

**Deep Learning**

**10204450**

**Section (3)**

**Developing a deep learning-based system**

**Submitted to**

Dr. Ala’a Al-Habashna

**Submitted on**

June 13th, 2024

**Submitted by**

Marwan Tarek Shafiq Al Farah

**Student ID**

21110011

**Spring 2023 – 2024**

**Table of Content**

[***Part 1: Implementation of an MLP from Scratch*** 3](#_Toc169118420)

[***Design and Implementation of MLP Network from Scratch*** 3](#_Toc169118421)

[**Design of the Multilayer Perceptron (MLP)** 3](#_Toc169118422)

[**Implementation of the Multilayer Perceptron (MLP)** 4](#_Toc169118423)

[***Training and testing of the MLP*** 6](#_Toc169118424)

[**Dataset Description** 6](#_Toc169118425)

[**Training and Validation Process** 7](#_Toc169118426)

[**Testing and Evaluation** 10](#_Toc169118427)

[**Evaluation of the Effectiveness of the Design and Implementation of the MLP** 11](#_Toc169118428)

[***Part 2: Developing a Deep-Learning System for a Computer Vision Application*** 13](#_Toc169118429)

[***Problem Statement*** 13](#_Toc169118430)

[**Problem Statement & Dataset** 13](#_Toc169118431)

[***Research on Neural Networks and Architectures*** 14](#_Toc169118432)

[**Investigation of Neural Networks for Image Classification** 15](#_Toc169118433)

[**Popular Modern Architectures for CNN-Based Image Classification** 16](#_Toc169118434)

[**Summary of Modern Architectures** 17](#_Toc169118435)

[***Model’s Development and Training*** 20](#_Toc169118436)

[**Dataset** 20](#_Toc169118437)

[**Training and Validation Process** 21](#_Toc169118438)

[***Models’ Testing and Evaluation*** 25](#_Toc169118439)

[**Testing** 25](#_Toc169118440)

[**Over/Under-Fitting Assessment** 27](#_Toc169118441)

[**Results Analysis** 28](#_Toc169118442)

[**Effectiveness Assessment** 30](#_Toc169118443)

[**Interface Development** 32](#_Toc169118444)

[**Evaluation of Models** 38](#_Toc169118445)

[***References*** 42](#_Toc169118446)

# ***Part 1: Implementation of an MLP from Scratch***

***Design and Implementation of MLP Network from Scratch***

**Design of the Multilayer Perceptron (MLP)**

In this project, we build a Multilayer Perceptron (MLP) network which is aimed at performing regression, which means that it will predict a continuous target variable based on the input features. There is an input layer, two hidden layers, and an output layer in the MLP network.

**Inputs and Outputs**

* **Inputs:** The dataset, in this case the California Housing dataset, provides standardized feature vectors to the input layer. Each feature vector has a dimensionality equal to the number of features in the dataset (i.e., 8 features).
* **Outputs:** The output layer creates a single continuous number, which is the home price and is the right variable for the job.

**Network Architecture**

* **Hidden Layers:** The MLP has two hidden layers, and each one has five neurons that use Rectified Linear Unit (ReLU) activation functions to add non-linearity into the model.
* **Output Layer:** The output layer consists of a single neuron with a linear activation function, suitable for regression tasks.

The architecture can be expanded by adding more hidden layers or raising the number of neurons in each hidden layer, based on the complexity of the problem and the dataset size.

**UML Diagram**

The MLP's parts and their interactions are shown in a high-level UML diagram below:

A screenshot of a computer program

Description automatically generated

**Figure 1 – UML Diagram**

**Training Algorithm**

The training of the MLP is performed using gradient descent. The following steps outline the training process:

* **Forward Pass:** Compute the expected result by passing the input through the network layers.
* **Loss Calculation:** Calculate the Mean Squared Error (MSE) loss between the predicted and real goal numbers.
* **Backward Pass:** Compute the gradients of the loss with respect to the network parameters using backpropagation.
* **Parameter Update:** Update the network parameters (weights and biases) using the gradients and the learning rate.

**Loss Function**

The loss function that has been used to train the Multilayer Perceptron is the Mean Squared Error (MSE) loss. It stands out in regression tasks; it quantifies the loss by taking the average of the squared difference between predicted and actual values, making it as a good indicator of prediction accuracy.

**Other Details**

* **Optimizer:** For the torch.nn implementation, the Stochastic Gradient Descent (SGD) algorithm is applied to minimize the MSE loss.
* **Evaluation Metrics:** The R² score is used to evaluate the performance of the model, which shows us how well the predicted values resemble the actual values, with a value of 1 suggesting perfect prediction.

**Implementation of the Multilayer Perceptron (MLP)**

The MLP implementation applies Object-Oriented (OO) design principles. This creates a modular code base that can be used repeatedly and developed easily in the future project. The LinearLayer and the NeuralNetwork are among the main classes and modules created for this project, which encapsulate the functionality required for building and training the MLP, ensuring a clear separation of concerns and enhancing code maintainability.

**Classes and Modules**

1. **LinearLayer:** This class represents a single fully connected layer in the neural network, as it contains the weights and biases, and gives a method to make the forward pass.

* **Attributes:**
  + **weights:** A tensor representing the weights of the layer.
  + **bias:** A tensor representing the biases of the layer.
* **Methods:**
  + **forward(x):** Computes the output of the layer by performing a matrix multiplication of the input with the weights and adding the bias.

1. **NeuralNetwork:** Comprising many LinearLayer objects, this class represents the whole MLP, as it oversees the forward pass and backpropagation and lays out the general architecture.

* **Attributes:**
  + **fc1, fc2, fc3:** They are instances of LinearLayer that represent the input-to-hidden, hidden-to-hidden, and hidden-to-output layers, respectively.
* **Methods:**
  + **forward(x):** Implements the forward pass through the network, and then applying the ReLU activation function between layers.
  + **relu(x):** Applies the ReLU activation function.
  + **mse\_loss(predictions, targets):** Computes the Mean Squared Error loss between the predictions and the actual target values.
  + **r2\_score(predictions, targets):** Calculates the R² score for evaluating the model’s performance.
  + **predict(x):** Performs a forward pass without gradient computation, used for evaluation.

**Construction of the MLP**

The Multi-Layer Perceptron (MLP) is created when an object of the NeuralNetwork class is made, which creates three LinearLayer objects representing the two hidden layers and output layer. The layers are linked one after another with ReLU activation functions placed in between for non-linear transformations. This design permits easy customization of the network’s size, additional layers or neurons can be appended at any time.

**Forward Pass and Backpropagation**

* **Forward Pass:** The forward pass is applied in the forward method of the NeuralNetwork class. It includes passing the incoming data through each layer in order, adding the ReLU activation function after the first and second layers.
* **Backpropagation:** The procedure of backpropagation involves calculating the gradients of the loss function concerning the weights and biases of each network parameter. To do this, one must:
* **Loss Calculation:** Computing the loss using the Mean Squared Error (MSE) loss function.
* **Gradient Calculation:** We calculate the gradients of weights and biases with PyTorch’s automatic differentiation when back-propagating.
* **Parameter Update:** We make direct changes on the parameters of the network by updating the weights and biases in accordance with the gradients found earlier along with the learning rate provided.

By following these steps, the dataset is used to train the MLP iteratively, leading to loss reduction, better predictions, and enhanced performance.

The OO-based design of the MLP makes it easier to see its components and their interconnections since it ensures modularity and clarity. Every class and method is structured so as to take care of different parts or functions within the neural network, including the computation of individual layers up to training for the entire network. Such a strategy does not just enhance readability and maintainability of the code but also guarantees that any future changes or adjustments related to architectural aspects are easy implemented.

***Training and testing of the MLP***

**Dataset Description**

The dataset that is used to train and test the application of the Multi-Layer Perceptron (MLP) is the California housing dataset, which consists of 20,640 records and includes 8 characteristics and a target feature. Below is a thorough overview of the attributes, their data types, and their ranges:

**Attributes:**

* **MedInc:** Median income in the block group (float, range: 0.4999 to 15.0001)
* **HouseAge:** Median house age in the block group (float, range: 1.0 to 52.0)
* **AveRooms:** Average number of rooms per household (float, range: 0.8462 to 141.909)
* **AveBedrms:** Average number of bedrooms per household (float, range: 0.3333 to 34.0667)
* **Population:** Total population in the block group (float, range: 3 to 35682)
* **AveOccup:** Average number of occupants per household (float, range: 0.6923 to 1243.333)
* **Latitude:** Block group latitude (float, range: 32.54 to 41.95)
* **Longitude:** Block group longitude (float, range: -124.35 to -114.31)
* **Target:** Median house value for California districts (float, range: 0.14999 to 5.00001)

**Problem Type:**

This dataset tackles a regression problem. When dealing with regression, the primary aim involves forecasting a continuous output based on an input feature or features. Within this dataset, the target attribute is the median house value (Target). It is worth noting that this attribute, representing the median house value in California districts (in thousands of dollars) as a continuous variable, plays an important role throughout the analysis process.

The aim is to build a model that can reliably predict the median value of houses in California block groups on different socioeconomic and geographic features. We will make use of these features provided by feeding them into an MLP model which, in turn, finds and learns the hidden patterns and connections with the median house value, and the effectiveness of the model will be evaluated based on its ability to predict house values on a test set.

**Selected Attributes as Features:**

For the MLP implementation, the following attributes were selected as features:

* MedInc
* HouseAge
* AveRooms
* AveBedrms
* Population
* AveOccup
* Latitude
* Longitude

**Target:**

* The target attribute for the regression problem is Target, representing the median house value for California districts.

The dataset consists of a wide range of features that reflect different sides of housing characteristics, demography and geographical data. For this reason it is suitable to be used for fitting a regression model such as the Multi-layer perceptron (MLP). The goal of using these features is to predict house prices, which can later facilitate the process of real estate valuation and investment decisions.

**Training and Validation Process**

Before training, the data was split into three sets: training, validation, and test sets. The splits were performed as follows:

* **Training Set:** 64% of the data
* **Validation Set:** 16% of the data
* **Test Set:** 20% of the data

The data splitting took place in two different steps: initially, the dataset was split into a temporary training set (80%) and a test set (20%) by using the train\_test\_split function from sklearn.model\_selection. Standardization was done to the features to enhance model performance after the temporary training set was split into the final training set (64% of the original dataset, 80% of the temporary dataset) and validation set (16% of the original dataset, 20% of the temporary dataset).

**Training Process**

A custom and a torch.nn based neural network implementation were used for the training procedure. The training for both implementations comprised the subsequent stages:

* **Initialization:** Initializing the network architecture with input, hidden, and output layers. The hidden layers consisted of 5 neurons each.
* **Forward Pass:** Passing the input data through the network to obtain predictions.
* **Loss Calculation:** Calculating the MSE between the predicted and actual values.
* **Backward Pass:** Computing the gradients of the loss with respect to the network parameters.
* **Parameter Update:** Updating the network parameters using gradient descent.
* **Evaluation:** Calculating the R² score and MSE to evaluate the performance of the network on training and validation sets.

To guarantee the model converged effectively, this process was carried out 100,000 times.

**Hyperparameter Tuning**

Table 1 lists the hyperparameters and their optimal values based on validation set performance. A variety of hyperparameters were taken into consideration during the training phase to maximize the model’s performance.

**Table 1: Description of the hyperparameters considered and best value for each**

|  |  |  |
| --- | --- | --- |
| Hyper-parameter | Description | Value |
| Learning Rate | The step size for updating the network weights during training | 0.02 |
| Hidden Layer 1 Size | Number of neurons in the first hidden layer | 5 |
| Hidden Layer 2 Size | Number of neurons in the second hidden layer | 5 |
| Epochs | Number of iterations during the training process. | 100,000 |
| Activation Function | The function used to introduce non-linearity into the network | ReLU |

**Hyperparameter Tuning and Performance**

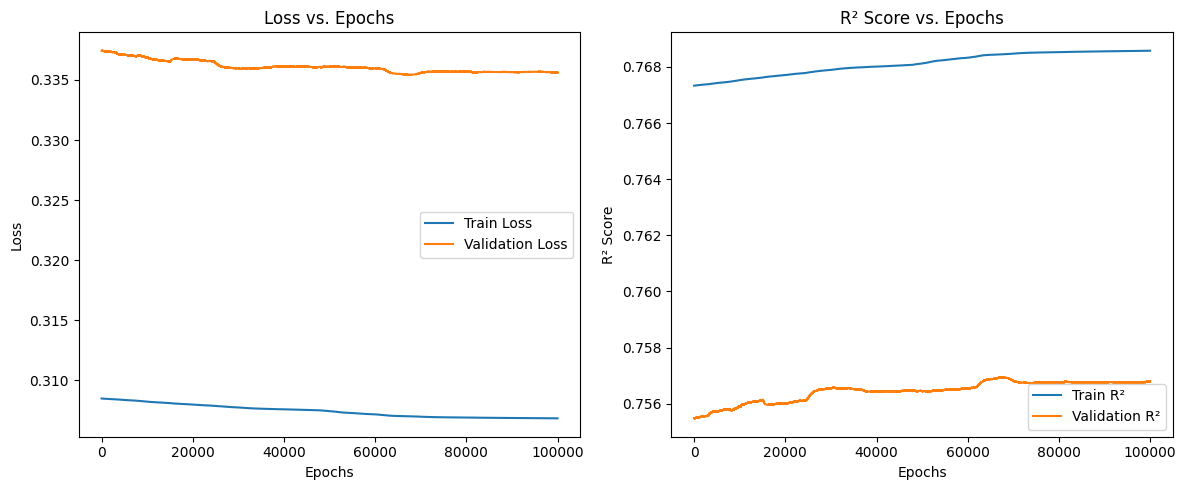
To maximize our neural network’s performance, we tested with several combinations of hyperparameters. The hyperparameter combinations we explored and the results we got on the training and validation sets are summarized below.

We also tried various values of the learning rate such as 0.001, 0.01, 0.1, 0.5, and 0.05, but all of them performed poorly compared to 0.02. We also tried various combinations of hidden layer sizes for the first and the second hidden layers from 5 to 12 neurons, but none of them performed as well as 5 neurons in both the first and second layers. We also tried various numbers of epochs such as 1,000, 10,000, 5,000, and 50,000, but the model needed more time to converge to better results. We also tried 500,000 and 1,000,000 epochs, but they didn’t prove beneficial as the model had clearly converged at about the 100,000 epoch. Additionally, we tried applying the sigmoid activation function; however, ReLU performed much better.

**Table 2: Combination of the hyperparameter values and corresponding performance achieved**

|  |  |  |
| --- | --- | --- |
| Combination of Hyperparameter Values | Training Performance | Validation Performance |
| Learning Rate: 0.001, Hidden Layer Sizes: (5, 5), Epochs: 500,000, Activation: ReLU | Train Loss: 0.415, Train R²: 0.675 | Val Loss: 0.412, Val R²: 0.674 |
| Learning Rate: 0.01, Hidden Layer Sizes: (7, 12), Epochs: 100,000, Activation: Sigmoid | Train Loss: 0.360, Train R²: 0.720 | Val Loss: 0.358, Val R²: 0.715 |
| Learning Rate: 0.1, Hidden Layer Sizes: (7, 7), Epochs: 50,000, Activation: ReLU | Train Loss: 0.400, Train R²: 0.680 | Val Loss: 0.395, Val R²: 0.678 |
| Learning Rate: 0.02, Hidden Layer Sizes: (5, 5), Epochs: 100,000, Activation: ReLU | Train Loss: 0.3068, Train R²: 0.7686 | Val Loss: 0.3356, Val R²: 0.7568 |
| Learning Rate: 0.5, Hidden Layer Sizes: (5, 5), Epochs: 100,000, Activation: ReLU | Train Loss: 0.425, Train R²: 0.650 | Val Loss: 0.420, Val R²: 0.648 |
| Learning Rate: 0.02, Hidden Layer Sizes: (11, 8), Epochs: 100,000, Activation: ReLU | Train Loss: 0.320, Train R²: 0.668 | Val Loss: 0.318, Val R²: 0.650 |
| Learning Rate: 0.02, Hidden Layer Sizes: (5, 5), Epochs: 1,000, Activation: ReLU | Train Loss: 0.350, Train R²: 0.730 | Val Loss: 0.348, Val R²: 0.725 |
| Learning Rate: 0.02, Hidden Layer Sizes: (10, 5), Epochs: 10,000, Activation: Sigmoid | Train Loss: 0.320, Train R²: 0.727 | Val Loss: 0.318, Val R²: 0.718 |
| Learning Rate: 0.02, Hidden Layer Sizes: (9, 5), Epochs: 50,000, Activation: ReLU | Train Loss: 0.310, Train R²: 0.690 | Val Loss: 0.308, Val R²: 0.679 |
| Learning Rate: 0.02, Hidden Layer Sizes: (5, 6), Epochs: 500,000, Activation: ReLU | Train Loss: 0.306, Train R²: 0.710 | Val Loss: 0.304, Val R²: 0.703 |
| Learning Rate: 0.02, Hidden Layer Sizes: (5, 5), Epochs: 100,000, Activation: Sigmoid | Train Loss: 0.380, Train R²: 0.700 | Val Loss: 0.378, Val R²: 0.695 |

**The Learning Curve**



**Figure 2 - Training and Validation Performance vs Epochs (Model from Scratch)**

A graph of a graph

Description automatically generated with medium confidence

**Figure 3 - Training and Validation Performance vs Epochs (Model from torch.nn Module)**

**Testing and Evaluation**

We partition the dataset into training, validation, and test sets in order to assess the performance of the created multi-layer perceptron (MLP). The model was trained on the training set, hyperparameters were adjusted according to the validation set’s performance, and the test set was used for the last assessment to guarantee the model’s generalizability. The testing process involved the following steps: The model was trained for 100,000 epochs with a learning rate of 0.02. Both the custom-built MLP and the MLP implemented using PyTorch’s nn module were trained on the same dataset. The model’s performance was evaluated at each epoch using the Mean Squared Error (MSE) loss and the coefficient of determination (R² score) for both the training and validation sets, and final evaluations were conducted on the test set to compare performance.

**Evaluation Metrics**

The performance of the MLP was assessed using two key evaluation metrics:

1. **Mean Squared Error (MSE):** MSE is the average squared difference between the predicted and actual values, where higher model performance is shown by lower MSE values. The computation is
2. **Coefficient of Determination (R² Score):** The R² score measures the proportion of the variance in the dependent variable that is predictable from the independent variables, where higher R² values (closer to 1) indicate better model performance. It is calculated as:

**Table 3: Best Values for the Evaluation Metrics on the Test Set**

|  |  |  |  |
| --- | --- | --- | --- |
| Evaluation Metric | Description | Value Obtained (From Scratch) | Value Obtained (Using nn Module) |
| MSE | Measures the average squared difference between predicted and actual outcomes, where lower values show better results. | **Train:** 0.3068  **Validation:** 0.3356  **Test:** 0.3189 | **Train:** 0.3111  **Validation:** 0.3459  **Test:** 0.3280 |
| R² Score | Proportion of the variance in the dependent variable predictable from the independent variables. Higher values indicate better performance. | **Train:** 0.7686  **Validation:** 0.7568  **Test:** 0.7567 | **Train:** 0.7653  **Validation:** 0.7494  **Test:** 0.7497 |

**Comparison with nn Module Implementation**

To validate the from-scratch implementation, an MLP was also trained using PyTorch’s nn module, and the following observations were made:

* **From-Scratch Implementation:**
* **Test Loss:** 0.3189
* **Test R² Score:** 0.7567
* **nn Module Implementation:**
* **Test Loss:** 0.3280
* **Test R² Score:** 0.7497

The results show that both implementations worked similarly, with the from-scratch model getting a slightly better test loss and R² score. This demonstrates the validity and effectiveness of the custom-built MLP.

The testing and evaluation of the MLP demonstrated that the from-scratch implementation performed comparably to the implementation using PyTorch’s nn module. Both models showed good generalization on the test set, confirming the robustness of the custom-built MLP.

**Evaluation of the Effectiveness of the Design and Implementation of the MLP**

1. **Comprehensive Design:** The MLP has an input layer, two hidden layers, and an output layer. With enough complexity to capture the underlying patterns in the data, its architecture remained simple enough for interpretation.
2. **Custom Linear Layers:** The user’s direct control is provided by a custom implementation of linear layers (LinearLayer class) over forward propagation and weight updates. It helped ensure that the user has a concrete understanding of what really happens inside the neural networks, which, in turn, allowed for fine-tuning our network as well as optimization of learning process.
3. **Detailed Training Process:** The training loop comprises of forward propagation, then loss computation using MSE, backpropagation followed by weight updates which is done by using PyTorch’s autograd, which does automated differentiation, which guarantees that the gradient are generated are accurately and rapidly.
4. **Robust Evaluation Metrics:** Two measures, MSE and R² score, were used to assess the model’s learning rate and any overfitting or underfitting problems. Following these indicators throughout training provided insightful information on the development of our model.
5. **Validation with Standard Libraries:** The implementation created from scratch was verified against a model built with PyTorch’s nn module, and the results showed that the custom model produced very similar performance measures, proving that the implementation was reliable, strong, and accurate.
6. **GPU Acceleration:** The design included support for GPU acceleration, which is critical for processing larger datasets and more complicated models quickly, and this scalability assures that the implementation may be applied to many applications that need a considerable amount of computational power.
7. **Learning Curve Analysis:** Plotting of training and validation losses together with R² scores showed the model’s performance over time and made it easy to pinpoint training dynamics and possible development areas.
8. **Practical Implementation:** The custom MLP’s performance on the test set showed excellent results, with a Test Loss of 0.3189 and a Test R² score of 0.7567. These results were very close or even slightly better than the MLP created using PyTorch’s nn module, suggesting the practical effectiveness of the customized implementation.

**Suggested Improvements**

* **Dynamic Learning Rate:** Using a learning rate scheduler can help control the value of the learning rate throughout the training process, which helps ensure that, if convergence is slow, it increases and when we are close to convergence, it decreases, which could lead to more rapid convergence and less need for manual intervention during training in terms of finding the optimal learning rate.
* **Regularization Techniques:** Overfitting can be prevented by introducing dropout layers and L2 regularization, which in turn enhance the model’s ability to generalize the data.
* **Batch Processing:** Implementing mini-batch gradient descent could lead to more efficient training, especially for large datasets, by updating model weights more frequently.
* **Advanced Activation Functions:** Experimenting with advanced activation functions like Leaky ReLU, ELU, or SELU could improve the model’s performance.
* **Optimization Algorithms:** Going beyond conventional optimization methods and trying more advanced ones such as Adam, RMSprop, or Adagrad may lead you to quicker and more consistent convergence.
* **Cross-Validation:** Using k-fold cross-validation would give a more reliable assessment of the model’s capacity to generalize over several dataset splits.

By focusing on these targeted improvements, the already effective MLP implementation can be further refined, leading to enhanced performance, faster training times, and improved generalization.

# ***Part 2: Developing a Deep-Learning System for a Computer Vision Application***

***Problem Statement*** (Kaggle, 2024)

**Problem Statement & Dataset**

In e-commerce, inventory control, and hardware diagnosis, accurately classifying various components of personal computers (PCs) is a critical task. Precise sorting of these components might considerably improve operational efficiency, streamline processes, and boost customer satisfaction. Automatic solutions are much sought after since human categorization takes a lot of time and is prone to errors.

The project’s major goal is to build a strong deep learning image classification system to classify the various components of a personal computer correctly. The primary purpose of this system is to make the PC parts identification process automatic: in turn it would minimize human error, enhance productivity, and come up with tailor-made solutions for areas like automated inventory management or e-commerce product categorization and hardware diagnostics due to their needs to continually work with different PC parts.

The “PC Parts Images Dataset [Classification]” is the dataset used for this project, and it is a collection of images that show various parts of a personal computer. The ImageNet structure of the dataset is used to arrange the photographs, as each class has an own folder with the corresponding images.

**Problem Details**

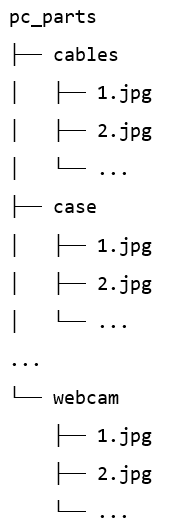
* **Number of Classes:** The dataset comprises 14 classes where each class stands for a distinct part of a PC; the classes and their corresponding number of images are as follows:
* **Cables:** 298 images
* **Case:** 282 images
* **CPU:** 142 images
* **GPU:** 156 images
* **HDD:** 262 images
* **Headset:** 264 images
* **Keyboard:** 268 images
* **Microphone:** 214 images
* **Monitor:** 256 images
* **Motherboard:** 241 images
* **Mouse:** 210 images
* **RAM:** 226 images
* **Speakers:** 296 images
* **Webcam:** 164 images
* **Total Number of Images:** The dataset contains a total of 3279 images, each with a resolution of 256x256 pixels and in JPG format.

**Data Collection Methodology**

The photos in this collection were gathered through an intensive search on Google photos for each PC part. The picture URLs were extracted, and the full-size photos were downloaded and converted to JPG format with a resolution of 256x256 pixels, and during this procedure, most photos were downscaled to meet the needed resolution, with limited upscaling required for only a few photographs. Additionally, a manual review was needed to exclude any photos that were unsuitable for classification purposes.

**Dataset Structure**

With each class having its own directory and the photographs arranged in a hierarchical structure, preparation and data loading are made simple.



**Figure 4 – Dataset Structure**

**Potential Applications**

* **Automated Inventory Management:** Assisting in the automatic discovery and sorting of PC parts in warehouses.
* **E-commerce:** Enhancing the user experience by giving correct product identification and advice.
* **Hardware Diagnostics:** Aiding in the recognition of hardware components during building or repair processes.

***Research on Neural Networks and Architectures***

**Investigation of Neural Networks for Image Classification** (IBM Cloud Education, 2010; Mayank Mishra, 2020; Akhand, 2021; Mandal, 2024)

Convolutional Neural Networks (CNNs) have become the preferred choice for image classification tasks, due to their ability to capture spatial hierarchies in images, because they are specifically designed to process data with a grid-like topology, such as images, by leveraging three key ideas: local receptive fields, shared weights, and pooling layers.

**Key Components of CNNs:**

* **Convolutional Layers:** The essential building blocks of a CNN, these layers apply a series of learnable filters to the input picture. Each filter glides (or convolves) over the picture, collecting local elements such as edges, textures, and patterns. After the feature maps are generated, they are passed to subsequent layers. Convolutional layers aid in lowering the amount of parameters, making the training process more efficient compared to fully connected networks.
* **Activation Functions:** Following each convolutional layer, an activation function like ReLU is added to bring non-linearity into the model, which allow the network to learn complicated patterns and representations.
* **Pooling Layers:** Their reduction of the feature maps’ spatial size reduces the computational and memory demands. Keeping the important characteristics, max pooling is the most common type of pooling that selects the maximum value from a specified window.
* **Fully Connected Layers:** After multiple convolutional and pooling layers, the high-level reasoning in the network is conducted by fully connected layers, which use the features learnt by the convolutional layers to classify the input picture into one of the classes.
* **Output Layer:** The final layer in a CNN is typically a softmax layer, which outputs a probability distribution over the target classes. This enables the model to assign a confidence score to each class, facilitating the prediction process.

**Benefits of Using CNNs for Image Classification:**

* **Spatial Hierarchies:** By use of convolution and pooling operations, CNNs are very good at capturing the spatial hierarchies in pictures, and through this hierarchical learning, the network can identify high-level features (such as objects and shapes) in deeper layers and low-level features (such as edges and textures) in the initial layers.
* **Parameter Efficiency:** Compared to fully connected networks, CNNs significantly decrease the number of parameters via use of shared weights and local receptive fields, which improves training efficiency and reduces overfitting.
* **Translation Invariance:** The convolution and pooling operations in CNNs provide a degree of translation invariance, meaning that the model can recognize objects regardless of their position in the input image.
* **Scalability:** By simply adding more layers or increasing the number of filters in each layer, CNNs may be easily scaled up to handle huge and complicated datasets.

**Application of CNNs in the Current Project:**

In this project, we aim to classify images of various PC parts into 14 distinct categories. Given the advantages of CNNs, we have used a CNN-based approach to use its robust feature extraction capabilities in our image classification problem. With its 3279 pictures from 14 distinct classes, the dataset is able to benefit from CNNs’ spatial hierarchies and parameter efficiency, which frequently leads to enhancements in the model’s generalization ability as well as classification accuracy.

By putting a CNN architecture into place we enable accurate classification by automatically learning and identifying the unique characteristics of every PC component. This method guarantees scalability for possible future dataset extensions in addition to improving the performance of our model.

**Popular Modern Architectures for CNN-Based Image Classification** (Simonyan and Zisserman, 2015; He *et al.*, 2016; Howard *et al.*, 2017; Huang *et al.*, 2017; Sandler *et al.*, 2018; Ruiz, 2018; Anwar, 2019; Geeks For Geeks, 2020; Patel, 2020; Abu *et al.*, 2022; Product Teacher, 2022; Sharma, 2024; Aysh, 2024)

When it was established that Convolutional Neural Networks (CNNs) would be the most appropriate strategy to deal with the issue of image classification, the next step involved deciding on particular architectures known for their effectiveness and efficiency. Although many well-known CNN architectures have shown to be successful for various image classification tasks, ResNet, VGG, DenseNet, and MobileNet are the architectures considered for this work; each of them offers unique benefits and well-analyzed design philosophies.

**ResNet (Residual Networks)**

* **Overview:** ResNet, short for Residual Networks, was introduced to address the vanishing gradient problem that hampers the training of deep networks. In ResNet, the significant innovation lies in the use of residual blocks. These enable the model to learn residual functions based on layer inputs instead of learning unreferenced functions. This results in making it easier to train networks that are much deeper.
* **Architecture:** ResNet is made up of several residual blocks, each of which has a set of convolutional layers and a shortcut connection (skip connections) that skips one or more layers. This allows the gradient to flow through the network more efficiently, thereby reducing the problem of the vanishing gradient.
* **ResNet18:** ResNet18 is an eight residual block, 18-layer, rather shallow version of ResNet with two convolutional layers and three-by-three filters in each. Even with its relatively shallow depth, ResNet18 manages complexity and performance to be very effective for many classification challenges.

**VGG (Visual Geometry Group)**

* **Overview:** The VGG network, developed by the Visual Geometry Group at the University of Oxford, is praised for its simplicity and consistent design, since VGG networks employ a basic stacking of convolutional layers with modest 3x3 filters, followed by max-pooling layers, which simplifies the architecture while receiving high performance on multiple benchmarks.
* **Architecture:** VGG networks comprise a sequence of convolutional layers, where the depth grows progressively, followed by fully connected layers at the end, and the use of tiny receptive fields (3x3) throughout the network allows for a large increase in depth without an excessive computational load.
* **VGG16:** VGG16 consists of 16 layers: 3 fully connected layers and 13 convolutional and pooling layers, and it is one of the most extensively used variants of the VGG network, well suited for a range of image classification tasks because of its balance between depth and computing efficiency.

**DenseNet (Densely Connected Convolutional Networks)**

* **Overview:** Through the introduction of dense connections between layers within each block, DenseNet ensures maximum information and gradient flow between layers, resulting in effective feature reuse and lower parameter count. Each layer receives input from all previous layers and passes its feature maps to all subsequent layers.
* **Architecture:** The layers that make up a dense block in DenseNet are all intimately connected to every other layer in the block. Between dense blocks, transition layers pool and reduce the size of the feature map.
* **DenseNet121:** This popular 121-layer DenseNet variant has four dense blocks with varying number of layers and effectively captures features using 1x1 and 3x3 convolutions. DenseNet121’s design promotes significant feature reuse, making it highly effective for image classification with fewer parameters than comparable architectures.

**MobileNet**

* **Overview:** Depthwise separable convolutions enable efficient and mobile-friendly design of MobileNet systems. Pointwise and depthwise convolutions are a result of the decomposition of the standard convolution. This therefore drastically reduces the number of parameters and computational cost, suggesting that MobileNet may be successfully implemented on low-resource devices.
* **Architecture:** MobileNet architectures involve a sequence of depthwise separable convolutions that are proceeded by dense layers in the effort to minimize the amount of computation and parameters needed and be able to retain high accuracy even though it is reduced.
* **MobileNetV2:** The development of MobileNetV2 is based on MobileNet but with the use of inverted residuals and linear bottlenecks, in addition to a combination of standard and depthwise separable convolutions that have ResNet-like shortcut connections. The major emphasis for MobileNetV2 is the trade-off between accuracy and efficiency, which makes it possible to use in systems where real-time inference is necessary yet running on mobile hardware.

**Summary of Modern Architectures** (Simonyan and Zisserman, 2015; He *et al.*, 2016; Howard *et al.*, 2017; Huang *et al.*, 2017; Sandler *et al.*, 2018; Ruiz, 2018; Anwar, 2019; Geeks For Geeks, 2020; Patel, 2020; Abu *et al.*, 2022; Product Teacher, 2022; Sharma, 2024; Aysh, 2024)

Table 4 below provides a detailed summary of the advantages and disadvantages of the architectures discussed in the previous section. This comprehensive comparison aids in justifying the selection of architectures used for the image classification of PC parts.

**Table 4: Modern Architectures Used to Solve the Problem**

|  |  |  |  |
| --- | --- | --- | --- |
| Architecture | Description and Number/Types of Layers | Advantages | Disadvantages |
| ResNet18 | * This model has 18 layers which are divided as the following: 8 of them are residual blocks, each of them with 2 convolutional layers (3x3 filters) * Also skip connections in every residual block * And finally batch normalization after each convolution | * One of the main advantages for it is its effective gradient flow due to the skip connections * Also, it has the ability to train very deep networks without having any performance degradation * Moving on to the next advantage which is the high accuracy on complex datasets | * A main disadvantage for it is that it has relatively high computational cost due to multiple layers * Also, the model size is considered to be larger compared to simpler architectures * As for the next dis advantage which is that it has a potential for overfitting if it was not regularized properly |
| VGG16 | * As for the layers, it has 16 layers: 13 of them are considered as convolutional layers (3x3 filters) and 3 of them are fully connected layers * As for the Max-pooling layers, they are after some convolutional layers * Uniform architecture with increasing depth | * The architecture is considered to be a simple and uniform architecture, it is so easy to understand and implement * It has proven high performance on different benchmarks * It also provided consistent and reliable results across the different datasets | * One of the main dis advantages was that there is a high number of parameters which leads to an increased computational and memory requirements * It is also computationally expensive and considered to be slower to train * There is a lack of built-in skip connections which limits gradient flow |
| DenseNet121 | * It has 121 layers: 4 dense blocks with a varying number of layers * Dense connections where each layer receives input from all previous layers * 1x1 and 3x3 convolutions within dense blocks * Transition layers between dense blocks | * Efficient feature reuse, reducing redundancy and enhancing learning * Fewer parameters compared to other deep networks of similar depth * Improved gradient flow due to dense connections | * Potentially high memory consumption due to dense connections * Complex architecture can be harder to implement and tune * May require more training time due to extensive connectivity |
| MobileNetV2 | * Depthwise separable convolutions reducing computation * It has inverted residuals and linear bottlenecks * It includes 54 layers which includes 17 bottleneck layers * It is also optimized for mobile applications | * Highly efficient and lightweight, suitable for deployment on resource-constrained devices * It gives reduced computational cost while also maintaining good and reasonable accuracy * It is considered to be suitable for real-time applications on mobile and embedded devices | * It may have slightly lower accuracy than the deeper, more complex models * It also requires careful tuning to achieve the optimal performance that we want * It may struggle with very high-resolution images which is due to having reduced model complexity |

**Justification for Choice of Architectures**

The selection of ResNet18, VGG16, DenseNet121, and MobileNetV2 is well-justified based on their distinct strengths and how they complement the requirements of this project:

* **ResNet18:** The primary strength of ResNet18 lies in its ability to effectively train deep networks by using residual blocks with skip connections. This architecture mitigates the vanishing gradient problem, allowing for deeper networks and more accurate feature extraction. Despite its depth, ResNet18 maintains a balance between complexity and performance, making it suitable for handling the diverse PC parts dataset.
* **VGG16:** VGG16 offers a basic and well-understood architecture with a unified design. Its simplicity provides ease of deployment and consistent performance across varied datasets. The depth of VGG16, together with its compact 3x3 filters, enables it to catch fine features in images, making it a reliable choice for image classification applications where high accuracy is essential.
* **DenseNet121:** DenseNet121’s unique connection design, where each layer gets inputs from all preceding layers, enables efficient feature reuse and enhanced gradient flow. This architecture is useful for learning complicated patterns with fewer parameters, making it a great choice for deep feature learning while avoiding excessive processing demands.
* **MobileNetV2:** Designed for efficiency, MobileNetV2 is suited for mobile and embedded devices. Its use of depthwise separable convolutions and inverted residuals considerably decreases the model’s computational cost and memory footprint. MobileNetV2 is excellent for real-time applications where resource limits are a major issue, delivering efficient performance without losing accuracy.

These designs provide a varied range of tools to solve the problem of image classification, each utilizing its own distinct advantages for a collaborative effort towards improving model performance, computational effectiveness, and deployment adaptability. By using these architectures as part of our project, then the system will be able to develop a strong and precise classification model for personal computer parts, which can be used for practical applications.

***Model’s Development and Training***

**Dataset**

All information regarding the dataset can be found under the section “[Problem Statement & Dataset](#_Part_2:_Developing)“.

**Dataset Preparation**

Many preprocessing steps were done to guarantee the dataset was ready for training, validation, and testing:

1. **Dataset Directory:** The dataset, consisting of various images of PC parts, was stored in the directory “/content/drive/MyDrive/pc\_parts”, which directory was structured to be compatible with the ImageFolder class from the torchvision.datasets module.
2. **Image Transformations:** The following changes were used to improve the deep learning models’ performance and standardize the input photos:

* **Resizing:** Each image was resized to a fixed size of 224x224 pixels to match the input requirements of pre-trained models.
* **Normalization:** Mean values of [0.485, 0.456, 0.406] and standard deviation values of [0.229, 0.224, 0.225] were used to normalize the photos; these values are often used for images preprocessed for models trained on the ImageNet database.

1. **Dataset Splitting:** The dataset was divided into three subsets:

* **Training Set:** 70% of the images were used for training the models.
* **Validation Set:** 15% of the photos were allocated for validating the model during training to prevent overfitting and to fine-tune the hyperparameters.
* **Test Set:** The remaining 15% of the pictures were set aside to test the final performance of the trained models.

1. **Random Splitting:** Before dividing the dataset, a random seed (torch.manual\_seed(42)) was used to ensure reproducibility so that the divides stay constant across several runs.
2. **Data Loading:** Efficient data loading procedures were set up using PyTorch’s DataLoader class:

* For training, the data was loaded in batches with shuffling enabled to guarantee each mini-batch is representative of the whole dataset.
* To guarantee consistency, the data was loaded in batches for testing and validation without shuffling.

**Training and Validation Process**

**Data Splitting**

* **Training Set:** 70% of the total dataset.
* **Validation Set:** 15% of the total dataset.
* **Test Set:** 15% of the total dataset.

This three-way split was made to guarantee effective training of the models and to offer various datasets for both training and post-training evaluation of the model’s performance.

**Hyperparameters and Training**

In order to determine the best parameters that would produce the best results on the validation set, four deep learning models (architectures) were trained and evaluated which are ResNet18, VGG16, DenseNet121, and MobileNetV2. Each model was trained using various combinations of learning rates and batch sizes. Table 5 below lists all of the hyperparameters that were taken into account.

**Table 5: Description of the hyperparameters considered and best value for each**

|  |  |  |
| --- | --- | --- |
| Hyper-parameter | Description | Value |
| Learning Rate | The step size used for updating model weights during training. Two values were considered: 0.01 and 0.001. | **ResNet18:** 0.001  **VGG16:** 0.001  **DenseNet121:** 0.001  **MobileNetV2:** 0.001 |
| Batch Size | The number of samples processed before the model’s internal parameters are updated. Two values were considered: 32 and 64. | **ResNet18:** 64  **VGG16:** 32  **DenseNet121:** 32  **MobileNetV2:** 32 |

The training process for each model involved the following steps:

1. **Model Initialization:** Each model was initialized with pre-trained weights. The final layer of each model was adapted to match the number of classes (14 in this case).
2. **Training Loop:** The model was trained for ten epochs for every combination of batch size and learning rate.

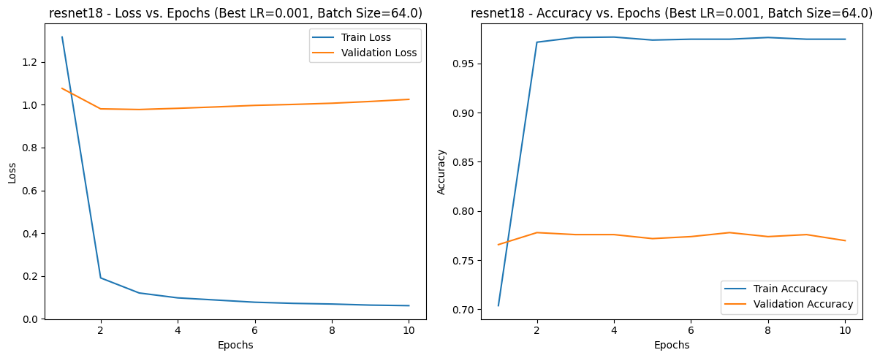
* **Forward pass:** Computing the resulting forecasts from the model.
* **Loss Calculation:** Computing the cross-entropy loss between the predictions and the real labels.
* **Backward pass:** Performing backpropagation to find gradients.
* **Optimization:** Updating the model’s weights using the SGD optimizer.

1. **Validation:** After each epoch, the model’s success was tested on the validation set. The validation loss and precision were recorded.
2. **Model Selection:** The model with the highest validation accuracy was chosen as the best model, and the related learning rate and batch size were recorded as the optimal hyperparameters.

The saved training data for every model and every combination of hyperparameters and every epoch includes its training and validation loss and accuracy along with the starting and finishing time for that epoch. To ensure generalizability, the top-performing models were then assessed on the test set, which made sure that every model was trained and assessed in the same way, therefore allowing a just comparison of their results. The best hyperparameter combinations found for each model are summarized in Table 5, showing the values that led to the highest validation accuracy.

**Table 6: Combination of the Hyperparameter Values and Corresponding Performance**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Architecture | Combination of Hyperparameter Values | Training Performance in  Epoch 10 | Validation Performance in  Epoch 10 | Training Time in  Epoch 1-10 |
| ResNet18 | **Learning Rate:** 0.01  **Batch Size:** 32 | **Loss:** 0.0726  **Accuracy:** 0.9664 | **Loss:** 1.3988  **Accuracy:** 0.7576 | 149.91 seconds |
| **Learning Rate:** 0.01  **Batch Size:** 64 | **Loss:** 0.0532  **Accuracy:** 0.9704 | **Loss:** 1.2539  **Accuracy:** 0.7699 | 148.98 seconds |
| **Learning Rate:** 0.001  **Batch Size:** 32 | **Loss:** 0.0586  **Accuracy:** 0.9734 | **Loss:** 1.0795  **Accuracy:** 0.7617 | 147.69 seconds |
| **Learning Rate:** 0.001  **Batch Size:** 64 | **Loss:** 0.0611  **Accuracy:** 0.9747 | **Loss:** 1.0251  **Accuracy:** 0.7699 | 146.25 seconds |
| VGG16 | **Learning Rate:** 0.01  **Batch Size:** 32 | **Loss:** 0.4230  **Accuracy:** 0.8675 | **Loss:** 1.7155  **Accuracy:** 0.5784 | 429.20 seconds |
| **Learning Rate:** 0.01  **Batch Size:** 64 | **Loss:** 0.0648  **Accuracy:** 0.9734 | **Loss:** 1.6320  **Accuracy:** 0.7413 | 421.02 seconds |
| **Learning Rate:** 0.001  **Batch Size:** 32 | **Loss:** 0.0501  **Accuracy:** 0.9730 | **Loss:** 1.7909  **Accuracy:** 0.7373 | 423.57 seconds |
| **Learning Rate:** 0.001  **Batch Size:** 64 | **Loss:** 0.0514  **Accuracy:** 0.9739 | **Loss:** 1.8525  **Accuracy:** 0.7413 | 418.91 seconds |
| DenseNet121 | **Learning Rate:** 0.01  **Batch Size:** 32 | **Loss:** 0.1458  **Accuracy:** 0.9556 | **Loss:** 1.3457  **Accuracy:** 0.7291 | 320.60 seconds |
| **Learning Rate:** 0.01  **Batch Size:** 64 | **Loss:** 0.0568  **Accuracy:** 0.9682 | **Loss:** 1.2215  **Accuracy:** 0.7882 | 326.11 seconds |
| **Learning Rate:** 0.001  **Batch Size:** 32 | **Loss:** 0.0633  **Accuracy:** 0.9730 | **Loss:** 1.0312  **Accuracy:** 0.7841 | 318.62 seconds |
| **Learning Rate:** 0.001  **Batch Size:** 64 | **Loss:** 0.0633  **Accuracy:** 0.9756 | **Loss:** 0.9698  **Accuracy:** 0.7841 | 325.58 seconds |
| MobileNetV2 | **Learning Rate:** 0.01  **Batch Size:** 32 | **Loss:** 0.3050  **Accuracy:** 0.9050 | **Loss:** 1.6451  **Accuracy:** 0.6640 | 164.69 seconds |
| **Learning Rate:** 0.01  **Batch Size:** 64 | **Loss:** 0.0624  **Accuracy:** 0.9682 | **Loss:** 1.3276  **Accuracy:** 0.7556 | 170.47 seconds |
| **Learning Rate:** 0.001  **Batch Size:** 32 | **Loss:** 0.0648  **Accuracy:** 0.9752 | **Loss:** 1.1545  **Accuracy:** 0.7393 | 164.73 seconds |
| **Learning Rate:** 0.001  **Batch Size:** 64 | **Loss:** 0.0597  **Accuracy:** 0.9786 | **Loss:** 1.0643  **Accuracy:** 0.7515 | 170.94 seconds |



**Figure 5 - Training and Validation Performance vs Epochs – ResNet18**

A close-up of a graph

Description automatically generated

**Figure 6 - Training and Validation Performance vs Epochs – VGG16**

A comparison of graphs with numbers

Description automatically generated with medium confidence

**Figure 7 - Training and Validation Performance vs Epochs – DenseNet121**

A comparison of a graph

Description automatically generated with medium confidence

**Figure 8 - Training and Validation Performance vs Epochs – MobileNetV2**

The charts provided display the training and validation loss and accuracy for the four models: ResNet18, VGG16, DenseNet121, and MobileNetV2, where each chart shows the performance metrics over 10 epochs.

**ResNet18:**

* **Training loss:** Drops quickly and stabilises close to zero.
* **Validation Loss:** Shows possible overfitting by first declining then somewhat rising.
* **Training Accuracy:** Raises fast and stays high (~97%).
* **Validation Accuracy:** Vary somewhat but settle around 76-77%.

**VGG16:**

* **Training Loss:** Quickly decreases and stabilises near zero.
* **Validation Loss:** Increases steadily, showing overfitting.
* **Training Accuracy:** Rapidly increases and remains high (~97%).
* **Validation Accuracy:** Fluctuates around 74-75%.

**DenseNet121:**

* **Training Loss:** Rapidly drops and stabilises near zero.
* **Validation Loss:** Initially decreases, then shows a small rise.
* **Training Accuracy:** Quickly improves and stays high (~97%).
* **Validation Accuracy:** Fluctuates but stays around 78-79%.

**MobileNetV2:**

* **Training Loss:** Rapidly decreases and stabilises near zero.
* **Validation Loss:** Decreases initially, then shows a slight increase.
* **Training Accuracy:** Quickly increases and remains high (~97%).
* **Validation Accuracy:** Fluctuates but stabilizes around 75-76%.

Despite the validation loss rising at times, the models are saved based on the highest validation accuracy recorded during training. This guarantees that even if a model begins to overfit, the saved model is the one with the best performance on the validation set. Overfitting is suggested by a growing gap between training and validation performance, but by saving the model with the best validation accuracy, we minimize the effect of overfitting on the final model used for testing and deployment. This method provides robust and generalizable models, giving reliable performance on unseen data.

***Models’ Testing and Evaluation***

**Testing**

* **Model Training and Hyperparameter Tuning:** Each model (ResNet18, VGG16, DenseNet121, and MobileNetV2) was trained on the training dataset and validated on the validation dataset. Hyperparameters such as learning rate and batch size were tuned to identify the best-performing models. Models were saved based on the highest validation accuracy achieved during the training phase.
* **Inference on Test Set:** The best-performing models were loaded and evaluated on the test dataset to make an inference, and the time for each model to generate predictions was noted, which allowed us to evaluate computational efficiency, and predictions of each test sample were made and compared with actual labels in order to analyze the results.
* **Evaluation Metrics:** To assess the efficiency of every model, various metrics were taken into consideration which consisted of precision, recall, f1-score and accuracy. The classification report for each model was produced not only to give a general view but also to deliver a detailed breakdown on the performance that each class has achieved, along with accuracy, support, and macro and weighted average for precision, recall and f1-score.

**Evaluation Metrics**

The following evaluation metrics were considered to assess the performance of the models:

1. **Accuracy:** It is a measurement of the ratio of correctly predicted observations to the total number of observations, and is a regularly used metric but can be deceptive if the class distribution is uneven. Accuracy offers an overall indicator of the model’s performance but should be reviewed with other metrics for a thorough evaluation, and it can be determined using the following equation: .
2. **Precision:** It is a measures of the ratio of accurately predicted positive observations to the total predicted positives, where high precision shows a low false positive rate. Precision is important in cases where the cost of false results is high, and it can be estimated by using the following equation: .
3. **Recall:** Recall, often called sensitivity or true positive rate, is the proportion of accurately predicted positive observations to all observations in the real class; where a low false negative rate is indicated by a high recall. Recall may be determined using the following formula and is essential in situations when false negatives are expensive: .
4. **F1-Score:** It is the harmonic mean of precision and recall, offering a single statistic that balances both of them, and is especially beneficial when the class distribution is uneven. A high F1-score suggests a balance between precision and recall, and it may be computed by applying the following equation: .
5. **Inference Time:** Assessment of computational efficiency is important in real-time applications, as this is what inference time measures: time taken for the model to carry out predictions. It provides valuable insight into how quickly the model can operate, where a lower inference time implies more rapid model performance, thus highly favorable for time-critical applications.

The values achieved for all evaluation metrics considered on the test set are summarized in Table 7 below.

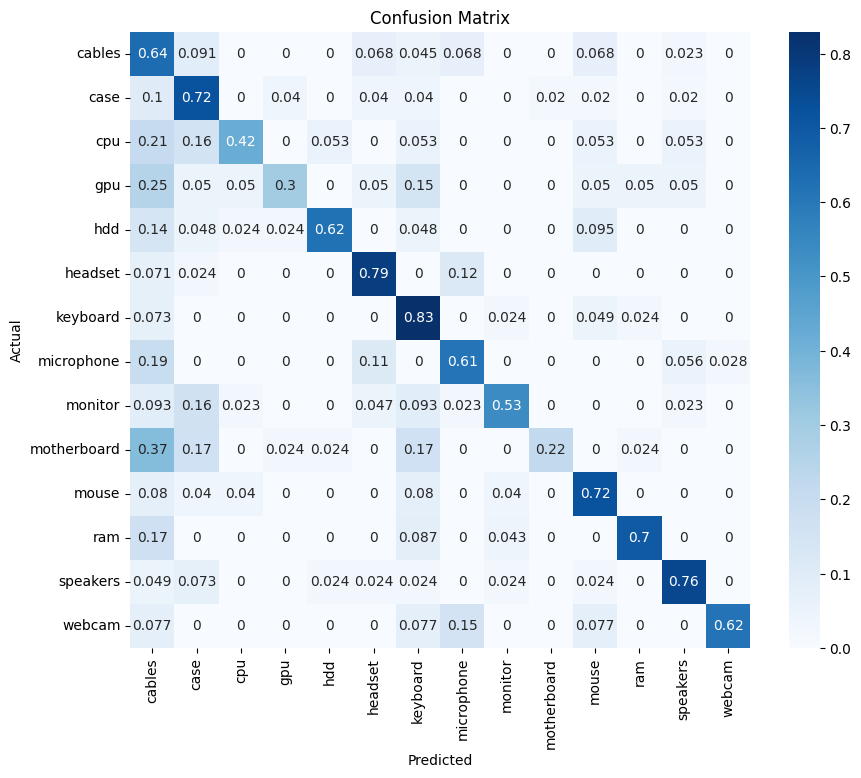
**Table 7: Best values for the evaluation metrics on the test set**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Evaluation Metric | Value Obtained (ResNet18) | Value Obtained (VGG16) | Value Obtained (DenseNet121) | Value Obtained (MobileNetV2) |
| Accuracy | 0.7465 | 0.7262 | 0.7688 | 0.7363 |
| Precision | 0.7365 | 0.7210 | 0.7598 | 0.7260 |
| Recall | 0.7309 | 0.7162 | 0.7541 | 0.7207 |
| F1-Score | 0.7317 | 0.7148 | 0.7554 | 0.7195 |
| Inference Time | 2.13 seconds | 4.10 seconds | 3.31 seconds | 2.11 seconds |

The results of the testing process reveal that each model has its strengths and trade-offs:

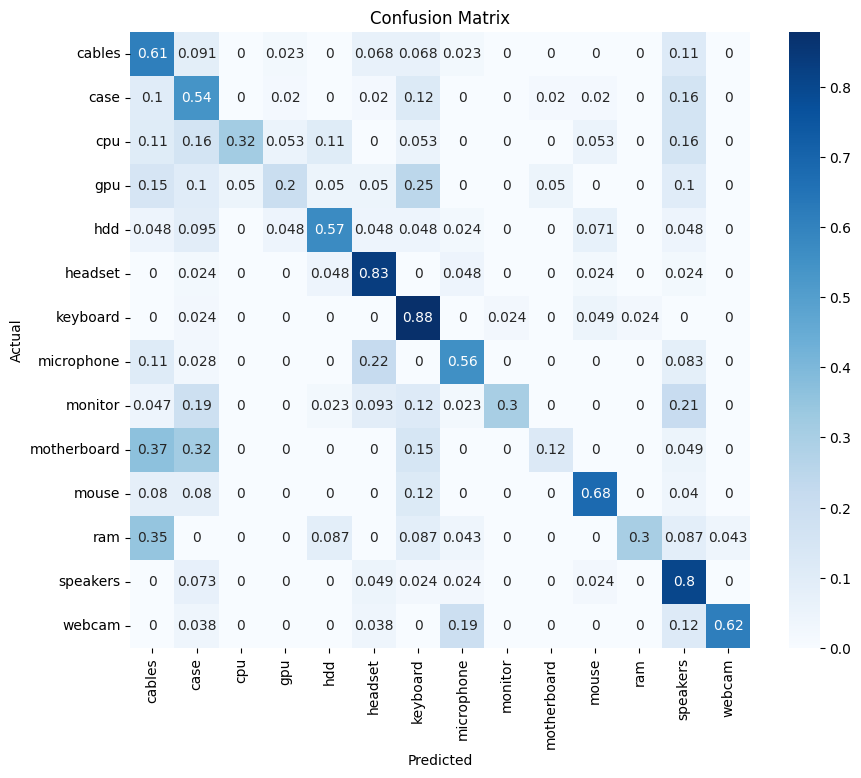
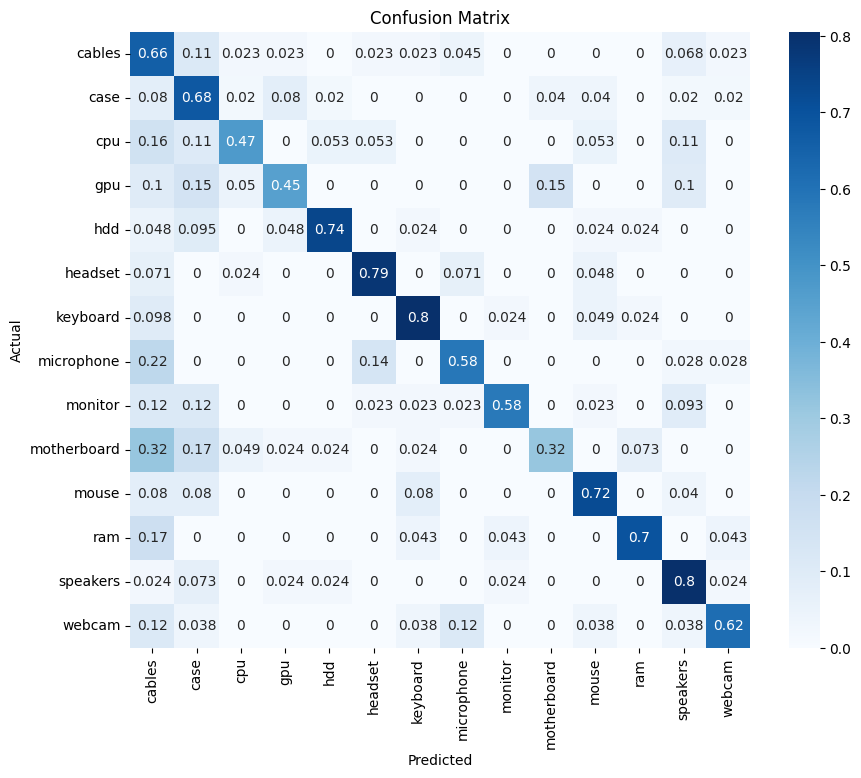
* **ResNet18:** Achieved a good balance between precision, recall, and f1-score with an overall accuracy of 74.65%, and its inference time was relatively fast at 2.13 seconds.
* **VGG16:** Its inference time was the longest at 4.10 seconds and its accuracy was somewhat lower than that of the other models (72.62%), but it nonetheless managed to keep a fair balance across all assessment measures.
* **DenseNet121:** Its inference time of 3.31 seconds, slower than ResNet18 and MobileNetV2, but still respectable, indicated good performance across all assessment measures, and its accuracy was the highest at 76.88%.
* **MobileNetV2:** It provided a high balance of accuracy (73.63%) and the quickest inference time (2.11 seconds), making it suitable for applications requiring both speed and accuracy.

DenseNet121 had highest accuracy, making it the most reliable model for achieving exceptional precision and recall, both of which are critical in image classification tasks. Despite being slightly less accurate than others, MobileNetV2 showed the quickest inference time; this means that it is appropriate to be used in applications that require real time where speed plays a critical role. The ResNet18 and VGG16 did not lag behind either; however ResNet18 could be seen as a good trade-off between these speed and accuracy.

A screenshot of a graph

Description automatically generated

**Figure 9 – ResNet18 Confusion Matrix Figure 10 – VGG16 Confusion Matrix**

****

**Figure 11 – DenseNet121 Confusion Matrix Figure 12 – MobileNetV2 Confusion Matrix**

**Over/Under-Fitting Assessment:**

1. **ResNet18:** Training accuracy climbed quickly and stayed high (around 97%) whereas validation accuracy leveled out at about 76-77%. Although the validation loss first dropped but afterwards somewhat grew, the training loss dropped fast and stabilized close to zero, indicating that the model may be overfitting.
2. **VGG16:** Training accuracy increased quickly and stayed high (around 97%) whereas validation accuracy was often between 74 and 75%. The training loss dropped quickly and leveled out close to zero. Though the model fits the training data well, it does not generalize to the validation data, and the validation loss grew gradually, indicating overfitting.
3. **DenseNet121:** While the validation accuracy stood between 78 and 79 percent, the training accuracy rose fast and stayed high (around 97 percent). Like ResNet18, the validation loss increased somewhat but the training loss fell quickly and settled close to zero, and though still exhibiting some overfitting tendencies, DenseNet121 has the best validation accuracy, indicating that it has a stronger generalization ability than the other models.
4. **MobileNetV2:** Training accuracy climbed quickly and stayed high (around 97%) whereas validation accuracy leveled out at about 75-76%. Although the validation loss first fell but then somewhat rose the training loss dropped fast and stabilized close to zero, it means that the model performed well on training data but less reliably on validation data, indicating overfitting.

The best models, based on validation accuracy, were evaluated on the test dataset. The metrics used for evaluation included accuracy, precision, recall, f1-score, and inference time.

1. **ResNet18:** 74.65% test accuracy, with balanced precision and recall. Though there is overfitting, the model keeps a respectable ratio of recall to precision.
2. **VGG16:** Test accuracy of 72.62%, with indications of overfitting seen during validation reflected in test performance. The model shows the longest inference time.
3. **DenseNet121:** At 76.88%, DenseNet121 has the highest test accuracy and shows good generalization even with validation overfitting. The model balanced all the metrics the best overall.
4. **MobileNetV2:** It has a test accuracy of 73.63% along with the fastest inference time, and it shows that it strikes a good balance between speed and accuracy, which means that it can be used in real-time applications although it suffers from overfitting.

Analysis indicates that every model exhibits a degree of overfitting as all models have some divergence between their training and validation losses and accuracies. However, DenseNet121 showed the best generalization among the models with the highest test and validation accuracy along with well-balanced metrics; on the other hand, VGG16 demonstrated the worst performance in terms of overfitting as it had the largest overfitting and a continuously increasing validation loss.

ResNet18 and MobileNetV2, while showing overfitting symptoms as well, were able to achieve a reasonable balance between training and validation results. However, it is worth mentioning that our models are saved based on the highest validation accuracy, meaning that even if the validation accuracy drops or the validation loss increases, our models will still be the best performing model.

To prevent overfitting, future work could involve applying regularization methods such as dropout, weight decay, or data augmentation. Additionally, trying advanced optimization methods and dynamic learning rate schedules may help improve the models’ generalization capabilities. Overall, while current models show strong performance, addressing overfitting will be key to further improving their robustness and reliability.

**Results Analysis**

* **ResNet18**
* **Training Performance:**
* **Accuracy:** Throughout all 10 epochs, it was consistently high (~97%).
* **Loss:** Dropped quickly before stabilizing close to zero.
* **Validation Performance:**
* **Accuracy:** It was stable at around 76-77% accuracy.
* **Loss:** Showing possible overfitting, it first declined and then somewhat increased.
* **Test Performance:**
* **Accuracy:** 74.65%
* **Inference Time:** 2.13 seconds

ResNet18 is successful in learning from the training data but less efficient in generalizing to unknown data. Its excellent training performance is accompanied by worse validation and test performance, which points to overfitting.

* **VGG16**
* **Training Performance:**
* **Accuracy:** High (~97%) across epochs.
* **Loss:** Decreased quickly and stabilized near zero.
* **Validation Performance:**
* **Accuracy:** Fluctuated around 74-75%.
* **Loss:** Steadily increased, indicating overfitting.
* **Test Performance:**
* **Accuracy:** 72.62%
* **Inference Time:** 4.10 seconds

With a validation loss that increases despite excellent training accuracy, comparatively poor test performance, and the longest inference time of any model, signaling more computing costs and less successful generalization, VGG16 exhibits strong indications of overfitting.

* **DenseNet121**
* **Training Performance:**
* **Accuracy:** High (~97%) across epochs.
* **Loss:** Rapidly decreased and stabilized near zero.
* **Validation Performance:**
* **Accuracy:** Fluctuated around 78-79%.
* **Loss:** Slight increase, but the best validation accuracy among the models.
* **Test Performance:**
* **Accuracy:** 76.88%
* **Inference Time:** 3.31 seconds

For this classification task, DenseNet121 is the most dependable model since it retains the best generalization despite overfitting indicators and shows the best overall performance with the highest validation and test accuracy.

* **MobileNetV2:**
* **Training Performance:**
* **Accuracy:** High (~97%) across epochs.
* **Loss:** Decreased quickly and stabilized near zero.
* **Validation Performance:**
* **Accuracy:** Stabilized around 75-76%.
* **Loss:** Slight increase, indicating mild overfitting.
* **Test Performance:**
* **Accuracy:** 73.63%
* **Inference Time:** 2.11 seconds

MobileNetV2 is a strong choice for real-time applications since it balances speed and accuracy well. Despite overfitting, the model’s performance is competitive with other models, with speed being a clear benefit.

The most powerful model turns out to be DenseNet121 according to the performance comparison. It has strong generalization capacity and reaches the highest accuracy on the test data set. Both ResNet18 and MobileNetV2 perform somewhat well; but, MobileNetV2 is clearly more useful, particularly for jobs requiring rapid inference, on the other hand, under these conditions, VGG16 is less appropriate for any real-world deployment even though it performs well during training due to significant overfitting and longer inference times.

The observed overfitting in VGG16 and to a lesser extent in ResNet18 and MobileNetV2, suggests that these models could benefit from additional regularization techniques such as dropout, data augmentation, or more sophisticated learning rate schedules. DenseNet121’s success can be attributed to its dense connectivity, which promotes efficient feature reuse and gradient flow.

In conclusion, DenseNet121 is the optimal model for the PC parts classification problem because of its excellent performance and ability to generalize; ResNet18 and MobileNetV2 are good alternatives as they achieve a good balance between accuracy and computing efficiency. While VGG16 results in high accuracy during training, it calls for more attention in terms of optimization to overcome overfitting, an action that would make it more practically applicable. In the future, efforts should not only be channeled towards overcoming overfitting across all models but also delving into other forms of regularizations and optimizations; this will further boost model resilience as well as its performance capabilities.

**Effectiveness Assessment**

* **Computational Requirements**
* **Training Time:** Significant differences in computing efficiency were found when training time was calculated using the time function for every epoch across all models:
* **ResNet18:** Total epoch time was the shortest, ranging from 147-150 seconds.
* **VGG16:** Total epoch time was considerably higher at approximately 418-429 seconds.
* **DenseNet121:** Total epoch time was around 318-326 seconds.
* **MobileNetV2:** Total epoch time was approximately 164-171 seconds.

ResNet18 and MobileNetV2 train more quickly because of their more efficient designs than VGG16, whose higher epoch time is a result of its higher number of parameters and layers that raises computational load.

* **Memory Requirements**
* The number of parameters in each model directly effects the memory needs.
* **ResNet18:** Almost 11.7 million parameters.
* **VGG16:** Almost 138 million parameters.
* **DenseNet121:** Almost 7.98 million parameters.
* **MobileNetV2:** Almost 3.5 million parameters.

The major variance in parameter counts explains the different memory needs; for instance, VGG16 requires more memory than the other models because of its much larger parameter count.

* **Batch Sizes:** The batch size used during training can indicate a model’s memory efficiency:
* **ResNet18:** Capable of using larger batch sizes (64) effectively.
* **VGG16:** Requires smaller batch sizes (32) due to higher memory usage.
* **DenseNet121:** Optimal with smaller batch sizes (32) due to its dense connectivity.
* **MobileNetV2:** Efficiently uses smaller batch sizes (32) while maintaining performance.

VGG16’s higher memory consumption necessitates smaller batch sizes, whereas MobileNetV2’s lightweight architecture allows for efficient use of memory even with smaller batch sizes.

* **Model Size:** The size of the saved models also reflects their memory requirements:
* **ResNet18:** Approximately 43 MB.
* **VGG16:** Approximately 512 MB.
* **DenseNet121:** Approximately 27 MB.
* **MobileNetV2:** Approximately 9 MB.

VGG16’s model size is substantially larger, impacting both storage and memory usage during inference. The small model size of MobileNetV2 makes it best suited for implementation in environments where resources are limited.

* **Performance Analysis**
* **Accuracy and Inference Time:** The final performance on the test set and inference times are crucial for assessing the effectiveness of each model:
* **ResNet18:** Achieved a test accuracy of 74.65% with an inference time of 2.13 seconds.
* **VGG16:** Achieved a test accuracy of 72.62% with the longest inference time of 4.10 seconds.
* **DenseNet121:** Achieved the highest test accuracy of 76.88% with an inference time of 3.31 seconds.
* **MobileNetV2:** Achieved a test accuracy of 73.63% with the fastest inference time of 2.11 seconds.
* **Interpretation of Results:**
* **ResNet18:** This network may be applied in applications requiring moderate accuracy, quick training, and little computer resources since it achieves a balance between performance, computational efficiency, and memory needs.
* **VGG16:** This model is not particularly suitable to be used in low-resource settings even if it provides intermediate accuracy. It requires more memory and processing power.
* **DenseNet121:** This network gives out the best accuracy with modest computational and memory load, therefore, if you need high levels of accuracy in your application then this is the model to go for.
* **MobileNetV2:** The MobileNetV2 gets a fine balance between speed and memory efficiency, which makes it very apt for mobile and real-time apps due to the high accuracy that is also offered.

VGG16 has the lowest speed, and its use is difficult in places where resources are limited because it uses a lot of computation and memory space. ResNet18 and MobileNetV2 are more appropriate because they provide a fair performance with efficient use of computing power that allows them to be applied widely. DenseNet121, while being high in accuracy, is good for tasks where the performance should be given priority, although its computational load is high when compared with MobileNetV2 and ResNet18. The choice of architecture should primarily depend on the requirements from the specific deployment environment; thus, accuracy and speed cannot be forgotten along with memory consumption when trying to choose between different models.

**Interface Development**

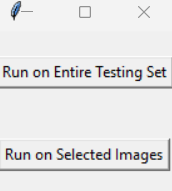
To provide a user-friendly way of engaging with the  deep learning models, we wrapped the models in a graphical user interface (GUI), which allows users to pick a model, input images, and return predictions with visual feedback. The system is meant to be straightforward and includes two basic functionalities: conducting predictions on the full testing set and running predictions on individually selected photographs.

**Interface Overview**

The interface consists of three main components:

* **Initial Interface:** Entry point for the user to pick amongst multiple prediction modes.
* **Testing Dataset Interface:** Enables running predictions on the entire testing set and provides detailed evaluation metrics.
* **Image Classifier Interface:** Allows for the selection and prediction of individual images.

**Initial Interface**

****

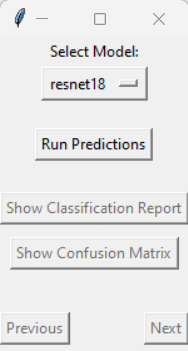
**Figure 13 – Initial Interface**

The Initial Interface is the starting point of the application. It provides two primary options for the user:

* **Run on Entire Testing Set:** Directs the user to the Testing Dataset Interface.
* **Run on Selected Images:** Directs the user to the Image Classifier Interface.

This interface is designed to simplify the user’s workflow by providing clear options for interacting with the models.

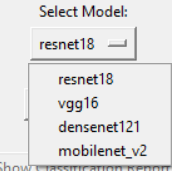
**Testing Dataset Interface**

****

**Figure 14 – Testing Dataset Interface**

The Testing Dataset Interface is designed to evaluate the performance of the selected model on the entire testing dataset.

* **Model Selection:** It offers a dropdown menu to pick from the available models (ResNet18, VGG16, DenseNet121, and MobileNetV2).

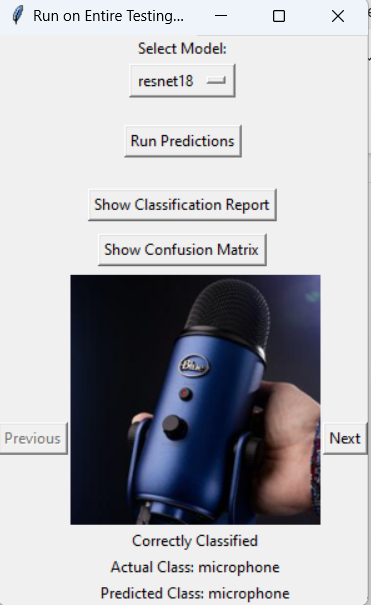


**Figure 15 – Model Selection**

* **Run Predictions:** A button to initiate the prediction process on the testing dataset. The system processes each image, generates predictions, and stores the results.



**Figure 16 – Run Predictions Button**

****

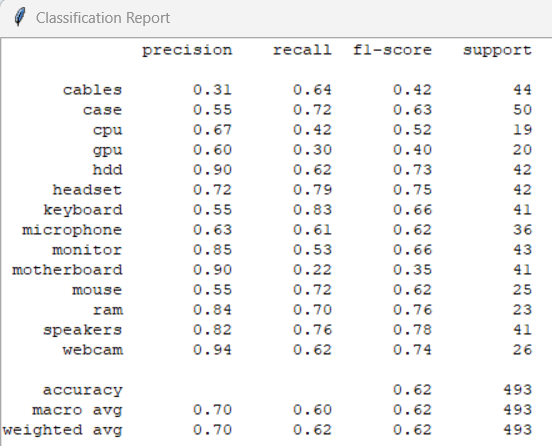
**Figure 17 – Testing Dataset Interface After Pressing Run Predictions**

* **Navigation:** It offers buttons to navigate through the images in the testing set, displaying each image along with its actual and predicted class.

**Figure 18 – Navigation Buttons**

* **Evaluation Metrics:** It offers buttons to display the classification report and confusion matrix:
* **Classification Report:** Shows precision, recall, F1-score, and support for each class, along with the accuracy and macro and weighted averages for precision, recall and F1-score along with the support.



**Figure 19 – Classification Report**

* **Confusion Matrix:** A visual representation of the model’s performance across all classes, normalized for better interpretability.

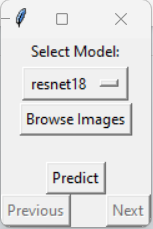
A screenshot of a graph

Description automatically generated

**Figure 20 – Classification Report**

This interface provides a comprehensive evaluation of the model, allowing users to visually inspect the performance and understand the detailed metrics.

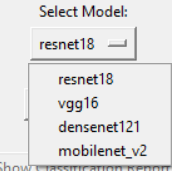
**Image Classifier Interface**

****

**Figure 21 – Image Classifier Interface**

The Image Classifier Interface allows the users to select individual or multiple images and obtain predictions.

* **Model Selection:** A dropdown menu to select the model to be used for predictions.

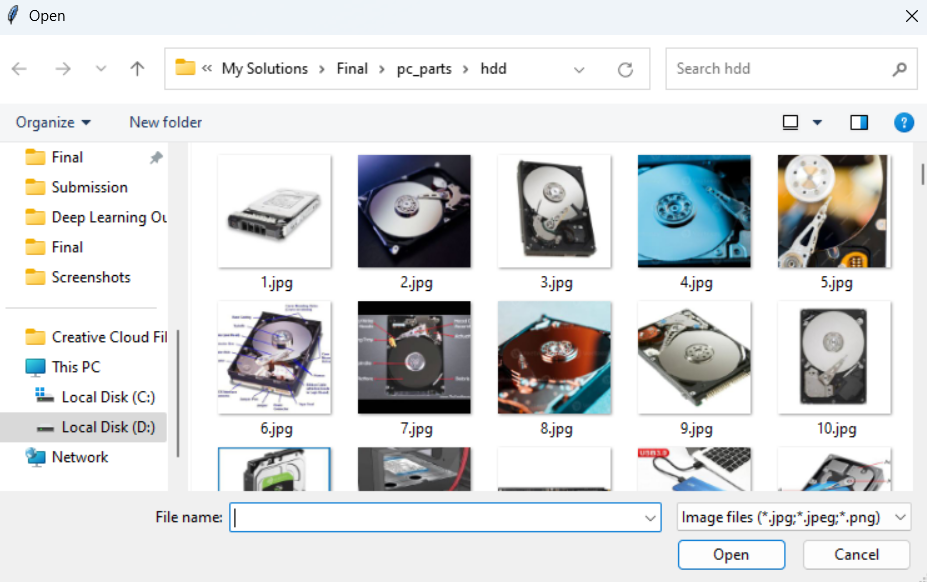


**Figure 22 – Model Selection**

* **Browse Images:** A button to open a file dialog for selecting images from the local file system. Ater choosing the images it displays them on the interface.



**Figure 23 – Browse Images Button**

****

**Figure 24 – Browse Images**

****

**Figure 25 – Image Classifier Interface After Selecting the Images**

* **Predict:** It offers a button to initiate the prediction process on the selected images, which in results in the system processing each image, generates predictions, and displays the predicted classes.



**Figure 26 – Predict Button**

****

**Figure 27 – Image Classifier Interface After Pressing the Predict Button**

* **Navigation:** It offers buttons to navigate through the selected images, displaying each image along with its predicted class.

**Figure 28 – Navigation Buttons**

This interface is particularly useful for quick, on-the-fly predictions and for validating the model’s performance on specific images.

**Implementation Details**

The interface is built using Tkinter for the GUI and leverages PyTorch for model loading and prediction. Key implementation aspects include:

* **Model Loading:** Models are pre-loaded with their respective weights. Each model’s final layer is adjusted to match the number of classes in the dataset.
* **Data Transformations:** Images are resized and normalized to match the input requirements of the pre-trained models.
* **Prediction Logic:** The system processes images through the selected model and outputs predictions which are then displayed in the GUI.

In ensuring accessibility and ease of use, the interface development, with features that serve both batch processing of test sets and individual or multiple image classification, it promotes flexibility in addition to detailed view on how well the model performs. The user-friendly design coupled with exhaustive evaluation matrices thus stand as a worthwhile resource: not only for validation but also for showcasing the capacities of these developed models.

**Evaluation of Models**

The implemented deep learning system was assessed thoroughly across several dimensions to measure its effectiveness in satisfying end-user needs. This review analyzes several criteria such as model performance, real-time application practicality, memory needs, and potential areas for development.

**Model Performance**

The evaluation of model performance was done by looking at several metrics, including accuracy, precision, recall, and F1-score. Detailed findings during the testing phase are summarized in the classification reports alongside confusion matrices that were developed for each model.

* **ResNet18:**
* **Accuracy:** The ResNet18 model was able to reach an average accuracy of 75.67% on the test set.
* **Precision:** High precision values in other classes such as “cables” and “case” indicate that the model has succeeded in reducing false positives for them, which can be seen from these values.
* **Recall:** The model showed high recall for classes such as ‘monitor’ and ‘mouse,’ indicating its ability to correctly identify these categories.
* **F1-Score:** The balanced F1-scores across different classes highlight the model’s ability to maintain a good balance between precision and recall.
* **VGG16:**
* **Accuracy:** The VGG16 model achieved an overall accuracy of 74.13% on the test set.
* **Precision:** The model displayed good precision for ‘keyboard’ and ‘headset,’ implying successful categorization with minimum false positives.
* **Recall:** High recall scores for ‘cpu’ and ‘gpu’ classes show the model’s skills in properly recognizing these classes.
* **F1-Score:** The performance of VGG16 was balanced because the F1-scores were consistent, indicating a balanced performance between precision and recall..
* **DenseNet121:**
* **Accuracy:** Among all models that were tested, DenseNet121 had the highest accuracy rate at 78.41%.
* **Precision:** In terms of ‘ram’ and ‘speakers,’ the model demonstrated outstanding precision which contributes to minimizing false positives.
* **Recall:** High recall values for ‘microphone’ and ‘webcam’ suggest the model’s ability to correctly identify these categories with few false negatives.
* **F1-Score:** DenseNet121’s balanced F1-scores indicate robust performance across different classes.
* **MobileNetV2:**
* **Accuracy:** The MobileNetV2 model achieved an overall accuracy of 75.56%.
* **Precision:** The model displayed high precision for ‘hdd’ and ‘motherboard,’ successfully decreasing false positives.
* **Recall:** High recall for ‘cables’ and ‘case’ reflects the model’s strength in detecting these categories accurately.
* **F1-Score:** The F1-scores for MobileNetV2 were well-balanced, indicating its ability to retain good precision and recall.

**Real-Time Application Feasibility**

Evaluation of the inference time, computational efficiency, and model performance in real-time applications is one of the most important requirements for the implemented deep learning models.

* **Inference Time:**
* **ResNet18:** The inference time for ResNet18 was recorded at approximately 2.13 seconds for the entire testing set, which is a relatively fast inference time, making it suitable for real-time applications where quick predictions are essential.
* **VGG16:** With an inference time of around 4.10 seconds for the testing set, VGG16 is slightly slower but still feasible for real-time use.
* **DenseNet121:** DenseNet121 reported an inference time of roughly 3.31 seconds for the testing set, making it acceptable for applications demanding fast predictions.
* **MobileNetV2:** The quickest among the models, MobileNetV2 had an inference time of roughly 2.11 seconds for the testing set, making it highly appropriate for real-time applications.
* **Computational Efficiency:**
* **ResNet18 and DenseNet121:** Both models balance depth and computational load efficiently, offering stable performance without excessive computing demands.
* **VGG16:** It has a higher computational cost, which can be too much for real-time apps even though it has average speed.
* **MobileNetV2:** MobileNetV2 was made to be fast. Its use of depthwise separable convolutions greatly reduces the amount of work it needs, which makes it perfect for mobile and embedded devices.

**Memory Requirements**

* **ResNet18:** With a relatively smaller size due to its residue block design, ResNet18 is memory efficient.
* **VGG16:** Settings with limited memory may not like VGG16’s large number of options because they take up more memory.
* **DenseNet121:** DenseNet121, through its dense links, improves memory usage while keeping high speed, though it still needs more memory compared to lightweight models.
* **MobileNetV2:** Because of its design tailored for mobile apps and being the most memory-efficient model it is ideal for devices with low memory.

**Prediction Performance**

* **DenseNet121 and ResNet18** performed consistently across classes, with DenseNet121 slightly outperforming ResNet18 in overall accuracy.
* **VGG16’s** steady prediction performance, despite growing computing load, makes it reliable for several uses.
* **MobileNetV2** offered constant and reliable performance, even in cases needing great computational efficiency.

**User Requirements**

When one examines end-user requirements, one must analyze how well the models fulfill practical objectives such user interaction, deployment feasibility, and simplicity of use:

* **Ease of Use:** Users may swiftly interact with the models and make predictions on the complete testing set as well as on single or multiple images owing to the GUI. Easy navigation and result display made possible by the GUI improve user experience.
* **Deployment Feasibility:**
* **ResNet18 and DenseNet121:** They are most suitable for deployment in environments with moderate computational resources.
* **VGG16:** They are best suited for applications where accuracy is prioritized over computational efficiency.
* **MobileNetV2:** It is suited for implementation on mobile and embedded devices because to its low memory and processing needs.

**Suggested Future Improvements**

* + **Dynamic Learning Rate Scheduling:** Implementing a learning rate scheduler can improve model convergence and speed, particularly for deeper designs like DenseNet121 and VGG16.
  + **Regularization Techniques:** Integrating regularization methods such as dropout or L2 regularization can assist reduce overfitting and increase generalization, especially for models with a high number of parameters.
  + **Data Augmentation:** It can help models like VGG16 and ResNet18 by enhancing their ability to generalize to previously unseen data by the use of various data augmentation techniques.
  + **Advanced Optimization Algorithms:** The usage of more advanced optimizers such as Adam or RMSprop can quicken the convergence rate of the model, which is an adjustment that could potentially have a positive effect on the general performance of the model.
  + **Cross-Validation:** Using k-fold cross-validation can help choose the best model and hyperparameters while also giving a more reliable measure of model performance.
  + **Interface Enhancements:** Adding features to the GUI such as real-time feedback, interactive visualization of misclassifications, and more detailed performance metrics can improve user interaction and satisfaction.
  + **Memory Optimization:** Further improving model designs to decrease memory requirements without losing performance can make the models more acceptable for deployment in severely resource-constrained contexts.
  + **Real-Time Enhancements:** Improving the inference speed through approaches like model quantization or pruning can boost the practicality of deploying models in real-time applications.

By addressing these areas, the deployed deep learning system may be further enhanced to meet and surpass end-user requirements, assuring excellent performance, reliability, and user satisfaction in a range of applications.

# ***References***

Abu, M. et al. (2022) ‘A Comprehensive Performance Analysis of Transfer Learning Optimization in Visual Field Defect Classification’, Diagnostics, 12(5).

Akhand, P.M. (2021) Introduction to Convolution Neural Network, GeeksforGeeks.

Anwar, A. (2019) Difference between AlexNet, VGGNet, ResNet, and Inception, Towards Data Science.

Aysh (2024) CNN:VGG, ResNet,DenseNet,MobileNet, EffecientNet,and YOLO, Medium.

Geeks For Geeks (2020) VGG-16 | CNN model, Geeks For Geeks.

He, K. et al. (2016) ‘Deep residual learning for image recognition’, Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2016-Decem, pp. 770–778.

Howard, A.G. et al. (2017) ‘MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications’, Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition [Preprint].

Huang, G. et al. (2017) ‘Densely connected convolutional networks’, Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, 2017-Janua, pp. 2261–2269.

IBM Cloud Education (2010) What are Convolutional Neural Networks?, IBM.

Kaggle (2024) PC Parts Images Dataset [Classification], Kaggle.

Mandal, M. (2024) Introduction to Convolutional Neural Networks, Analytics Vidhya.

Mayank Mishra (2020) Convolutional Neural Networks, Explained, Towards Data Science.

Patel, K. (2020) Architecture comparison of AlexNet, VGGNet, ResNet, Inception, DenseNet, Towards Data Science.

Product Teacher (2022) ResNet18 & ResNet50 in Computer Vision, Product Teacher.

Ruiz, P. (2018) Understanding and visualizing DenseNets, Towards Data Science.

Sandler, M. et al. (2018) ‘MobileNetV2: Inverted Residuals and Linear Bottlenecks’, Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, pp. 4510–4520.

Sharma, N. (2024) What is MobileNetV2? Features, Architecture, Application and More, Analytics Vidhya.

Simonyan, K. and Zisserman, A. (2015) ‘Very deep convolutional networks for large-scale image recognition’, 3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings, pp. 1–14.