Individual Final Report – Shumel Siraj

Introduction. An overview of the project

The rapid growth of remote sensing technologies has significantly advanced surveillance and security systems for water bodies. I understand the importance of maritime monitoring in detecting and preventing criminal activities in international waters. Deep learning techniques are now widely used in remote sensing applications for image classification and object detection.

Traditional manual ship classification methods are time-consuming and error-prone, limiting their real-time effectiveness. Thus, I am motivated to develop an automated ship classification system using deep learning techniques to revolutionize maritime monitoring and management.

In this project, I aim to classify maritime scenes using optical aerial images from the visible spectrum to facilitate maritime monitoring and surveillance. Developing an automated ship classification system has the potential to greatly improve our ability to manage maritime activities, making it a crucial area of research in computer vision and remote sensing.

Outline of shared project:

I. Introduction

A. Overview of the shared project

B. Importance and relevance of the project

II. Literature Review

A. Thoroughly read and analyze the research paper.

1. Understand the authors' objectives and methodology
2. Investigate the context and motivation behind the research
3. Discuss the key findings and contributions to the field

III. Data Acquisition and Preprocessing

A. Obtain the dataset required for the project.

B. Conduct initial exploratory data analysis

C. Develop the appropriate data preprocessing pipeline.

1. Data cleaning
2. Feature extraction
3. Feature scaling and normalization

IV. Model Development

A. Train an initial model based on the research paper

B. Evaluate the performance of the initial model

1. Discuss relevant evaluation metrics
2. Analyze areas for improvement

V. Model Refinement

A. Modify the model architecture and parameters based on evaluation results.

B. Retrain the model with the modified architecture.

C. Re-evaluate the performance of the refined model.

1. Discuss improvements and any remaining limitations

VI. Results

A. Present a detailed analysis of the final model's performance.

B. Compare the performance of the initial and refined models.

C. Highlight key findings and insights.

VII. Conclusion

A. Summarize the project's main contributions and achievements.

B. Discuss potential applications and implications of the findings.

C. Identify future research directions and potential improvements.

VIII. Report and Presentation Preparation

A. Compile and organize the project's findings into a comprehensive report.

B. Develop a clear and engaging presentation to effectively communicate the project's outcomes.

C. Prepare for potential questions and discussions related to the project.

2. Description of your individual work. Provide some background information on the development of the algorithm and include necessary equations and figures.

3. Describe the portion of the work that you did on the project in detail. It can be figures, codes, explanation, pre-processing, training, etc.

I utilized some references from my previous semester's Machine Learning 1 project, titled "**COVID-19 Detection from X-rays with CNN & Pre-Trained Models Using Transfer Learning.**" I then adapted the CNN model architectures and other code implementations to suit the requirements of this current project.

I downloaded the dataset directly from Kaggle to the SSH using the Kaggle API, following the instructions provided in the Kaggle dataset download guide text file included in the Git repository.

- **Imported necessary libraries and set environment variables:**

1. Imported required libraries such as NumPy, OpenCV, TensorFlow, and Scikit-learn.
2. Set the TensorFlow warning log level to hide warning messages.

- **Define constants and variables:**
  1. Define the categories of images, number of categories, channels (set to 3 for RGB images), image size, number of epochs, batch size, and learning rate.

- **Preprocessing/Image augmentation**
  - **Check if preprocessed data files exist or if the user forces preprocessing. If not, proceed with the following steps:**
    1. Create an **ImageDataGenerator** object for image augmentation, including rotation, zoom, and flipping.
    2. Loop through each category in the dataset and read the images, assign a label, and preprocess the images by resizing and applying data augmentation.
    3. Append the preprocessed images and their labels to the 'data' list.
    4. Convert the lists of images and labels to NumPy arrays.
    5. Split the data into training, validation, and testing sets using **Scikit-learn's train_test_split** function.
    6. Normalize the image arrays by dividing them by **255** and reshape them for input into a CNN model.
    7. Save the preprocessed data to pickle files.

  - **If preprocessed data files exist and the user doesn't force preprocessing, load the data from the existing pickle files.**
    1. Return the training, validation, and testing datasets (both features and labels).

  - **When executed, the provided code snippet performs the following steps:**
    1. Imports the necessary libraries and sets environment variables.
    2. Defines constants and variables.
    3. Preprocessed the image dataset (including image augmentation, resizing, and splitting into train, validation, and test sets) and saves or loads the preprocessed data in pickle files.

- **Define the CNN model:**
  1. Defined the model that takes number of classes and learning rate as inputs.

2. Define a Sequential CNN model with convolutional layers, batch normalization, pooling layers, and dense layers.

- **Train the CNN model:**
  1. Trained the model that takes the model, training and validation data, the number of epochs, and batch size as inputs.
  2. Computed class weights for handling imbalanced data.
  3. Compile the model with the **'adam' optimizer and 'sparse_categorical_crossentropy' loss.**
  4. Train the model using the given data, class weights, and callbacks for early stopping.
  5. Print the model summary.
  6. Return the trained model.

- **Evaluate the model:**
  1. Evaluate the model on the train, validation, and test data and print the loss and accuracy.
  2. Extract features (Neural Codes) from the CNN model for train, validation, and test data.
  3. Define a parameter grid for KNN hyperparameters.
  4. Perform randomized search with cross-validation to find the best hyperparameters for the KNN classifier.
  5. Train the KNN classifier using the best hyperparameters on the extracted features.
  6. Evaluate the KNN classifier on train, validation, and test features using F1 score, recall, precision, and accuracy metrics. Print the evaluation results.

Results. Describe the results of your experiments, using figures and tables wherever possible. Include all results (including all figures and tables) in the main body of the report, not in appendices. Provide an explanation of each figure and table that you include. Your discussions in this section will be the most important part of the report.

The table presents the performance of various deep learning models (CNN, VGG16, VGG19, Inception, ResNet50, and Xception) on three datasets (Validation, Train, and Test) using four metrics: Accuracy, F1 Score, Recall, and Precision. VGG16 demonstrates the best performance across all datasets, with the highest Accuracy, F1 Score, Recall, and Precision. In contrast, ResNet50 has the lowest performance, with the least Accuracy and F1 Score in all datasets. The other models show intermediate performance levels, with Inception and Xception having relatively similar results, while VGG19 is slightly better.

Model	Dataset	Accuracy	F1 Score	Recall	Precision
CNN model	Validation	0.61	0.604	0.619	0.656
	Train	0.68	0.662	0.682	0.719
	Test	0.62	0.614	0.628	0.659
VGG16	**Validation**	**0.77**	**0.767**	**0.776**	**0.764**
	Train	**0.80**	**0.851**	**0.812**	**0.811**
	Test	**0.78**	**0.775**	**0.784**	**0.772**
VGG19	Validation	0.77	0.765	0.776	0.758
	Train	0.80	0.797	0.805	0.805
	Test	0.76	0.765	0.776	0.767
Inception	Validation	0.73	0.724	0.794	0.795
	Train	0.79	0.793	0.794	0.795
	Test	0.70	0.698	0.704	0.703
ResNet50	Validation	0.47	0.457	0.478	0.487
	Train	0.55	0.526	0.550	0.529
	Test	0.45	0.425	0.454	0.452
Xception	Validation	0.74	0.730	0.740	0.727
	Train	0.78	0.772	0.782	0.790
	Test	0.72	0.707	0.720	0.726

The VGG16 model performs better on RGB images dataset compared to the other models due to its architecture and pre-training. Few Reasons can be:

- **Architecture:** VGG16 has a deeper and more uniform architecture than some other models, consisting of multiple convolutional layers followed by max-pooling layers. This design enables the model to learn complex features and patterns more effectively, which is essential when dealing with high-resolution RGB images.
- **Pre-training:** VGG16 is pre-trained on a large dataset, ImageNet, which contains millions of images covering a diverse range of objects and scenes. This pre-training allows the model to learn meaningful features from the data, making it a powerful feature extractor that can generalize well to other datasets, including the RGB images dataset in question.

- **Small filters:** VGG16 uses small filters (3x3) in its convolutional layers, which allows it to capture local patterns more effectively. This approach makes it easier for the model to identify and differentiate between various objects and textures within the RGB images.
- **Transfer learning:** VGG16 is well-suited for transfer learning, where the knowledge gained from pre-training can be fine-tuned to perform well on a new, related task. In the case of the RGB images dataset, VGG16 can adapt its learned features to effectively classify and identify objects in the new dataset, leading to better performance compared to other models.

Before arriving at this result, I experimented with various approaches, including different preprocessing methods and image augmentations. One noteworthy attempt, inspired by some articles on the combination of CNN and kNN, involved not using the softmax function for feature extraction. According to these articles, features were extracted without the softmax function and used in kNN. I tried this approach with pretrained models, but it led to poor performance. Possible reasons for this outcome include:

1. **Loss of interpretability:** The softmax function is used to convert raw output values into probability distributions. By not applying softmax, the extracted features won't represent meaningful probabilities, making them harder to interpret and analyze.
2. **Noise and outliers:** As the extracted features are not probability values, they might introduce noise and outliers, which can negatively impact the performance of the kNN classifier.
3. **Suboptimal performance:** The kNN algorithm might not perform as well with these non-probability feature values, leading to inferior classification results.
4. **Reduced robustness:** The presence of noise and outliers may compromise the robustness of the feature extraction process, making the model more sensitive to variations in input data.

Summary and conclusions. Summarize the results you obtained, explain what you have learned, and suggest improvements that could be made in the future.

The better performance of the VGG16 model on the RGB images dataset can be attributed to several key factors, including its deep and uniform architecture, the use of small filters, pre-training on an extensive dataset, and compatibility with transfer learning techniques. The intricate design of the VGG16 model allows it to efficiently learn and generalize from the input data, consequently outperforming other models in terms of accuracy and effectiveness.

From this project I leaned the importance of leveraging pretrained models when dealing with small datasets. Utilizing pretrained models can help overcome the limitations of insufficient data by capitalizing on the knowledge acquired from previous training on larger, diverse datasets. This strategy

can significantly enhance the model's capacity to extract meaningful features and improve its overall performance in the context of the research task at hand.

To improve the performance and explore other deep learning techniques, I can consider the following options:

1. **MixUp augmentation**: MixUp is a data augmentation technique that generates new training examples by mixing images and their corresponding labels. This can help improve generalization and prevent overfitting.
2. **Ensemble methods:** I can train multiple models and combine their predictions to improve overall performance. I can explore techniques like bagging, boosting, or stacking to combine predictions from different models.
3. **Deep Learning Architectures:** I can experiment with different deep learning architectures such as EfficientNet, DenseNet, and MobileNet, which offer varying levels of accuracy and efficiency trade-offs.
4. **Image pre-processing:** I can experiment with different pre-processing techniques such as histogram equalization, adaptive histogram equalization, and denoising to improve the quality of input images, making it easier for the model to learn relevant features.
5. **Learning rate scheduling:** I can experiment with learning rate schedulers like cyclic learning rates, cosine annealing, and exponential decay to adapt the learning rate during training for better convergence.

Percentage of Code

The code showcased in the screenshot was originally developed for a prior machine learning project and has since been tailored to accommodate the demands of this current undertaking. Although no code lines were directly borrowed from online sources, I consulted a range of digital resources to grasp the necessary functions and implementation techniques. Consequently, I adapted these functions and methods to align with the project's unique requirements, guaranteeing that the code was both personalized and optimized to achieve the desired results.

```python
#define the categories
CATEGORIES = ['coast', 'coast_ship', "detail", "land", "multi", "ship", "water"]
num_classes = 7
CHANNELS = 3  # set number of channels to 3 for RGB images

data = []
Image_Size = 100
n_epochs = 50
batch_size = 32
learning_rate = 0.001


def preprocess_data(x, y, force_preprocessing=False):
    # check if preprocessed data exists
    already_preprocessed = os.path.exists('x_train.pkl') and os.path.exists('y_train.pkl') and os.path.exists(
        'x_test.pkl') and os.path.exists('y_test.pkl') and os.path.exists(
        'x_val.pkl') and os.path.exists('y_val.pkl')

    if not already_preprocessed or force_preprocessing:
        # apply image augmentation
        datagen = ImageDataGenerator(rotation_range=45,
                                     zoom_range=0.2,
                                     horizontal_flip=True,
                                     vertical_flip=True)

        # read and preprocess the images
        for category in CATEGORIES:
            path = os.path.join(image_dataset_path, category)  # path of dataset
            for img in os.listdir(path):
                try:
                    img_path = os.path.join(path, img)  # Getting the image path
                    label = CATEGORIES.index(category)  # Assigning label to image
                    arr = cv2.imread(img_path)  # RGB image
                    new_arr = cv2.resize(arr, (Image_Size, Image_Size))  # Resize image
                    new_arr = datagen.random_transform(new_arr)  # apply image augmentation
                    data.append([new_arr, label])  # appending image and label in list
                except Exception as e:
                    print(str(e))
```

```
for features, label in data:
    x.append(features)  # Storing Images all images in X
    y.append(label)  # Storing al image label in y

x = np.array(x)  # Converting it into Numpy Array
y = np.array(y)

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size=0.2, random_state=42)  # 0.125 = 0.1 / 0.8

# load
# normalize images
x_train = x_train / 255
x_val = x_val / 255
x_test = x_test / 255

# reshape images for CNN
x_train = x_train.reshape(-1, Image_Size, Image_Size, CHANNELS)
x_val = x_val.reshape(-1, Image_Size, Image_Size, CHANNELS)
x_test = x_test.reshape(-1, Image_Size, Image_Size, CHANNELS)

# save preprocessed data to pickle files
with open('x_train.pkl', 'wb') as f:
    pickle.dump(x_train, f)

with open('y_train.pkl', 'wb') as f:
    pickle.dump(y_train, f)

with open('x_val.pkl', 'wb') as f:
    pickle.dump(x_val, f)

with open('y_val.pkl', 'wb') as f:
    pickle.dump(y_val, f)

with open('x_test.pkl', 'wb') as f:
    pickle.dump(x_test, f)

with open('y_test.pkl', 'wb') as f:
    pickle.dump(y_test, f)
```

```
        with open('y_val.pkl', 'wb') as f:
            pickle.dump(y_val, f)

        with open('x_test.pkl', 'wb') as f:
            pickle.dump(x_test, f)

        with open('y_test.pkl', 'wb') as f:
            pickle.dump(y_test, f)

    else:
        # load preprocessed data from pickle files
        x_train = pickle.load(open('x_train.pkl', 'rb'))
        y_train = pickle.load(open('y_train.pkl', 'rb'))
        x_val = pickle.load(open('x_val.pkl', 'rb'))
        y_val = pickle.load(open('y_val.pkl', 'rb'))
        x_test = pickle.load(open('x_test.pkl', 'rb'))
        y_test = pickle.load(open('y_test.pkl', 'rb'))

    return x_train, y_train, x_val, y_val, x_test, y_test

def model_definition(num_classes, learning_rate):
    # define the CNN model
    cnn_model = Sequential()
    cnn_model.add(Conv2D(64, (3, 3), activation='relu'))
    cnn_model.add(BatchNormalization())
    cnn_model.add(MaxPooling2D((2, 2)))
    cnn_model.add(Conv2D(32, (3, 3), activation='relu'))
    cnn_model.add(BatchNormalization())
    cnn_model.add(MaxPooling2D((2, 2)))
    cnn_model.add(Flatten())
    cnn_model.add(Dense(128, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.01)))
    cnn_model.add(Dropout(0.5))
    cnn_model.add(Dense(64, activation='relu'))
    cnn_model.add(Dropout(0.5))
    cnn_model.add(Dense(7, activation='softmax'))

    return cnn_model
```

References.

- Feng, Y., Diao, W., Sun, X., Yan, M., & Gao, X. (2019, August 14). *Towards automated ship detection and category recognition from high-resolution aerial images*. MDPI. Retrieved April 20, 2023, from https://www.mdpi.com/2072-4292/11/16/1901
- J. Alghazo, A. Bashar, G. Latif and M. Zikria, "Maritime Ship Detection using Convolutional Neural Networks from Satellite Images," 2021 10th IEEE International Conference on Communication Systems and Network Technologies (CSNT), Bhopal, India, 2021, pp. 432-437, doi: 10.1109/CSNT51715.2021.9509628.
- (2023). TensorFlow. Retrieved from https://www.tensorflow.org/api_docs/python/tf