **Mean Accuracy** of 79.23%, a **Mean Recall** of 75.66%, and a **Mean Precision** of 77.52%. This represents a significant improvement over the initial base *CNN* model (67.17% **Mean Accuracy**) and the models incorporating individual parameter adjustments. This optimized *CNN* architecture likely benefited from a combination of the explored parameters and potentially other fine-tuning strategies not explicitly detailed in the individual model descriptions.

However, similar to our observations during training, the potential for overfitting remains a crucial consideration. While the final "Best Model" demonstrates strong performance metrics, monitoring the training and validation curves for this specific configuration is essential to ensure robust generalization. Implementing early stopping based on validation performance would be a prudent strategy to prevent overfitting and ensure the model's effectiveness on unseen data.

Model	**Mean Accuracy (%)**	**Mean Recall (%)**	**Mean Precision (%)**
Base Model	67.17	61.54	68.66
Model 1: Stride	66.09	60.74	63.48
Model 2: Batch Normalization	66.18	62.53	66.83
Model 3: Kernel Size	65.15	55.11	58.13
Model 4: 2 Layers	66.09	57.67	58.29
Model 5: 3 Layers	74.32	66.39	67.74
Model 7: 4 Layers	77.50	73.45	75.94
Model 8: Best Model	79.23	75.66	77.52

Table 3: Comparison of performance metrics on different *Convolutional Neural Network* experiments

.

7 Transfer Learning (TL)

Training *CNN* models can be rather challenging, especially when starting from scratch. As mentioned in the previous section, training *CNN* architectures reusing the definitions from the Regression exercise caused overfitting. A better approach was reducing the total number of dense layers to improve our results.

Instead of continuing trying to find an appropriate architecture, *Transfer Learning (TL)* allows you to leverage pre-trained *CNNs*, such as *VGG*, *ResNet*, or *Inception*,that have already learned rich feature representations from large-scale datasets like *ImageNet*.

The idea behind is leverage the broad dataset used to train the model so we can achieve strong performance with less training time and data.

The experiment consists of two phases.

7.1 Model Selection

Currently we have a selection of 5 models: ***VGG16***, ***VGG18***, ***Xception***, ***ResNet50*** and ***InceptionResNetV2***.

Among the models tested, *ResNet50* [14] and *InceptionResNetV2* [15] stand out with the highest **Mean Accuracy** scores of 77.456% and 77.035%, respectively, indicating strong overall performance. Followed by *VGG16* and *VGG19* that show moderate performance, being *VGG19* [12] better than *VGG16* [11] across all metrics, especially in **Mean Accuracy** (73.667%) and **Mean Precision** (70.981%).

Finally *Xception* [13], despite its architectural complexity, performs worse than the other architectures with the lowest **Mean Recall** (68.008%) and **Mean Accuracy** (71.749%). This establishes *ResNet50* as the most effective and reliable option among the evaluated models.

7.2 Trainable Layers

After picking *ResNet50* as a default model, we tried to further optimized the model by unfreezing some of the deeper layers, this theoretically will allow the model to refine decision boundaries.

Models like *ResNet50* are trained over general datasets, fine tuning image details in the last layers by unfreezing "x" number of layers we able capture our own dataset peculiarities to further specialize the model to our own necessities.

At first, we started unfreezing 50 layers which led to a drastic performance drop, with a mean accuracy of only 12.35%. This likely due to overfitting and the disruption of well-generalized low-level features.

In contrast, unfreezing just 35 layers resulted in a significant performance boost, achieving a mean accuracy of 81.01%. Unfortunately, the pattern didn't continue to raise. Unfreezing only 15 layers also yielded strong results, with a mean accuracy of 77.08%, this suggest that 35 layers is the optimal configuration.

This suggests that selectively fine-tuning the deeper layers strikes a balance between preserving robust, generalizable features and adapting high-level representations to the specific patterns of our dataset, ultimately leading to better model specialization and improved accuracy.

7.3 Data Augmentation Discussion

Finally, we improve data augmentation techniques [16] as a final addition to our training pipeline. It aimed at improving not only generalization but also addressing the limitations of relatively small datasets.

Although it significantly improved the model's ability to learn robust, invariant features. It also introduced important considerations during validation. In training our first iteration, we noticed how the mean accuracy started to decrease in the validation set with the same configuration. Later we understood that this is actually expected.

Since augmentation is only applied to the training set, not to the validation set, it ensures that performance metrics reflect the model's ability to generalize to unaltered real-world data.

However augmented validation data can lead to misleading performance estimates. Overall, the addition of data augmentation helped reduce overfitting and led to a more resilient model without compromising the integrity of validation results.

After different experimentation, we implemented the following data augmentation configuration.

```
data_augmentation = Sequential([
  RandomRotation(factor=0.1),           # Rotate ±10% of 360 degrees (±36°)
  RandomTranslation(0.2, 0.2),          # Translate ±20% horizontally and vertically
  RandomZoom(0.2),                      # Zoom in/out by ±20%
  RandomFlip("horizontal")              # Flip horizontally
 ])
```

These transformations were chosen to simulate the types of variability the model might encounter in real-world scenarios, without distorting the content in a way that could confuse the *CNN*.

7.4 Best Model

Our most effective model configuration combined the robust architecture of *ResNet50* with a fine-tuning strategy that involved unfreezing the last 35 layers. This approach allowed the network to retain the powerful feature extraction capabilities learned during pretraining, while also adapting more precisely to our specific dataset.

Leaving the final model as described in Figure 12.

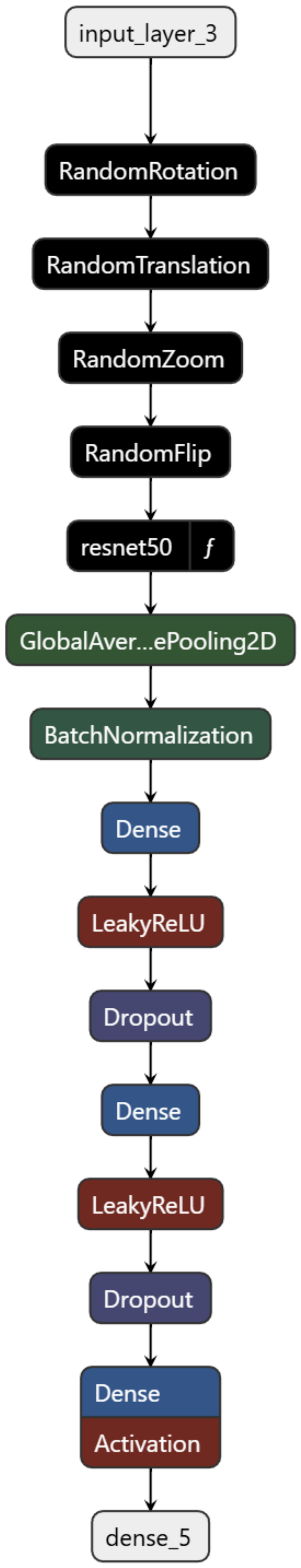


Figure 12: Transfer Learning final Model.

With this approach we got the following metrics:

- **Mean Accuracy**: 81.010%
- **Mean Recall**: 76.919%
- **Mean Precision**: 84.852%

Figure 13 shows improvements in both training and validation accuracy over the 54 epoch. Furthermore, the validation accuracy and loss consistently outperform the training set, suggesting that regularization techniques are working.

This pattern indicates that the model is not overfitting and is generalizing well to unseen data. In conclusion, the training process shows stable convergence and good performance.

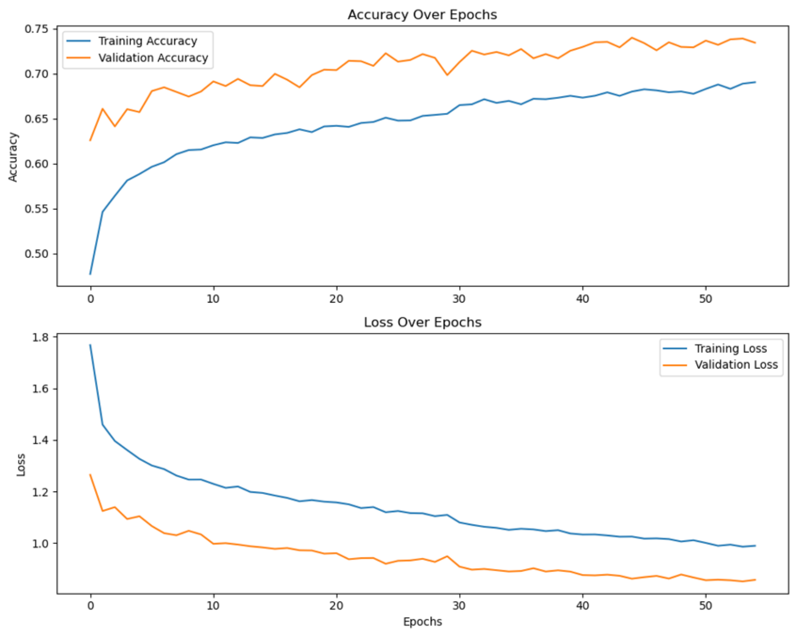


Figure 13: Training & Validation of *Transfer Learning* architecture.

Moreover, Figure 14 reveals that most classes are being correctly predicted with high accuracy, such as Cargo plane (94%), Small car (93%), Fishing vessel (87%), and Building (92%).

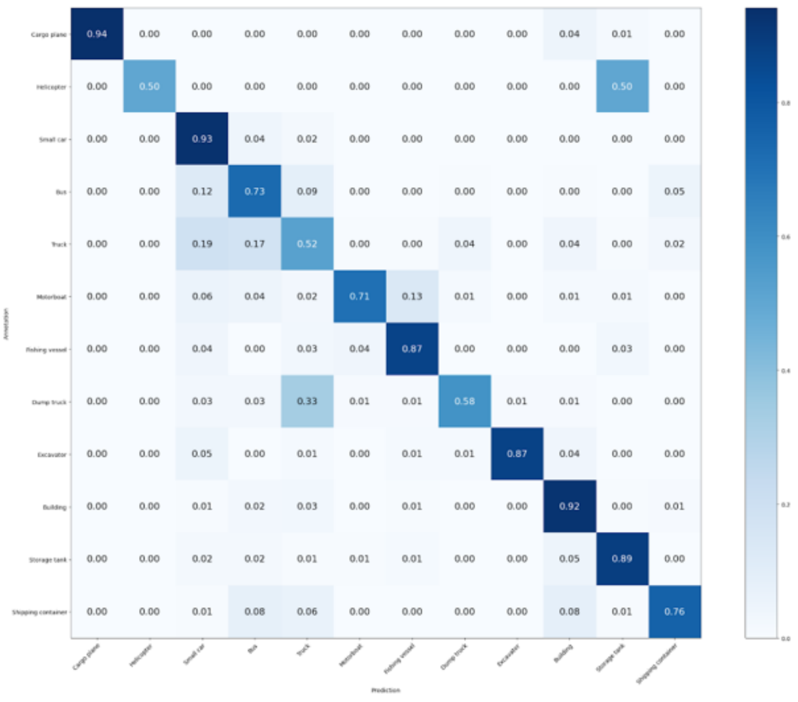


Figure 14: Confusion Matrix of best *Transfer Learning* architecture.

7.5 Conclusion

TL experiments demonstrate the advantage of leveraging pre-trained models, as shown in Table 4. Among all configurations, the fine-tuned *ResNet50* with the last 35 layers unfrozen achieved the highest scores across all key metrics, including a **Mean Accuracy** of 81.01%, **Mean Precision** of 84.85%, and **Mean Recall** of 76.91%. Compared to other tested models like *VGG19* and *InceptionResNetV2*, which showed decent but slightly lower results, this configuration provided the best balance between generalization and specialization. However, aggressive unfreezing of layers led to overfitting, highlighting the importance of fine-tuning depth. The integration of data augmentation also played a crucial role in reducing overfitting and boosting the model's robustness.

Model	**Mean Accuracy (%)**	**Mean Recall (%)**	**Mean Precision (%)**
Base Model	57.58	45.14	49.33
Model 1: VGG16	72.54	69.12	69.73
Model 2: VGG19	73.67	70.26	70.98
Model 3: ResNet50	77.46	73.82	75.78
Model 4: Xception	71.75	68.01	69.75
Model 5: InceptionResNetV2	77.04	73.65	79.34
Model 7: Best Model	81.01	76.91	84.85

Table 4: Comparison of performance metrics on different *Transfer Learning* experiments

.

8 Project Conclusion

This study explored the application of various *Deep Learning (DL)* techniques to image classification using the *xView* dataset. The experiments were conducted in a step-by-step manner, progressively increasing the complexity of the models.

Initially, we experimented with *Feedforward Neural Networks (ffNNs)*, modifying the impact of different optimizers and training parameters on model performance. The best-performing *ffNN* model achieved a **Mean Accuracy** of 58.746%, demonstrating the feasibility of using *ffNN* for this classification task, while also revealing challenges in accurately classifying underrepresented classes.

Subsequently, we implemented *Convolutional Neural Networks (CNNs)* to leverage their proficiency in extracting spatial hierarchies from image data. The optimization of convolutional parameters and the inclusion of data augmentation techniques led to substantial improvements in classification accuracy. The "Best Model" *CNN* architecture achieved a **Mean Accuracy** of 79.23%, significantly outperforming the *ffNNs* models.

Finally, we used *Transfer Learning (TL)*, using pre-trained *ResNet50*, to further enhance performance. This approach demonstrated the effectiveness of leveraging knowledge acquired from large-scale datasets to improve classification accuracy and efficiency.

Throughout the study, we faced the challenge of the class imbalance present in the *xView* dataset. Future work should focus on developing strategies to mitigate the bias towards frequent classes and improve the recognition of infrequent object categories. Furthermore, continued exploration of advanced regularization techniques and architectural optimizations could lead to further improvements in the accuracy and robustness of *DL* models for satellite image classification.

References

[1] xView (2018). "DIUx xView 2018 Detection Challenge". https://xviewdataset.org.

[2] Kaggle (2024). "Kaggle: Your Machine Learning and Data Science Community". https://www.kaggle.com.

[3] TensorFlow (2024). "Keras: The high-level API for TensorFlow". https://www.tensorflow.org/guide/keras?hl=en.

[4] Journal of Machine Learning Research 18 (2018). "Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization". 1-52. http://jmlr.org/papers/v18/16-558.html

[5] StackOverflow (2019) - "Keras model training memory leak" https://stackoverflow.com/questions/58137677/keras-model-training-memory-leak

[6] TensorFlow (2024). "Regularizer — TensorFlow Core v2.0 API Docs". https://www.tensorflow.org/api_docs/python/tf/keras/Regularizer.

[7] CeSViMa (2024). "CeSViMa". https://www.cesvima.upm.es.

[8] TensorFlow (2024). "Convolutional Neural Network (CNN)". https://www.tensorflow.org/tutorials/images/cnn?hl=en.
https://www.tensorflow.org/tutorials/images/cnn?hl=en

[9] TensorFlow (2024). "Conv2D Layer — TensorFlow Core v2.0 API Docs". https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D.

[10] TensorFlow (2024). "ImageDataGenerator — TensorFlow Core v2.0 API Docs". https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator.

[11] TensorFlow (2024). "VGG16 — TensorFlow Core v2.0 API Docs". https://www.tensorflow.org/api_docs/python/tf/keras/applications/VGG16.

[12] TensorFlow (2024). "VGG19 — TensorFlow Core v2.0 API Docs". https://www.tensorflow.org/api_docs/python/tf/keras/applications/VGG19.

[13] TensorFlow (2024). "Xception — TensorFlow Core v2.0 API Docs". https://www.tensorflow.org/api_docs/python/tf/keras/applications/Xception.

[14] TensorFlow (2024). "ResNet50 — TensorFlow Core v2.0 API Docs". https://www.tensorflow.org/api_docs/python/tf/keras/applications/ResNet50.

[15] TensorFlow (2024). "InceptionResNetV2 — TensorFlow Core v2.0 API Docs". https://www.tensorflow.org/api_docs/python/tf/keras/applications/InceptionResNetV2.

[16] TensorFlow (2024). "Data augmentation". https://www.tensorflow.org/tutorials/images/data_augmentation?hl=en.