Implementation and Analysis of ResNet-50 for Image Classification

A Deep Learning Project on Multi-Class Classification using Residual Neural Networks

---

Prepared by:

Sai Pavan Kumar Yedduri, Intekhab Alam

Date:

April 6, 2025

Abstract:

This project implements and analyzes a ResNet-50 convolutional neural network for multi-class image classification across 10 distinct object categories. The document covers data collection methodology, network architecture, training pipeline implementation, hyperparameter tuning, model evaluation, and result analysis. The implemented model achieves 55.29% accuracy on the test dataset through systematic optimization of learning parameters and training techniques.

Keywords:

Deep Learning, ResNet-50, Convolutional Neural Networks, Image Classification, Residual Learning, Bottleneck Architecture, Hyperparameter Tuning, Cross-Validation

Table of Contents

1. Description of Data:

The dataset comprises real-world images spanning 10 distinct object categories, including *bike, bottle, chair, cup, fork, knife, plant, shoe, spoon,* and *t-shirt*. Real-world images were captured to ensure a diverse and representative dataset supporting the principles of independent and identically distributed (i.i.d.) data for robust model training and evaluation in machine learning.

1.1 Data Collection Methodology:

To create a dataset that accurately represents real-world scenarios, images were captured from various perspectives, ensuring that some instances of the same category remain pairwise disjoint. This means that objects were photographed from different angles, orientations, and settings, preventing redundancy and ensuring the model generalizes well to unseen data.

Furthermore, no restrictions were imposed on the image dimensions (height and width), allowing for the natural variance in real-world image sizes. The images were taken across multiple resolutions and lighting conditions by different people at different locations, reinforcing the dataset's diversity and preventing biases that could arise from uniform environmental conditions. These measures were implemented to simulate real-world variance and ensure that the dataset is independent, meaning that one image does not influence another, and identically distributed, as each class is well-represented without an inherent bias toward specific conditions.

Under these specific conditions, an image database is created with a total of 3,804 images collected across the 10 categories. The distribution of images per class is visualized in Figure 1, demonstrating the dataset's near-uniform class distribution, which helps mitigate class imbalance issues in the classification task.

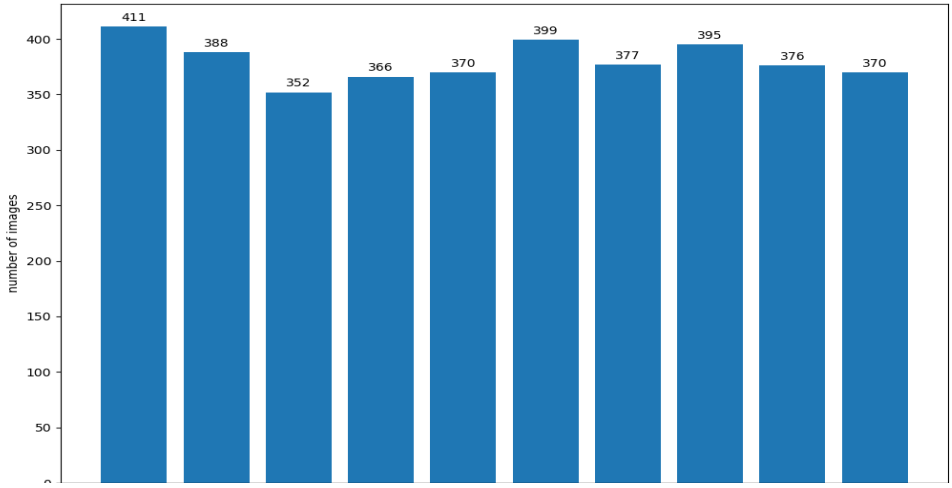

Figure 1:Visualization of distribution of number of images in each class, across 10 classes

1.2 Preparation of data:

The captured dataset was subsequently split into training and testing subsets, maintaining almost 80:20 ratio per class. This process involves shuffling the images within each class, randomly selecting almost 80% for training and 20% for testing, and copying them into separate directories. Since 80% of the number of images can't always be a number, it is rounded to the nearest possible number. Thereby training data is split by almost 80 percent in each class, may not be exactly 80% of the captured data. All the transformations are performed on the training data and only used for the training model. The remaining 20% test data is used to evaluate the model for predictions. The data is initially split into train and test data so that transformations can be applied on the training data and augment the training data. The class distribution of the augmented training data and test data can be observed in Figure2.

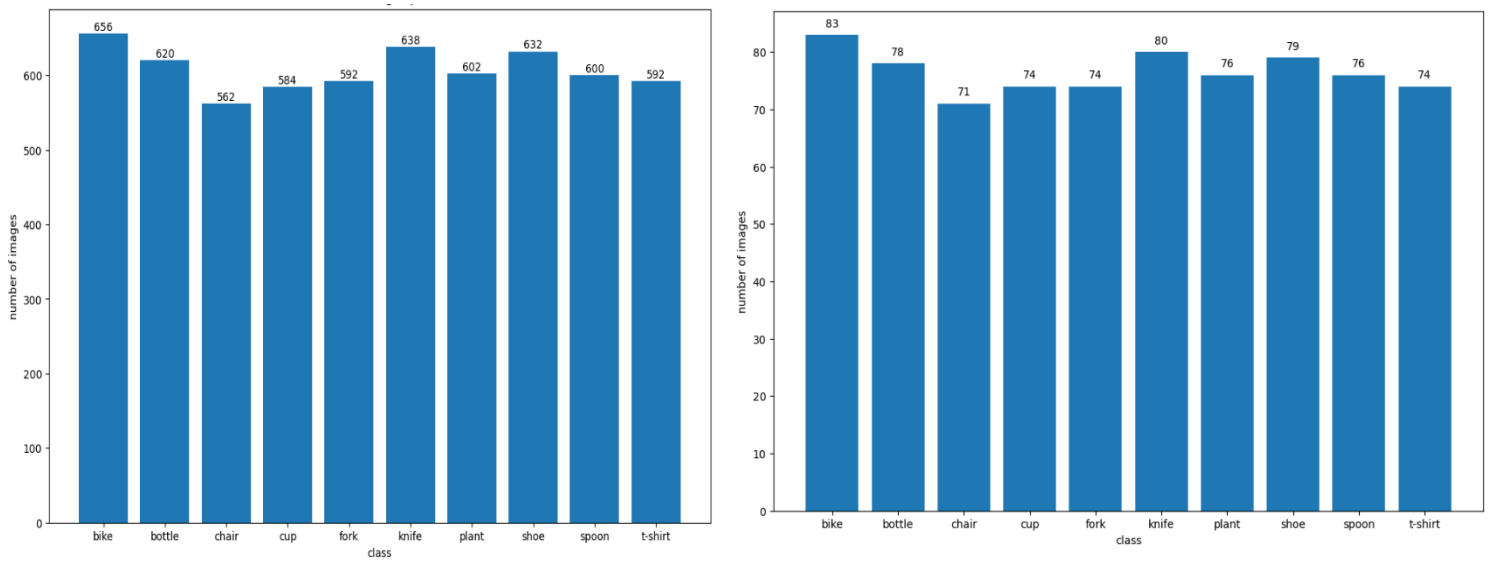

Figure 2: Comparison of the class distribution of images between augmented train data (on the left) and test data (on the right)

Introducing more randomness to the data would improve the generalization of the model. Inorder to introduce randomness, various techniques are considered and applied in each class of training data such as flipping the random images horizontally, rotation of images, by applying color variations (brightness, contrast, saturation) of the images. As a result of augmentation, every image is transformed, the size of original class distribution of training data is doubled after augmentation. So, the model can train on more variations of data images due to augmentation, which could help the model to make accurate predictions with high confidence.

The applied parameters were chosen, in a way to preserve the class identity in the images while introducing variations that improve the model's ability to generalize across different real-world scenarios.

Augmentation Technique	Applied Parameters	Reason for Selection
Random Resized Crop	Size: (224, 224)	Ensures fixed input size to the model and enabling them to convert to tensor later.
Random Horizontal Flip	p=0.5 (50% probability)	To introduce data by flipping the random images
Random Rotation	±30 degrees	To introduce the images with different angles
Color Jitter	brightness=0.2, contrast=0.2, saturation=0.2, hue=0.2	To introduce different lightening conditions, in terms of brightness, contrast, saturation of colors
ToTensor	Convert images to tensors	Standardizes the input format for deep learning models, ensuring compatibility with PyTorch

Table 1: Table describing the reason for selection of the technique and parameters in the data augmentation techniques

1.3 Dataflow Implementation for Efficient Image Transformation and Augmentation

The augmentation was performed by ImageTransformer class primarily designed to transform (rotations, flip, color augmentation), augmenting the training data, resizing the test data and saving the transformed images. The images are transformed into a batch in the ImageTransformer, rather than loading all the 3,804 images at once, which otherwise would result in out-of-memory error in most systems. ImageTransformer reads all-image paths and the respective class indices as a tuple and saves the indices of the tuple, as soon as it's initialized with image path. Then, the data loader loads the batch images during iteration by passing the tuple indices as a batch, which get transformed and saved at the mentioned save_dir directory. The clear visualization of dataflow is referred in the flowchart Figure3. During the image transformation, random states are controlled by random state parameters. This would help to maintain reproducibility in transformed images, even if the class runs multiple times.

The global and per-class statistics, especially mean and standard deviation, were calculated by compute_class_stats function, which is primarily designed to be used later while normalizing at different stages of k-fold cross validation and best model retraining.

Inorder to save the save computational time while accessing the training and test data, the data were saved data loaders using pickle along with those random states generated through controlled parameters., calculated per-class and global statistics and batch size considered.

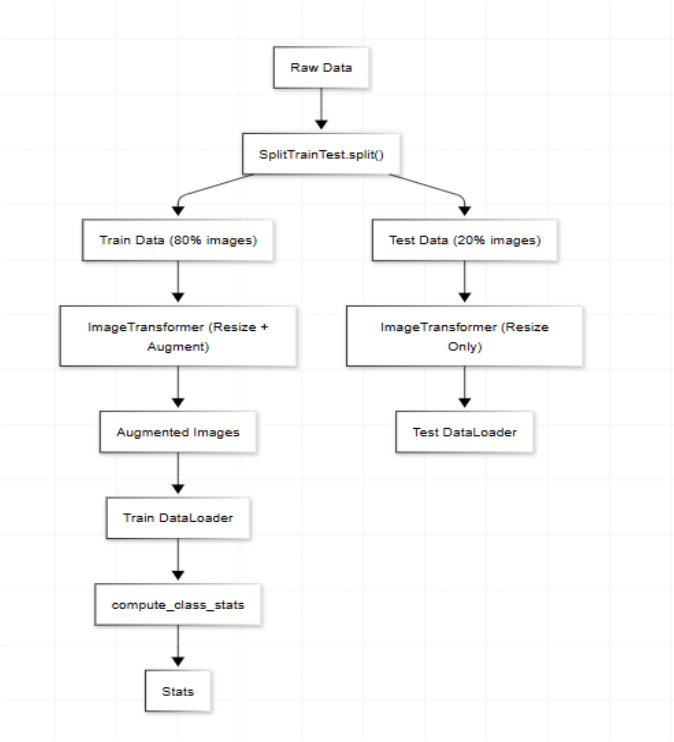

Figure 3: Flowchart indicating transformation of data for prepration of traning and test data

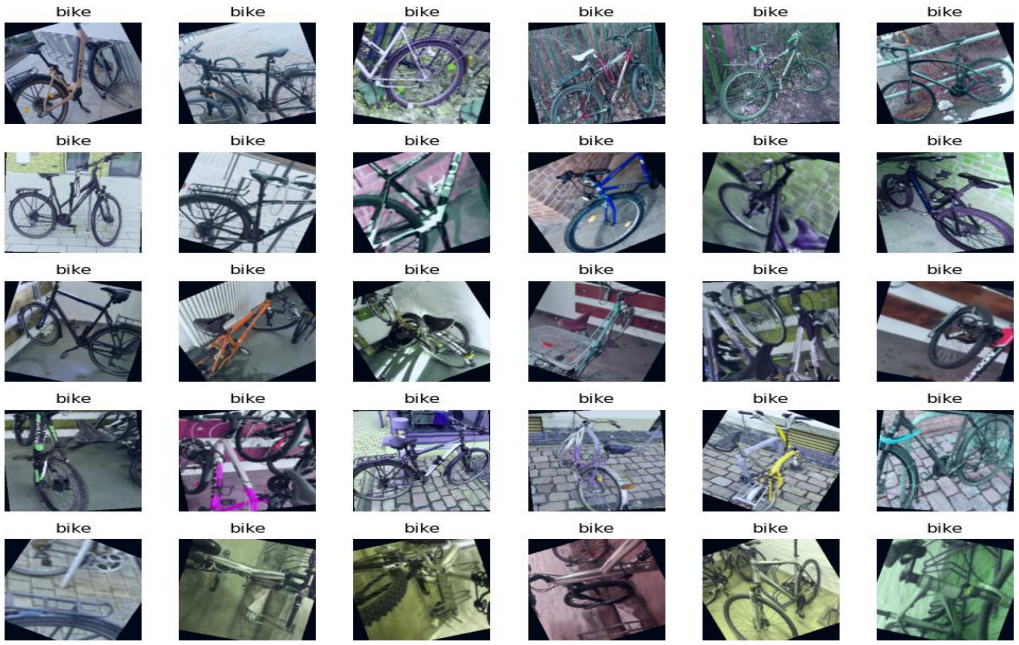

Figure 4:Visualization of sample of bike images after data augmentation.

2. Description of Residual Neural Network-50 Implemented:

Residual Neural Networks (ResNets) are a type of convolutional neural network (CNN) designed to address the vanishing gradient and degradation problems that arise when training deep networks. One of the primary challenges in deep networks is the vanishing gradient problem, where gradients diminish during backpropagation, preventing effective weight updates. This makes it difficult for very deep networks to learn. Although techniques like batch normalization help stabilize training, they do not fully resolve the issue. Additionally, deeper networks were observed to have higher training errors, not due to overfitting but because of difficulties in optimizing deeper architectures [1, 2].

To overcome these challenges, the Residual Learning Framework was introduced. Instead of learning a direct mapping H(x), the network learns a residual function which makes optimization easier[3]

$$F(x) = H(x) - x$$

The input ‘x’ is added directly to the transformed output F(x), as shown as “addition” in Figure 4 through a shortcut connection, ensuring that information is preserved, and gradients can flow easily. This identity mapping helps maintain a constant error even as the network depth increases. Since the model only needs to optimize the residual function F(x) rather than learning a direct transformation, it avoids vanishing/exploding gradients and improves convergence.

$$H(x) = F(x) + x$$

2.1 Construction of Bottleneck Block - Building Blocks of Resnet-50

The key component of this framework is the bottleneck block, a well-structured building block that enables deeper architecture.

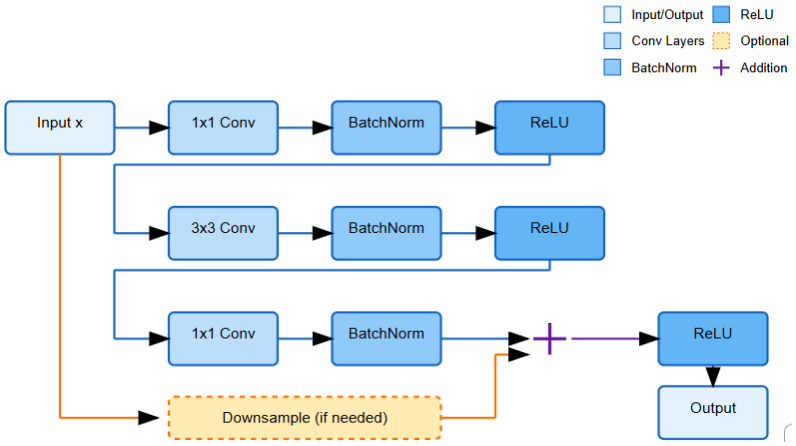

Figure 5: Architecture of the implemented Bottleneck Residual Block in ResNet-50, illustrating the three convolutional layers along with batch normalization and RELU transformation

Each bottleneck block of Resnet-50 is implemented with three convolutional layers:

- **1×1 Convolution (Dimensionality Reduction):** This layer reduces the number of feature channels before applying the computationally expensive 3×3 convolution, making the operation more efficient. After this convolution, the number of channels is halved.

- **3×3 Convolution (Feature Transformation):** This layer extracts spatial features from the reduced-dimensional representation, learning complex patterns while maintaining efficiency using 3x3 kernel. After the convolution, the number of out channels remain the same as the out channels of 1x1 convolution.

- **1×1 Convolution (Dimensionality Restoration):** This final layer restores the original number of channels so that the transformed output can be added to the input via the shortcut connection, enabling effective residual learning. The shape of the out channels is doubled after this convolution compared to the out channels of previous convolution.

2.2 Construction of a Residual Layer

Each residual layer is implemented by stacking the multiple Bottleneck Blocks, where:

- The first Bottleneck Block uses the given stride, considered stride:1 as default, adjusts to input size.
- The subsequent blocks operate normally, as the output shape of the residual layer is maintained as the number of input channels by expansion factor of 4 as shown in Figure 5.

```
# Renewing every residual layer and appending the set of bottlenecks
residual_layers=[]
residual_layers.append(BottleneckBlock(self.in_channels,
                                       out_channels,
                                       stride,
                                       downsample))

# Updating input channels for the next residual layer by multiplying the
# outchannels by the exanpansion factor 4 for resnet-50
self.in_channels=out_channels*BottleneckBlock.expansion

# Appending the bootleneck blocks to form residual layer
for _ in range(1,blocks):
    residual_layers.append(BottleneckBlock(
        self.in_channels,
        out_channels))

return nn.Sequential(*residual_layers)
```

Figure 6: Demonstration of creation of residual layers from bottleneck blocks

- If in case, a mismatch of shapes is caused during the transition between the different residual layers, i.e. output of the previous residual layers doesn't match the input shape of the current layer, a down sampling operation can be performed as shown in implementation of Figure 6.

```
# Initializing to downsample if the output channels is not equal to input channels and stride>1
if stride!=1 or self.in_channels!=out_channels*BottleneckBlock.expansion:
    downsample=nn.Sequential(
        nn.Conv2d(self.in_channels, out_channels*BottleneckBlock.expansion,
                  kernel_size=1,stride=stride,bias=False),
        nn.BatchNorm2d(out_channels*BottleneckBlock.expansion))
```

Figure 7: Condition of down sampling during shape mismatch of residual layers

2.3.1 Workflow Construction for Forward Propagation in ResNet-50

Before diving into residual layers, ResNet-50 begins with standard convolutional processing:

2.3.1. Initial convolution & max pooling layers:

a. **7×7 Convolution**: The input image (e.g., 3×224×224 for RGB images) passed through a 7×7 convolution layer results in 64 channels with a stride of 2 and padding 3. This layer helps capture low-level features such as edges and textures. The size of the output image after 7x7 convolution is computed as (64,112,112).

   The output shapes of the set of feature maps were decided based on the input size, kernel size, padding and stride based on the following convolution equation. The output size for an image of input size 224, 7x7 kernel, padding 3, stride 2 is 112 x 112.

$$\text{Output Size} = \frac{\text{Input Size} - \text{Kernel Size} + 2 \times \text{Padding}}{\text{Stride}} + 1$$

b. **Batch Normalization**: After the 7x7 convolution, an initial batch normalization is performed to stabilize the training and improve the convergence.

c. **Rectified Linear Unit (ReLU) activation**: After each initial convolution, the ReLU activation is applied to introduce non-linearity in the feature maps in the model estimation and detect non-linear patterns in the images.

d. **Max-Pooling:** Max pooling is applied after initial convolution, ensuring that only the most relevant spatial information is forwarded while discarding redundant data. 3×3 max-pooling layer with a stride of 2 reduces spatial dimensions. After Max-Pooling, the outset of feature maps size is calculated to be reduced to 56x56 with 64 channels resulting in a shape (64, 56, 56) using convolution equation.

2.3.2. Forward Propagation Through Residual Layers:

Once the input has passed through the initial convolutional, batch normalization, activation, and pooling layers, it enters the core of the ResNet-50 architecture: the residual layers. These layers consist of stacked bottleneck blocks, which allow for efficient deep feature extraction while mitigating the vanishing gradient problem through shortcut connections.

Each residual layer is responsible for progressively refining the extracted features by applying transformations through convolutional operations.

a. **Processing in Each Residual Layer:**

- The feature maps enter a sequence of bottleneck blocks, each containing three convolutional layers (1×1, 3×3, and 1×1), as described earlier.

- Each bottleneck block includes a skip (shortcut) connection that allows the original input to bypass the transformation, directly adding to the output of the block.

- If the input and output dimensions do not match (e.g., due to stride changes), a 1×1 convolution is applied in the shortcut path to ensure alignment.

- Batch normalization is applied after each convolution to stabilize training, and ReLU activation follows to introduce non-linearity.

b. **Layer-Specific Adjustments:**

- ResNet-50 consists of four main residual layers, each containing a different number of bottleneck blocks:

Layer 1: 3 Bottleneck Blocks
Layer 2: 4 Bottleneck Blocks
Layer 3: 6 Bottleneck Blocks
Layer 4: 3 Bottleneck Blocks

- The first bottleneck block in each residual layer adjusts the feature map size using a stride of 2, reducing spatial dimensions while increasing depth.

- The subsequent blocks maintain the same spatial dimensions while refining feature representations.

c. **Feature Transformation at Each Layer:**

- Each residual layer learns progressively higher-level representations, starting from basic edges and textures in the early layers to complex patterns and object structures in deeper layers as indicated by output feature map size of Table2 below.

- The increasing depth (64, 128, 256, 512 channels) allows the network to capture richer hierarchical features, essential for robust classification.

Residual Layer	Input Feature Map	Transformation in Bottleneck Blocks	Output Feature Map
Residual Layer 1	(64, 56, 56)	1×1 Conv (64 → 64) → 3×3 Conv (64 → 64) → 1×1 Conv (64 → 256)	(256, 56, 56)
Residual Layer 2	(256, 56, 56)	1×1 Conv (256 → 128) → 3×3 Conv (128 → 128) → 1×1 Conv (128 → 512)	(512, 28, 28)
Residual Layer3	(512, 28, 28)	1×1 Conv (512 → 256) → 3×3 Conv (256 → 256) → 1×1 Conv (256 → 1024)	(1024, 14, 14)
Residual Layer 4	(1024, 14, 14)	1×1 Conv (1024 →512) →3×3 Conv (512 →512) → 1×1 Conv (512 →2048)	(2048, 7, 7)

Table 2:Table illustrating feature size transformation in all 4 residual layers of Resnet-50

2.3.3. Final Average Pooling and Fully Connected Layers

After passing through the residual layers, the extracted features are in a high-dimensional space. Before classification, these features need to be condensed into a more manageable form:

a. **Average Pooling:**

- Instead of using a traditional fully connected layer with flattening, ResNet-50 employs an average pooling layer to summarize feature maps into a single vector.

- The average pooling layer computes the spatial average of each feature map, resulting in a 1D feature vector of size equal to the number of output channels (512).
- This approach significantly reduces the number of parameters, minimizing overfitting risks.

b. **Fully Connected Layer and SoftMax Activation**

- The pooled feature vector is passed through a fully connected (dense) layer that maps it to the final class probabilities. It converts the 2048 feature maps into 10 logits.

- A SoftMax activation function is applied, converting raw logits into probability scores for each class.

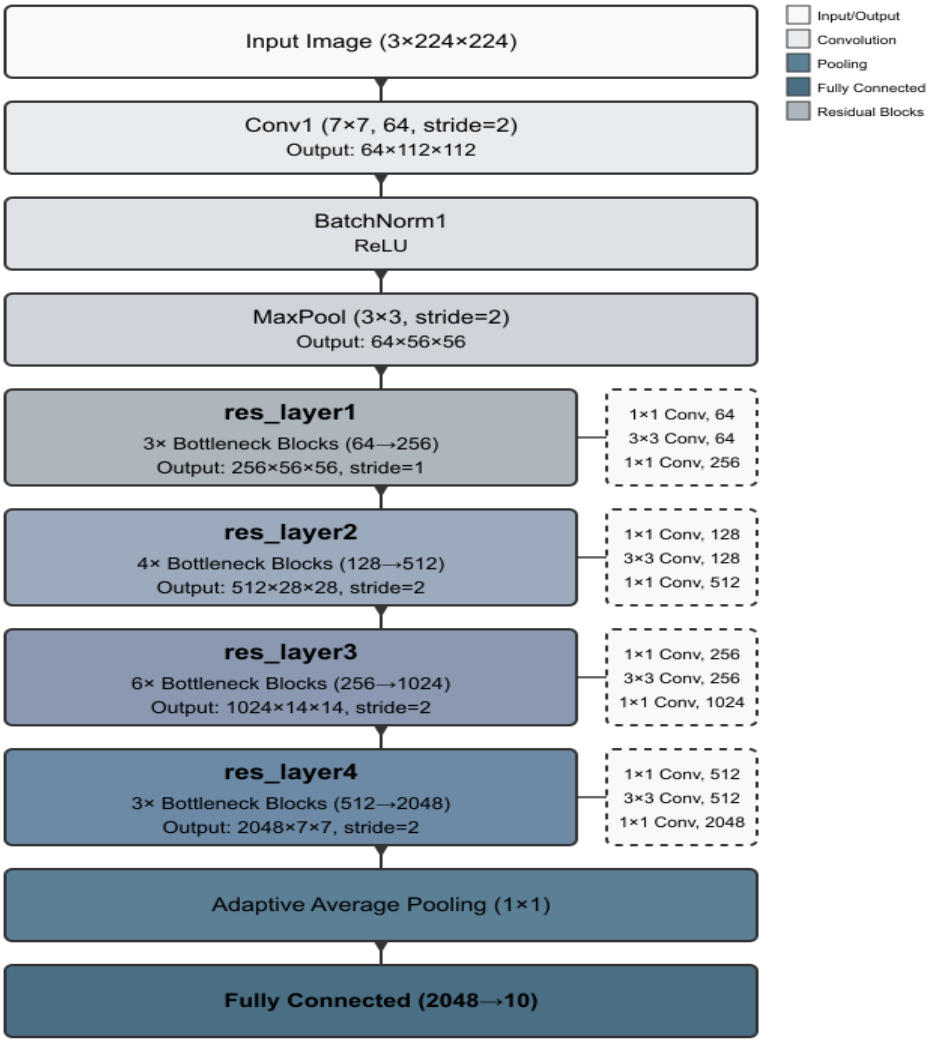

Figure 8: Implemented ResNet-50 CNN architecture featuring an input layer (3×224×224), initial convolutional layers, and four residual layer blocks with bottleneck structures (containing 3, 4, 6, and 3 bottleneck blocks respectively), followed by pooling and a fully convolutional layer.

3. Training of ResNet-50 CNN model on Image database:

The training pipeline is primarily designed to train the model with the best parameters to perform well on unseen test data, aligning with the objectives of machine learning. ModelTrainingPipeline class in the 'main.py' script is created to automate this purpose, adopting Object oriented programming adhering to modular workflow methodology, from efficient data loading, preprocessing of data, augmenting the data, identifying the best parameters, training the model with best parameters. For each step of the machine learning process, different scripts have been created with their own main function and can perform the task by themselves upon meeting the requirements of the class and function documentation.

3.1 Initialization of environment:

Before preprocessing, the class initializes and sets up the environment by defining paths for data, results, and plots. It also configures the computer device (CPU or GPU) to ensure efficient computation.

3.2 Data Preparation:

It is important to ensure the data is prepared by splitting of augmented data into train and test data and converting to train_dataloader and test_datalaoder for efficient loading of augmented data and reproducibility as illustrated in the 1.3 chapter and Figure3. The dataset is split into training and test sets (typically 80% training, 20% testing) using the custom SplitTrainTest module.

Decisions during preparation of the data

- **Reproducibility:** The splitting process shuffles images per class, ensuring a randomized and unbiased split while maintaining the class distribution.

- **Resizing train and tensor for image size standardization**: After splitting the data into train and test data, both the datasets are resized and converted to tensor in order to maintain standardization of the data. Moreover, PyTorch requires the image to be a tensor for to be with [0,1] for performing data augmentation techniques.

- **No Normalization while preprocessing:**
    - While normalization is important during training, the decision to not to normalize during preprocessing has been made, because normalizing with the computed mean and standard deviation of the whole data would expose the influence of test data while training. This results in over-optimistic accuracies during training and may result in overfitting.

    - Train data is also explicitly not normalized at the stage as we are performing k-fold cross validation during hyperparameter tuning. During k-fold cross validation, sets

of training images are passed as folds and accuracies are measured on each subset of fold. If train data is normalized at the preprocessing stage, this would include the mean of standard deviations of all the train data in each image, which would again result in over-optimistic accuracies for each fold and may result in choosing not the best hyperparameter for the data to train the model.

3.3. Hyperparameter Tuning:

3.3.1 Defining the Search Space for Hyperparameter Tuning

Reason for Grid search:

- Grid search was chosen because it thoroughly explores all possible combinations, ensuring the best hyperparameters are found. This method works well for small search spaces, like the one defined in the parameter grid below in Table3. In contrast, other methods like Random Search and Bayesian Optimization may overlook the optimal combination and require a more complex setup.
- For the implementation of hyperparameter tuning the search space is defined with 16 combinations with 2 combinations each, as shown in Table3.
    - **Decision on considered learning rate parameters:**
        - Learning rates of 0.001 and 0.01 are good choices because they help the model gradually update the optimizer's weights and move toward the global minimum.
        - If 0.001 converges too slowly or doesn't reach the minimum due to limited epochs, 0.01 offers a more aggressive approach for faster convergence.

    - **Decision on considered learning rate parameters:**
        - For optimization, both Stochastic Gradient Descent (SGD) and Adaptive moment estimation (Adam) optimizers are considered for hyperparameter tuning. SGD is preferred for its stability with large datasets, while Adam performs well in noisy environments. So, both SGD and Adam optimizer.

    - **Decision on considered learning rate parameters:**
        - For hyperparameter tuning, 7 and 12 epochs were chosen to balance training efficiency, computational cost, and model generalization.
        - If more computational resources are available, increasing to 20 epochs could be beneficial for further optimization.

    - **Decision on considered batch size parameters:**

- To determine the optimal batch size, 16 and 32 were considered. A batch size of 16 can improve generalization but may introduce noise, leading to less stable training.

- A batch size of 32 offers more stable training while maintaining efficiency.

- Training a ResNet-50 model with batch sizes beyond 32 would require significant computational power and memory, making larger batch sizes less practical.

- **Decision on considered number of folds:**
  - k = 2 was chosen due to limited time and computational resources while still allowing for cross-validation.
  - If more computational resources were available, 5 folds would be a better choice, as it provides a balance between training data sufficiency and reducing bias.

Hyperparameter tuning	Considered Parameters
Learning Rate (--lr)	0.001, 0.01
Batch Size (--bs)	16, 32
Epochs (--epochs)	7, 12
Optimizer (--optim)	SGD, Adam
Number of K-Folds (- k_folds)	2

Table 3: Considered parameters for hyperparameter tuning.

3.3.2. Performing Hyperparameter Search

Hyperparameter tuning is performed to identify the best parameters of Resnet-50 model on the train data such as learning rate, optimizer, batch size, number of epochs. The hyperparameter tuning works by conducting grid search over a combination set of hyperparameters using k-cross validation. The idea of k-fold cross validation is to train the model on major subsets of the training data "training fold" and at the same time test the model on the minor(last) subset of fold training data called "validation fold" based on how the model learnt on the training fold. The process is repeated for specified number of epochs on all combinations of parameters. During the training of model, CrossEntropyLoss()loss function as it computes the log-probabilities of each class of multi-class classification problem making training more stable.

- **Decisions**
  - The model was trained using 16 different combinations of training data to maximize learning.

- During training, `torch.no_grad()` and `model.eval()` functions in PyTorch were specifically used for validation. This reduces memory usage and speeds up inference since gradients are not needed during validation.

- During hyperparameter tuning, accuracy was estimated using validation data. The best validation accuracy was tracked and updated throughout training by comparing with the previous best validation accuracy, as shown in Figure 8. This approach ensures the selection of the best parameter combinations from the grid.

```
# Finding the best validation accuracy by comparision with previous best in the loop
if avg_val_accuracy>=best_accuracy:
    best_accuracy=avg_val_accuracy
    self.best_parameters = params
    self.best_model_state = model_state
    self.best_performances={
            "train_loss":results_df["train_loss"],
            "train_accuracy":results_df["train_accuracy"],
            "val_loss":results_df["val_loss"],
            "val_accuracy":results_df["val_accuracy"],
            "batch_size":batch_size,
            "num_epochs": num_epochs,
            "val_preds": val_preds,
            "val_labels": val_labels}
```

Figure 9: Validation accuracy calculation during hyperparameter tuning of hyperparametertuning.py script

- **Normalization on each fold before training**:
  Normalization is performed on every fold just before retraining to avoid data leakage possibility. This ensures proper validation on each fold just on the training fold that the model has learnt.

3.3.3. Possible improvements during Hyperparametertuning:

- **Early stopping**:The hyperparametertuning was implemented for every epoch. But it would be to better to use early stopping, where the training halts if the performance worsens. This would save more computational time and memory during training.

- While training the model, only accuracy and loss performance metrics were predicted. It is advisable to choose F1 score to determine precision-recall trade offs.

4. Retraining the model:

Decisions:

- In this step, the model will be trained again with the entire training data, unlike in hyperparameter tuning which trains only on different subsets of train data, with the

best parameters. The retraining of the model ensures the model trains on full dataset along with the best parameters.

- Before retraining training the model, the entire training data is normalized to determine to improve the learning of model during training. The implementation of this can be observed in Figure9.

- During training and evaluation, it is assumed that the last epoch would produce the best accuracy. So, the best train and validation accuracy and loss is considered.
- The Hyperparameter tuning and retraining is performed with CrossEntropyLoss function, as it is suitable for multi-class classification problems.

```
# Loading the training data and stats
train_loader, train_stats = load_dataloader("train_dataloader.pth")

# Normalizing the full training data
normalized_train = TransformedSubset( train_loader.dataset,
                                      transform=transforms.Normalize(mean=train_stats['global']['mean'].tolist(),
                                                                     std=train_stats['global']['std'].tolist()))

# Creating the DataLoader with best batch size (already loaded from checkpoint)
full_train_loader = DataLoader(normalized_train,
                               batch_size=self.model.hyperparameters["batch_size"],
                               shuffle=True)

# training the model on full training data
train_losses, train_accuracies, _, _, _, _ = train_evaluate(
    model=self.model,
    train_batch_loader=full_train_loader,
    criterion=self.criterion,
    optimizer=self.optimizer,
    num_epochs=self.model.hyperparameters["num_epochs"],
    device=self.device,
    val_batch_loader=None)
```

Figure 10: Python code illustrating the consideration of full training data while retraining on the test data along with normalization. This code is part of "retrain_full_model" of ModelTrainingPipeline

5.Evaluation of models on test data:

Decisions:

- After training the model on the entire training data, model is evaluated on the test data using test data loader. Before evaluating the model, the test data is normalized with the mean and standard deviation of training data computed during the preparation of the data as described in code of Figure 11. This ensures standardization of images as required by Resnet-50 model. Furthermore, the accuracy and loss curves are plotted.

```python
for batch_idx, (images, labels) in enumerate(test_loader):
    # Storing the original images for visualization before normalization
    if batch_idx == 0 and visualize:
        # Convering to numpy for visualization later
        batch_images_np = images.permute(0, 2, 3, 1).cpu().numpy()
        all_batch_images = batch_images_np
        all_class_indices = labels.cpu().numpy()

    # Applying the normalization
    normalized_images = torch.stack([normalize(img) for img in images])

    # Enabling GPU
    normalized_images, labels = normalized_images.to(device), labels.to(device)

    # Forward pass: Predicting class for each through optimization with loss function
    outputs = model(normalized_images)
    loss = criterion(outputs, labels)
    test_loss += loss.item()

    # Predicting the label with maximum confidence and calculating the accuracy
    _, predicted = torch.max(outputs.data, 1)
    test_total += labels.size(0)
    test_correct += (predicted == labels).sum().item()
```

Figure 11: Python code part of test.py script, illustrates the performing normalizing before testing.

- It has been decided to collect the accuracy of the test loss on the test data, as well as the model predictions on the test data.

Data Loading Conditions for running test.py script:

- The test data path must exist and contain images organized in class folders. Since the model is trained in 10 different datasets. It is expected that the test data also have 10 different datasets to load them as batches or even shuffle them.
- The script needs access to previously saved training statistics for normalization
- The "train_dataloader.pth" file must be accessible for loading normalization parameters.

6. Result discussion:

6.1 Best Hyperparameters:

After rigorous training and validation on different sets of the training for different combinations of parameters, the table below mentioned a overview of the performance of the model on training data during hyperparameter tuning.

Learning Rate	Batch size	Epochs	Optimizer	Avg. Val. Accuracy	Average Validation loss
0.001	16	7	Adam	19.88%	2.25
0.001	16	7	SGD	31.90%	2.17
0.001	16	12	Adam	30.82%	2.72
0.001	16	12	SGD	36.61%	2.46
0.001	32	7	Adam	25.34%	2.38
0.001	32	12	Adam	27.37%	2.74
0.001	32	7	SGD	31.45%	1.54
0.001	32	12	SGD	31.37%	2.67
0.01	16	7	Adam	28.73%	2.78
0.01	16	12	SGD	32.56%	1.8
0.01	16	7	Adam	32.26%	2.6
0.01	16	12	SGD	31.34%	2.78
0.01	32	7	Adam	25.56%	2.56
0.01	32	12	Adam	32.37%	2.67
0.01	32	7	SGD	34.17%	2.56

Table 4: Overview of the average validation accuracy and average loss during hyperparameter tuning for all 16 combinations.

- It can be observed that the combination of SGD optimizer, 16 batch size, and 12 epochs achieved the highest validation accuracy (36.61%). So, the parameter combination is considered for retraining on entire data.

- Moreover, the model performs well during the validation when it is being trained on Stochastic gradient descent, compared to Adam optimizer. This is because SGD uses a fixed learning rate and nosier updates, which acts as implicit generalization, encouraging the weights to reach a better minimum. Therefore, generalizing better on unseen data.
- It can be observed that from the below figure that, though the best parameter combination reaches good accuracy on the model. During training, training and validation accuracies were collected in a list on each epoch. Each graph below indicates accuracy and loss on 2 folds. So, epochs 1 to 12 represents the training accuracies and validation losses of the 1st fold and epochs 12 to 24 represents the training and validation accuracies and losses of represents the accuracy of 2nd folds. It can be observed that during second fold the validation loss started unstable unexpectedly and reached to a nominal loss of around 2.5.

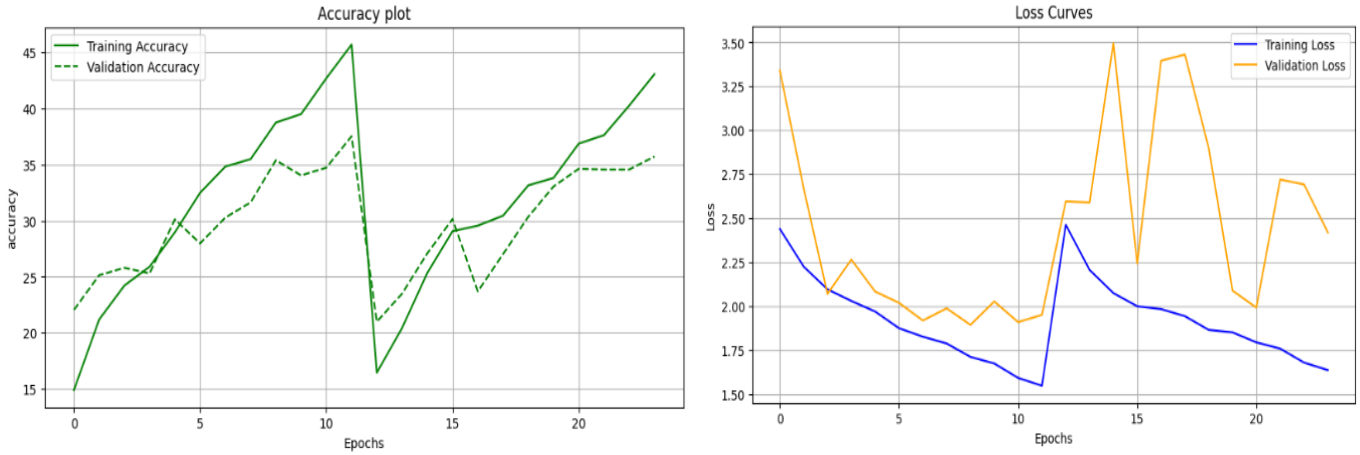

Figure 12:Training accuracy and validation curves for the best hyperparameter combination on learning_rate=0.001, optimizer=SGD, batch size=16, epochs=12, no.of folds=2

6.2 Training the Resnet-50 model on the entire training data:

The results were plotted as accuracy and loss curves as described below.

- As the epochs increase, the accuracy increases and loss decreases, indicating the increasing model learnability and updation of weights by SGD optimizer in the direction of the minimum loss.

- It is observed that models are trained with a training accuracy of 79.89%

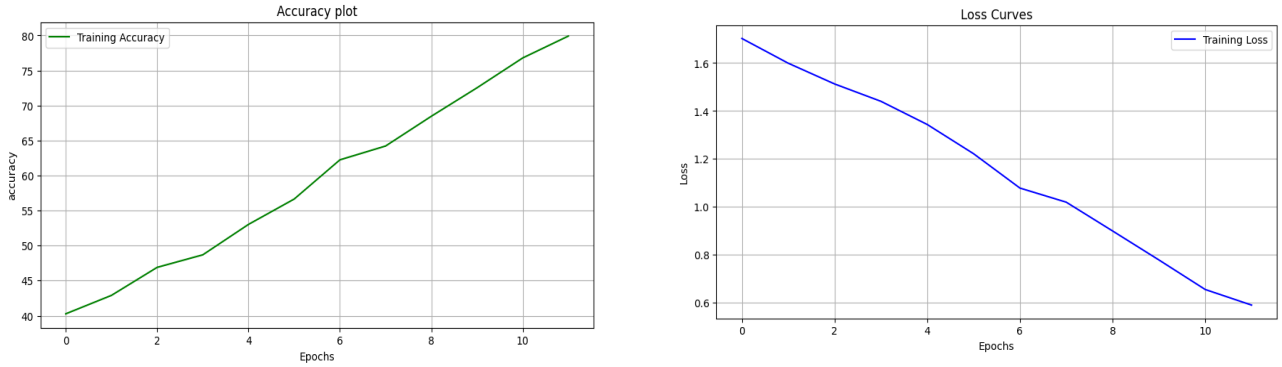

Figure 13: Plots indicating training accuracy and loss curves on learning_rate=0.001, optimizer=SGD, batch size=16, epochs=12

6.3. Results of evaluation of model on test data:

The trained model is evaluated on the test data using the ‘test.py’ script. The model performs. The model weights that were saved during training as pickle file are loaded to the initialized model along with the CrossEntropy loss function. Then model is evaluated on the test data by fixing the weight parameters.

After evaluating the test data, it can be observe hat the model performed moderately good with an accuracy of 55.29% and a loss of 1.98.

```
Test Results:
Test Loss: 1.9800, Test Accuracy: 55.29%
Confusion matrix saved to C:\Users\saiye\Desktop\linux_scientific_computing_project2\change_code\new_scripts\..\results\p
ts\test_confusion_matrix.png

Sample predictions:
Image 1: Predicted: bottle, Actual: bike
Image 2: Predicted: bike, Actual: bike
Image 3: Predicted: bike, Actual: bike
Image 4: Predicted: bike, Actual: bike
Image 5: Predicted: bike, Actual: bike
Sample predictions visualization saved to C:\Users\saiye\Desktop\linux_scientific_computing_project2\change_code\new_scri
s\..\results\plots/sample_predictions.png
Evaluation complete. Overall accuracy: 55.29%
```

Figure 14:Result on test accuracy and on test loss after execution of test.py script

From the correlation matrix mentioned in the Figure15 below, it can be observed that model performed very good in all other classes, except in bottle, spoon and t-shirt. The model should be trained in more new images of these classes or improve the quality of data in these classes for the model to learn well.

This can also be seen on a batch of images randomly selected from the test data in the Figure 16 below.

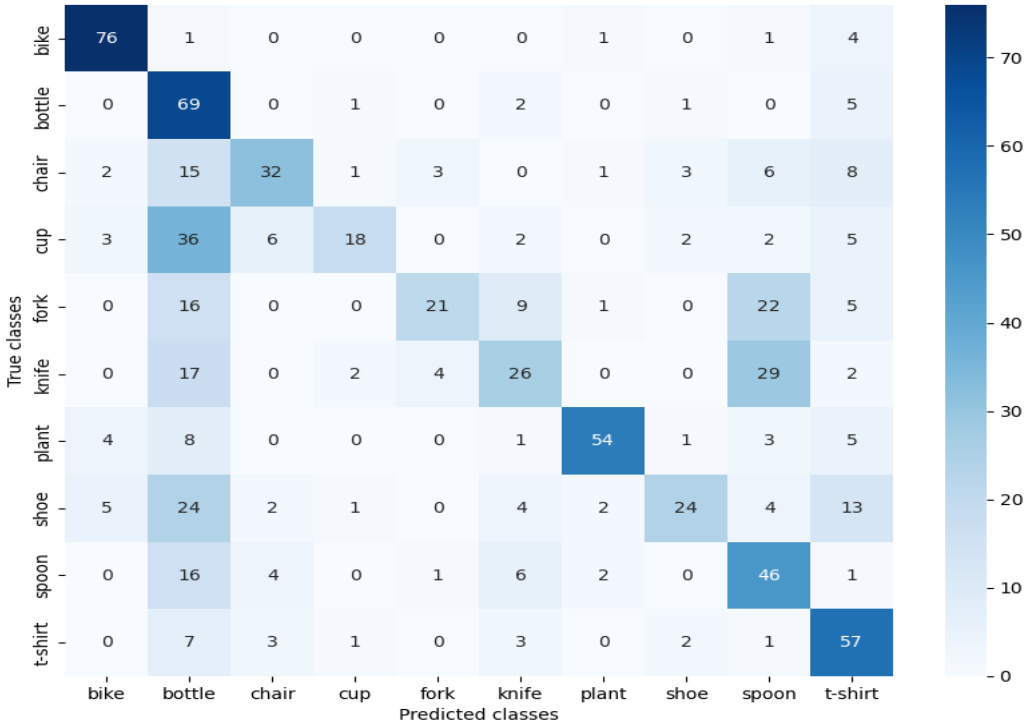

Figure15: Correlation matrix of test data indicating the no. of per-class correct and incorrect classifications

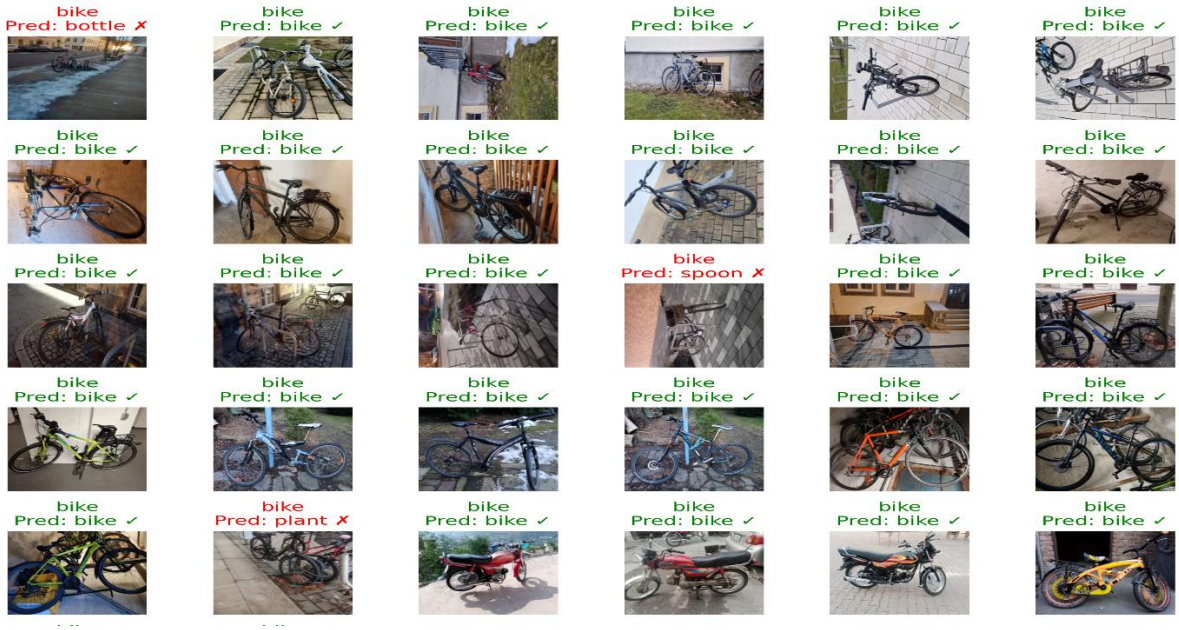

Figure16:Visualization of predicted class labels of the images with their true class names

7. References:

1. K. He and J. Sun. Convolutional neural networks at constrained time cost. In CVPR, 2015.
2. R. K. Srivastava, K. Greff, and J. Schmidhuber. Highway networks. arXiv:1505.00387, 2015.
3. ResNet: He et al. (2015), https://arxiv.org/abs/1512.03385