# Machine Learning Models for Image Classification

## Table of Contents



## Overview

This guide explains two models that can classify images into 5 different groups (labeled 0-4):

- **CNN (Deep Learning)** - uses neural networks
- **Random Forest** - uses decision trees

Both models work with 100x100 pixel color images and create predictions with performance reports.

## CNN Model (Deep Learning)

### How the CNN Works

```
# CNN Setup
image_size = (100, 100, 3)  # 100x100 color images
filter_size = (3, 3)        # Small filters
pool_size = (2, 2)          # Pooling size

Layers:
1. Conv2D(64 filters) + MaxPooling2D
2. Conv2D(128 filters) + MaxPooling2D
3. Conv2D(256 filters) + BatchNorm + MaxPooling2D
4. Conv2D(512 filters) + MaxPooling2D
5. Flatten + Dropout(0.5)
6. Dense(256, relu) + Dropout(0.5)
7. Dense(5, softmax) # Final output
```

### CNN Settings

| Setting | Value |
|---|---|
| **Optimizer** | Adam |
| **Loss Function** | Sparse Categorical Crossentropy |
| **Dropout Rate** | 0.5 |

| Setting | Value |
|---|---|
| **Batch Norm** | Yes (layer 3) |
| **Early Stop** | Yes (wait 10 epochs) |
| **Learning Rate** | Reduces when stuck |
| **Class Weights** | {0: 1.0, 1: 1.5, 2: 1.0, 3: 1.0, 4: 2.5} |

### Training Settings

- **Epochs**: Number you choose
- **Batch Size**: Number you choose
- **Safety Features**:
  - Stops early to prevent overfitting
  - Adjusts learning rate automatically
  - Saves the best model

### How to Run CNN

```
python3 cnn.py <epochs> <batch_size>

# Example:
python3 cnn.py 50 32
```

### Main CNN Functions

**`CNN()`**

Creates and sets up the neural network.

**`CNN_train(train_data, validation_data, epochs, batch_size, class_weight)`**

Trains the CNN with:

- `train_data`: Training images and labels
- `validation_data`: Test images and labels
- `epochs`: How many times to train
- `batch_size`: How many images at once
- `class_weight`: Balance for different classes

**`create_report(trained_model, validation_data)`**

Tests the model and saves results to `reports/reports.csv`.

## Random Forest Model

### How Random Forest Works

```
# Random Forest Setup
n_estimators = 500      # Number of trees
max_depth = 20          # How deep trees can go
random_state = 69       # For consistent results
n_jobs = -1            # Use all computer cores
```

### Random Forest Settings

| Setting | Value |
|---|---|
| **Number of Trees** | 500 |
| **Max Tree Depth** | 20 |
| **Use All Cores** | Yes |
| **Random Seed** | 69 |
| **Data Format** | Flatten images to 1D |

### How to Run Random Forest

```
python3 random_forest.py
```

### Main Random Forest Functions

**`RF()`**

Sets up the Random Forest classifier.

**`RF_train(train_data)`**

Trains the model:

- Changes images to flat format
- Trains on the flat data

**`create_report(trained_model, validation_data)`**

Tests the model and saves results to `RFreports/reports.csv`.

## How to Install

### Install Everything You Need

```
pip install -r requirements.txt
```

# How to Use

## Steps to Run

1. **Run CNN**:

```
python3 cnn.py 50 32  # 50 training rounds, 32 images at once
```

2. **Run Random Forest**:

```
python3 random_forest.py
```

## Shared Functions

`load_data(file)`

Loads data from files:

- **Input**: File name ('train', 'validation', 'test')
- **Output**: Images and labels (if available)
- **Processing**: Makes pixel values between 0 and 1

`generate_confusion_matrix(real, prediction, names, precision)`

Creates a confusion matrix picture:

- Shows how well the model did
- Saves picture in reports folder
- Backs up code

`make_predictions(trained_model, validation_data, test_images, validation_accuracy)`

Makes final predictions:

- Tests on validation data
- Creates confusion matrix
- Saves predictions to `predictions.csv`

## What We Measure

- **Accuracy**: How often the model is right
- **Confusion Matrix**: Detailed breakdown of results
- **Loss** (CNN only): How much error there is

# Model Comparison

| Feature | CNN | Random Forest |
|---|---|---|
| **Training Time** | Longer | Shorter |
| **Memory Used** | More | Less |
| **Image Quality** | Better | Good |
| **Easy to Understand** | Hard | Easy |
| **Settings to Tune** | Many | Few |
| **Computer Use** | GPU/CPU | CPU only |

## When to Use Each

### CNN

- **Best for**: Big datasets, best performance
- **Needs**: GPU (recommended)
- **Training Time**: 2-3 minutes (with good GPU)

### Random Forest

- **Best for**: Quick testing, limited resources
- **Needs**: Multi-core CPU
- **Training Time**: 2-3 minutes (with CPU)

## Common Problems

### CNN Issues

1. **Out of Memory**: Use smaller batch_size
2. **Overfitting**: Early stopping helps (already included)
3. **Slow Training**: Use GPU if you have one

### Random Forest Issues

1. **Not Enough Memory**: Use fewer trees
2. **Takes Too Long**: Make trees less deep or use fewer
3. **Bad Results**: Try different image processing

## How It Works

### Loading Data

Both models use the same way to load data:

```
def load_data(file):
    # Read the CSV file
    dataFrame = pd.read_csv(f'{file}.csv')
    images, labels = [], []
```

```
    # Look at each image
    for _, row in dataFrame.iterrows():
        img_id = row["image_id"]
        img_path = f"{file}/{img_id}.png"
        try:
            # Load and normalize image
            img_array = np.array(Image.open(img_path)) / 255.0
            images.append(img_array)

            # Add label if we have it
            if "label" in dataFrame.columns:
                labels.append(row["label"])
        except Exception as e:
            print(f"Error loading image {img_path}: {e}")
            continue

    return (np.array(images), np.array(labels)) if labels else
np.array(images)
```

### Creating Reports

Both models create detailed reports with:

- How well they performed
- Confusion matrix pictures
- Time-stamped report folders
- Code backups for reference

### Making Predictions

1. **Train Model**: Learn from training data
2. **Test**: Check on validation data
3. **Confusion Matrix**: Create visual analysis
4. **Final Predictions**: Predict on test data
5. **Save**: Write predictions to CSV file

## Making Models Better

### CNN Improvements

- BatchNorm for stable training
- Dropout to prevent overfitting
- Early stopping to avoid overtraining
- Smart learning rate changes

### Random Forest Improvements

- Use all CPU cores at once
- Good tree depth for best results

- Smart memory usage
- Fast predictions

# Results and Reports

## CNN Results

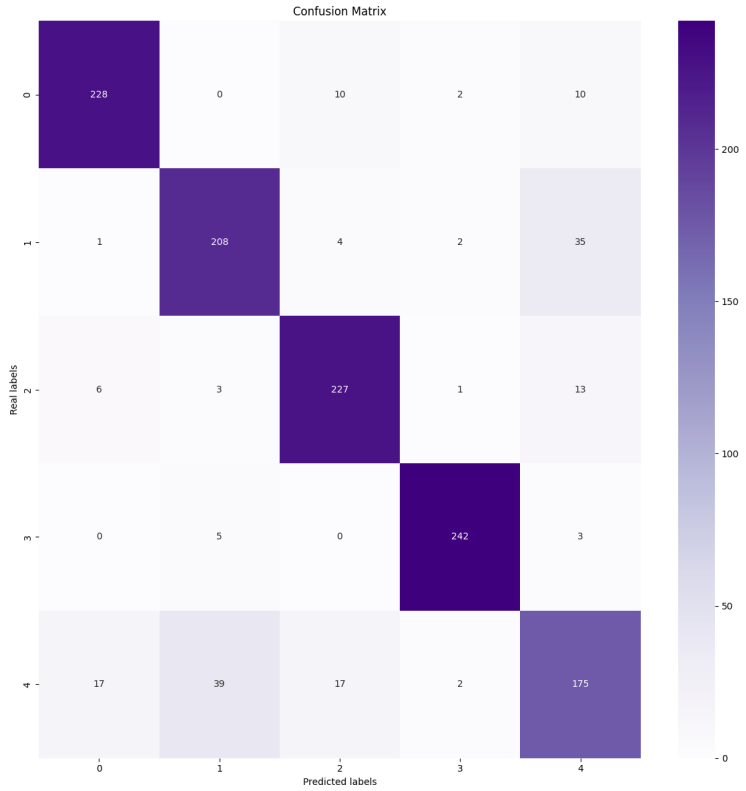
### Things That Didn't Work:

- Tried using two separate CNN models (one to find class 4, another for classes 0-3), but this gave poor results due to errors building up between the two steps.
- Tried data augmentation to create more training examples, but this was done incorrectly and didn't help the model learn better.

### Things That Worked Well:

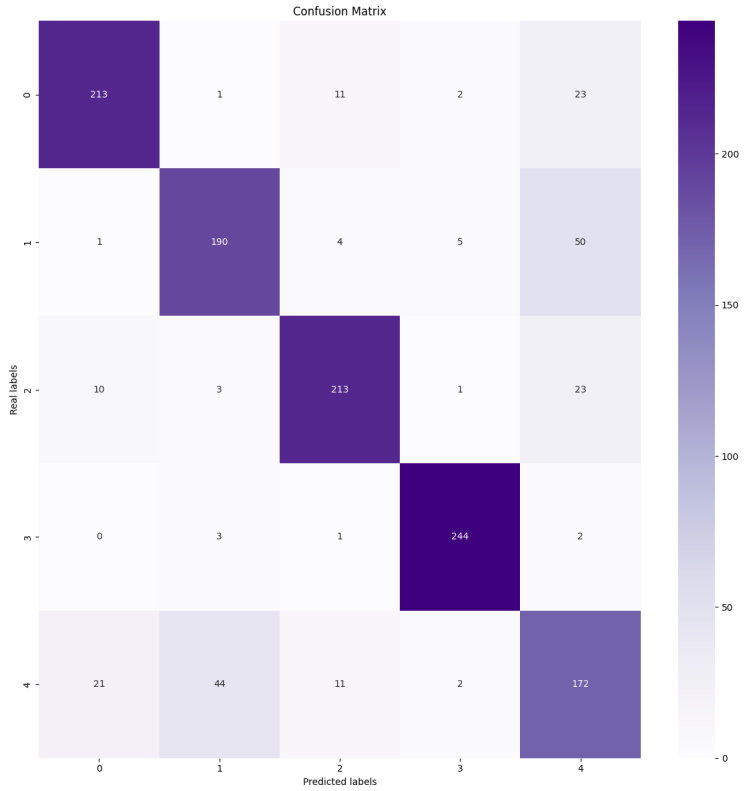
| Category | Setting | Value |
|---|---|---|
| **Model** | Conv Filters | [64, 128, 256, 512] |
| | Dense Units | 256 |
| | Dropout Rate | 0.3 |
| **Training** | Optimizer | Adam |
| | Loss Function | Sparse Categorical Crossentropy |
| | Early Stop Wait | 7 epochs |
| | Learning Rate Drop | 0.2 |
| **Class Balance** | Classes 0-3 | 1.0 |
| | Class 4 | 2.5 |
| **Data** | Normalize | /255.0 |
| | Data Type | float32 |

Accuracy: 86.4%



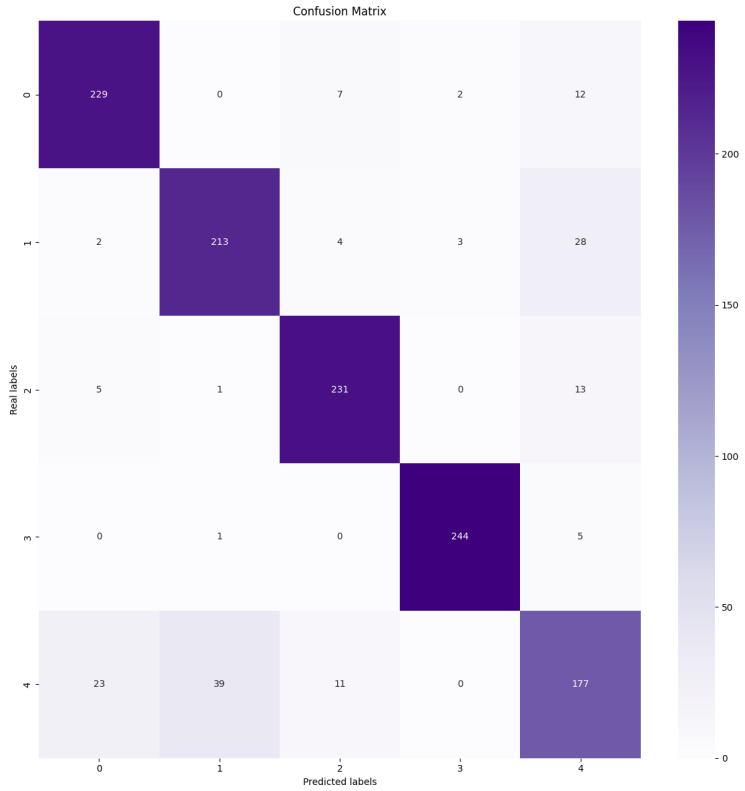

| Category | Setting | Value |
|---|---|---|
| **Model** | First Filter Size | (5, 5) |
| | Conv Filters | [64, 128, 256, 512] |
| | Dense Units | 256 |
| | Dropout Rate | 0.2 |
| **Training** | Optimizer | Adam |
| | Loss Function | Sparse Categorical Crossentropy |
| | Early Stop Wait | 7 epochs |
| | Learning Rate Drop | 0.2 |
| | LR Drop Wait | 3 epochs |
| **Class Balance** | Classes 0-3 | 1.0 |
| | Class 4 | 2.5 |
| **Data** | Normalize | /255.0 |
| | Data Type | float32 |

Accuracy: 82.5%




| Category | Setting | Value |
|---|---|---|
| **Model** | Conv Filters | [32, 64, 128, 256] |
| | Dense Units | 256 |
| | Dropout Rate | 0.3 |
| **Training** | Optimizer | Adam |
| | Loss Function | Sparse Categorical Crossentropy |
| | Early Stop Wait | 10 epochs |
| | Learning Rate Drop | 0.2 |
| | LR Drop Wait | 3 epochs |
| **Class Balance** | Classes 0-3 | 1.0 |
| | Class 4 | 1.5 |
| **Data** | Normalize | /255.0 |
| | Data Type | float32 |

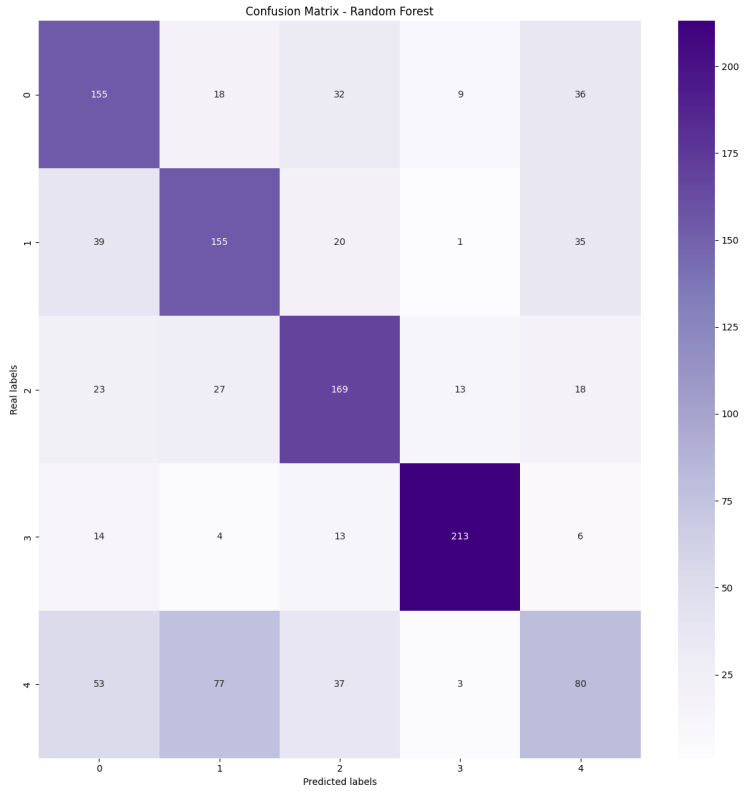
Accuracy: 87.5%




# Random Forest Results

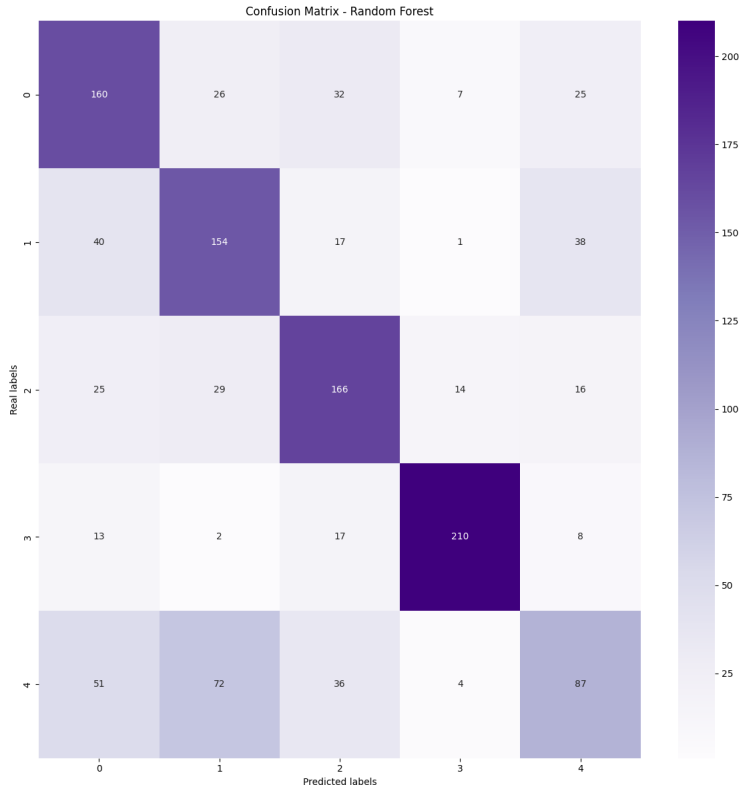
| Category | Setting | Value |
|---|---|---|
| **Model** | Number of Trees | 500 |
| | Max Tree Depth | None |
| | Data Format | Flatten to 1D |
| **Training** | Random Seed | 69 |
| | Use All Cores | Yes |
| | Bootstrap | True |
| **Data** | Normalize | /255.0 |
| | Data Type | float32 |
| | Processing | Image flattening |

Accuracy: 61.0%




| Category | Setting | Value |
|---|---|---|
| **Model** | Number of Trees | 500 |
| | Max Tree Depth | None |
| | Data Format | Flatten to 1D |
| **Training** | Random Seed | 69 |
| | Use All Cores | Yes |
| | Bootstrap | True |
| **Data** | Normalize | /255.0 |
| | Data Type | float32 |
| | Processing | Image flattening |

Accuracy: 62.2%




## Summary

Both models work well for image classification but have different strengths. The CNN performs better for complex image tasks, while Random Forest is faster and easier to understand. Both include good reporting, error handling, and optimization features that make them ready to use in real projects.