Understanding Object Classification in Images Using Convolutional Neural Networks

13659848: Junghyun Shin

University of Technology Sydney

31005: Advanced Data Analytics Algorithms, Machine Learning

Subject coordinator: Dr. Jun Li

Abstract

In this study, I delve into the realm of object recognition using Convolutional Neural Networks (CNNs). The dataset, sourced from Kaggle, comprises images with associated annotations, facilitating the extraction of objects and their classification. My methodology encompasses data preprocessing, where images undergo cropping and resizing based on annotations, followed by the design and implementation of a CNN model tailored to this task. Upon training the model, I employ several evaluation metrics and visualization techniques, granting insights into the model's performance. Through this journal, I aim to elucidate the intricacies of each step undertaken, showcasing my comprehensive understanding of the end-to-end process of object classification with CNNs.

Introduction

Object recognition is a fundamental task in the field of computer vision, aiming to identify and classify objects within digital images. The ability to accurately recognize and classify objects has vast applications, ranging from self-driving cars to medical imaging. While traditional methods relied on hand-crafted features, the advent of deep learning, especially Convolutional Neural Networks (CNNs), has revolutionized the domain (LeCun, Bengio, & Hinton, 2015).

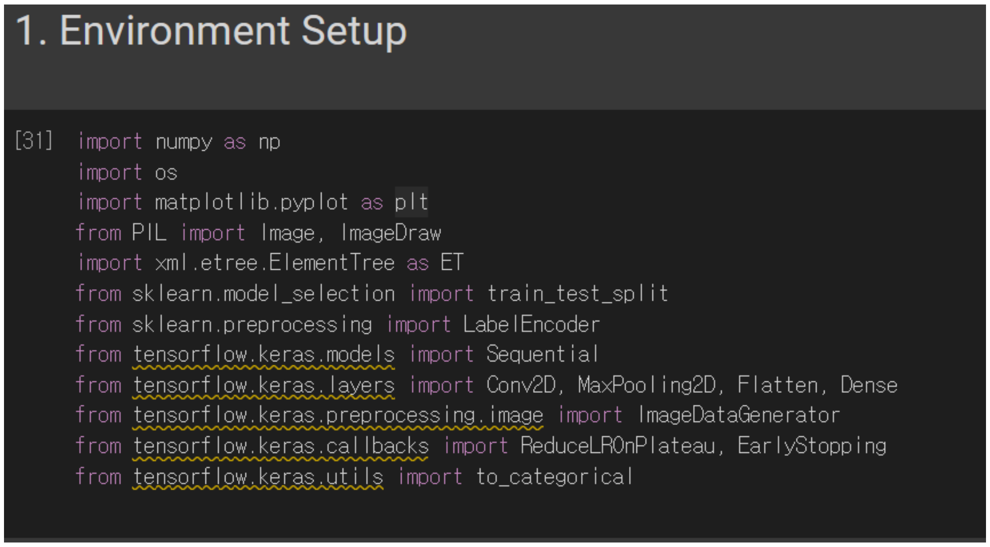
CNNs, with their hierarchical structure, are uniquely suited to process visual data. Their architecture is designed to automatically and adaptively learn spatial hierarchies of features from images. By leveraging multiple convolutional layers, these networks can capture intricate patterns and details, making them a powerful tool for object recognition tasks.

In this study, I aim to delve deep into the application of CNNs for object recognition. Utilizing a dataset from Kaggle, I take the reader through the steps of preprocessing the data, designing the neural network architecture, and evaluating the model's performance. By detailing each step, I aim to provide a comprehensive understanding of the process and intricacies involved in using CNNs for object recognition.

Methodology

Setting Up the Deep Learning Environment

To set up a conducive environment for deep learning, various libraries and tools were imported. Here's a step-by-step explanation of the code and the role of each library:



1. Environment Setup

```
[31] import numpy as np
     import os
     import matplotlib.pyplot as plt
     from PIL import Image, ImageDraw
     import xml.etree.ElementTree as ET
     from sklearn.model_selection import train_test_split
     from sklearn.preprocessing import LabelEncoder
     from tensorflow.keras.models import Sequential
     from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
     from tensorflow.keras.preprocessing.image import ImageDataGenerator
     from tensorflow.keras.callbacks import ReduceLROnPlateau, EarlyStopping
     from tensorflow.keras.utils import to_categorical
```

Numpy: A fundamental package for scientific computing in Python, NumPy offers support for large multi-dimensional arrays and matrices, along with mathematical functions to operate on these arrays.

OS: This module facilitates interfacing with the underlying operating system. It's utilized here to navigate the file system and retrieve the dataset files.

Matplotlib: An essential visualization library in Python, Matplotlib aids in plotting graphs. It is used in this project to visualize the dataset, training progress, and results.

PIL (Python Imaging Library): This library supports opening, manipulating, and saving image file formats. It's crucial in this project for operations like reading images and manipulating their dimensions.

xml.etree.ElementTree: An XML processing library, it's used to parse the XML annotation files associated with the dataset, extracting essential information about object locations and labels.

Scikit-learn: One of the most popular machine learning libraries in Python, it's used in this project for splitting the dataset into training and testing sets.

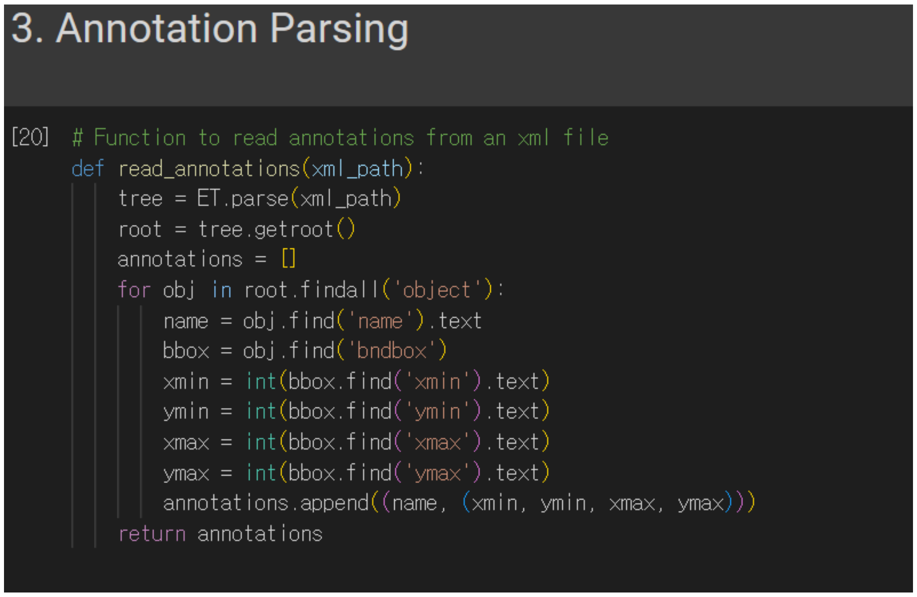
TensorFlow and Keras: TensorFlow is a robust open-source library for numerical computation and machine learning (Abadi et al., 2016). Keras, on the other hand, is an interface for TensorFlow, designed to build and train deep learning models with ease. Several components from Keras, like the Sequential model, layers (Conv2D, MaxPooling2D, Flatten, Dense), and utilities (ImageDataGenerator, callbacks, etc.), are employed to define, compile, and train the neural network.

By setting up this environment, a foundation is laid, equipped with all the tools and utilities required to process the data, build the model, and conduct deep learning experiments.

Data Acquisition and Exploration

The dataset was sourced from Kaggle and stored on Google Drive. After mounting the Google Drive on the Colab environment, the dataset was extracted using the **unzip** command.

Annotation Parsing



3. Annotation Parsing

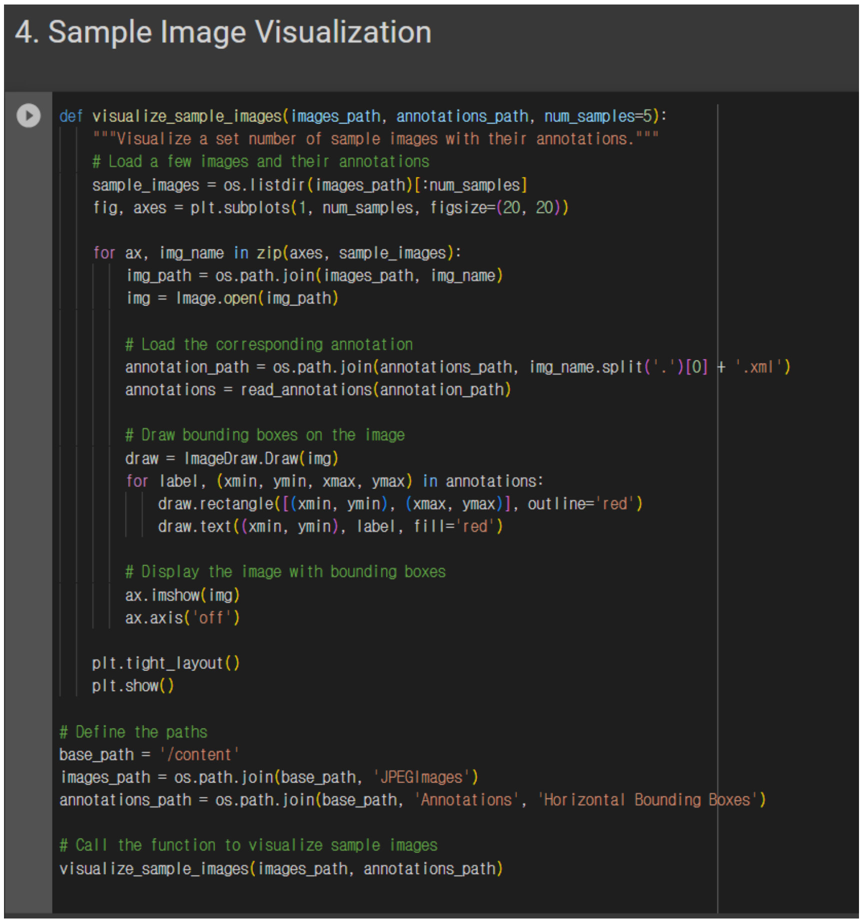
```
[20] # Function to read annotations from an xml file
     def read_annotations(xml_path):
         tree = ET.parse(xml_path)
         root = tree.getroot()
         annotations = []
         for obj in root.findall('object'):
             name = obj.find('name').text
             bbox = obj.find('bndbox')
             xmin = int(bbox.find('xmin').text)
             ymin = int(bbox.find('ymin').text)
             xmax = int(bbox.find('xmax').text)
             ymax = int(bbox.find('ymax').text)
             annotations.append((name, (xmin, ymin, xmax, ymax)))
         return annotations
```

In this project, object annotations, which denote the location and class of items within an image, are stored in XML files. The **read_annotations()** function serves a pivotal role in parsing these files. The primary aim is to extract two essential pieces of information from each XML document:

- The label of the detected object, representing its class.
- The bounding box coordinates which demarcate the object's location in the image.

By iterating through the XML structure, the function efficiently gathers this data for every annotated object in an image, returning a list of tuples. Each tuple contains the object's label and its bounding box coordinates. This extraction process ensures that the model has the necessary information to learn from the annotated images.

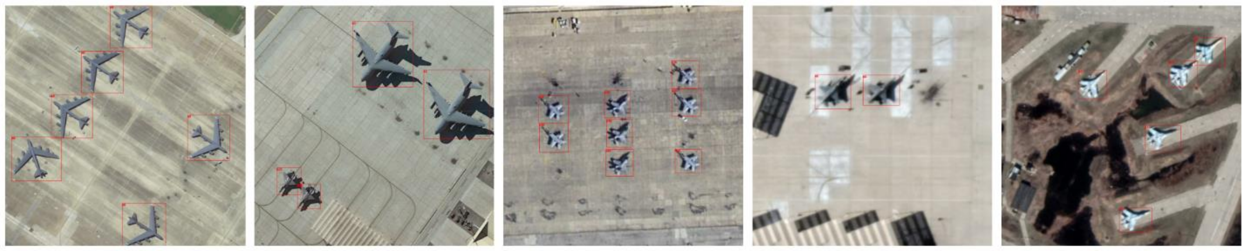
Sample Image Visualization



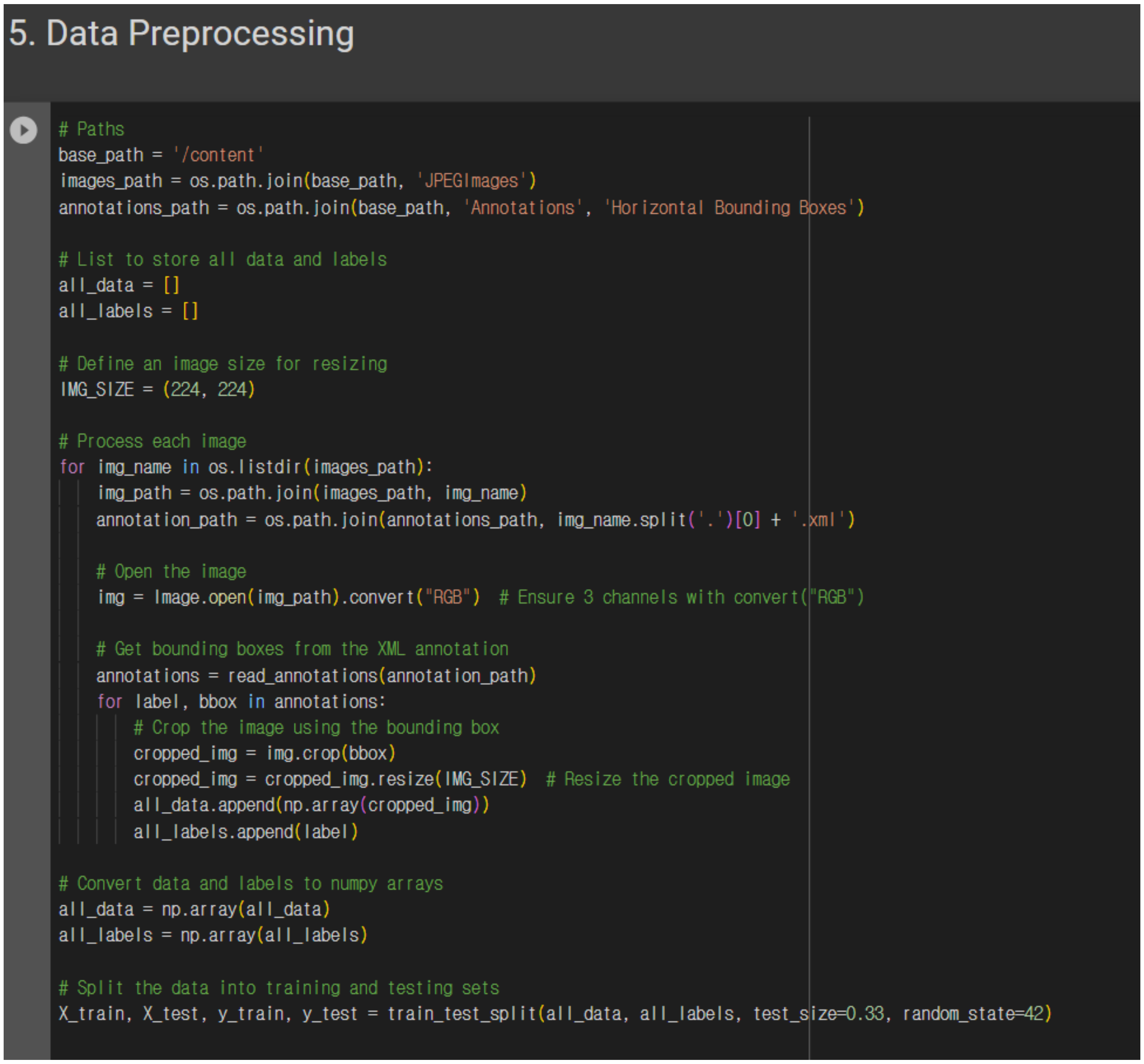
4. Sample Image Visualization

```
def visualize_sample_images(images_path, annotations_path, num_samples=5):
    """Visualize a set number of sample images with their annotations."""
    # Load a few images and their annotations
    sample_images = os.listdir(images_path)[:num_samples]
    fig, axes = plt.subplots(1, num_samples, figsize=(20, 20))

    for ax, img_name in zip(axes, sample_images):
        img_path = os.path.join(images_path, img_name)
        img = Image.open(img_path)

        # Load the corresponding annotation
        annotation_path = os.path.join(annotations_path, img_name.split('.')[0] + '.xml')
        annotations = read_annotations(annotation_path)

        # Draw bounding boxes on the image
        draw = ImageDraw.Draw(img)
        for label, (xmin, ymin, xmax, ymax) in annotations:
            draw.rectangle([(xmin, ymin), (xmax, ymax)], outline='red')
            draw.text((xmin, ymin), label, fill='red')

        # Display the image with bounding boxes
        ax.imshow(img)
        ax.axis('off')

    plt.tight_layout()
    plt.show()

# Define the paths
base_path = '/content'
images_path = os.path.join(base_path, 'JPEGImages')
annotations_path = os.path.join(base_path, 'Annotations', 'Horizontal Bounding Boxes')

# Call the function to visualize sample images
visualize_sample_images(images_path, annotations_path)
```

Visual inspection of the data was carried out using the **visualize_sample_images()** function. This function displays a subset of the images with their corresponding bounding boxes and labels. Such visualization is crucial to ensure data integrity and understand its structure.



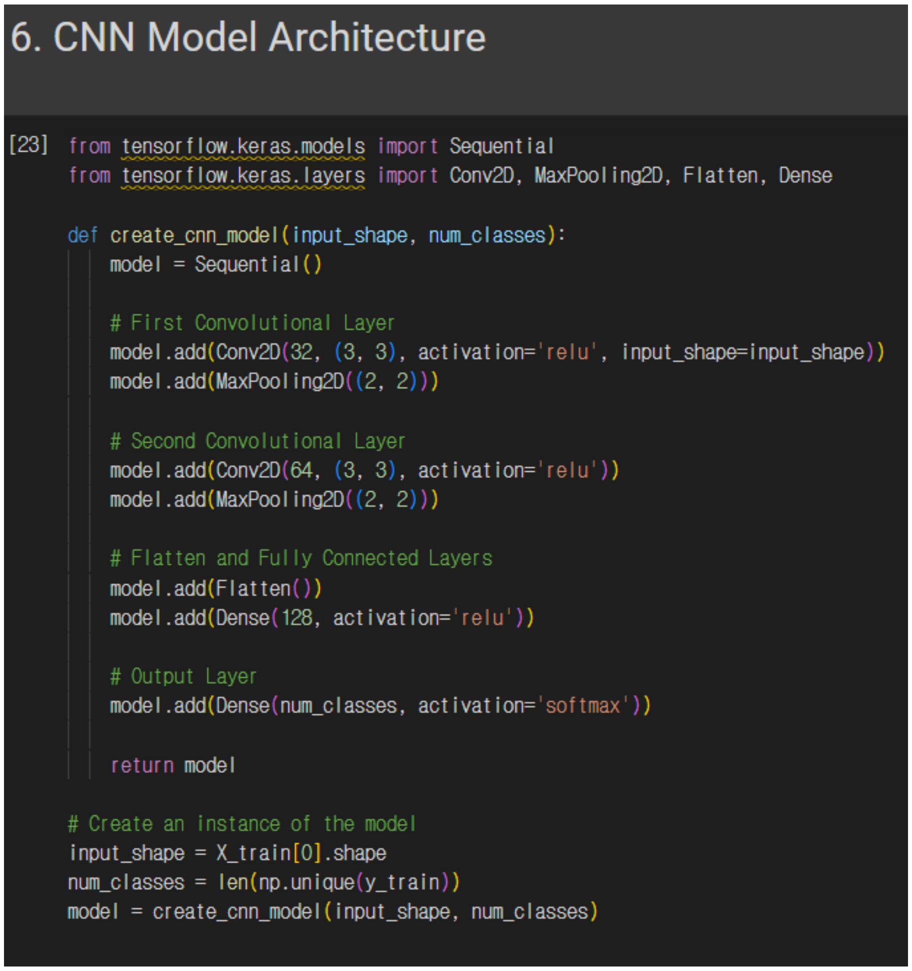
Data Preprocessing


5. Data Preprocessing

```
# Paths
base_path = '/content'
images_path = os.path.join(base_path, 'JPEGImages')
annotations_path = os.path.join(base_path, 'Annotations', 'Horizontal Bounding Boxes')

# List to store all data and labels
all_data = []
all_labels = []

# Define an image size for resizing
IMG_SIZE = (224, 224)

# Process each image
for img_name in os.listdir(images_path):
    img_path = os.path.join(images_path, img_name)
    annotation_path = os.path.join(annotations_path, img_name.split('.')[0] + '.xml')

    # Open the image
    img = Image.open(img_path).convert("RGB")  # Ensure 3 channels with convert("RGB")

    # Get bounding boxes from the XML annotation
    annotations = read_annotations(annotation_path)
    for label, bbox in annotations:
        # Crop the image using the bounding box
        cropped_img = img.crop(bbox)
        cropped_img = cropped_img.resize(IMG_SIZE)  # Resize the cropped image
        all_data.append(np.array(cropped_img))
        all_labels.append(label)

# Convert data and labels to numpy arrays
all_data = np.array(all_data)
all_labels = np.array(all_labels)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(all_data, all_labels, test_size=0.33, random_state=42)
```

- **Image Cropping and Resizing**: Images in the dataset contained multiple objects. For the purpose of this study, each object was treated as a separate data point. Images were cropped based on bounding boxes extracted from XML annotations. These cropped sections were resized to 224×224224×224 pixels using bilinear interpolation. This transformation ensures uniform input size, which is essential for CNNs.
- **Data Splitting**: The **train_test_split** function from scikit-learn was employed. This ensures a random split of data into training and testing sets, using a 67:33 ratio. A fixed **random_state** ensures reproducibility of results.

Model Architecture:



6. CNN Model Architecture

```
[23] from tensorflow.keras.models import Sequential
     from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

     def create_cnn_model(input_shape, num_classes):
         model = Sequential()

         # First Convolutional Layer
         model.add(Conv2D(32, (3, 3), activation='relu', input_shape=input_shape))
         model.add(MaxPooling2D((2, 2)))

         # Second Convolutional Layer
         model.add(Conv2D(64, (3, 3), activation='relu'))
         model.add(MaxPooling2D((2, 2)))

         # Flatten and Fully Connected Layers
         model.add(Flatten())
         model.add(Dense(128, activation='relu'))

         # Output Layer
         model.add(Dense(num_classes, activation='softmax'))

         return model

     # Create an instance of the model
     input_shape = X_train[0].shape
     num_classes = len(np.unique(y_train))
     model = create_cnn_model(input_shape, num_classes)
```

Model Architecture

In this project, a Convolutional Neural Network (CNN) is employed to process and classify the image data. CNNs have shown remarkable results in image recognition tasks, making them a suitable choice for this project.

Convolutional Neural Networks (CNNs)

CNNs are a category of deep neural networks specially designed to process grid-like data such as images (LeCun, Bottou, Bengio, & Haffner, 1998). The primary difference between CNNs and other neural networks is the presence of convolutional layers at the beginning, which apply convolutional filters to the input data. This process enables the network to focus on local patterns or features, making CNNs particularly effective for image data.

The model architecture comprises the following layers:

1. **Convolutional Layer (Conv2D)**: The foundational layer of a CNN where the image is convolved with a set of filters. Each filter detects specific features like edges, textures, or patterns. Mathematically, a convolution involves a filter (a small matrix) that slides over the input data

(such as an image) to produce a feature map, effectively transforming the input data.
$Feature\ Map = Input * Filter$

MaxPooling Layer (MaxPooling2D): Pooling layers reduce the dimensions of the data by downsampling it, retaining only the most significant information. In MaxPooling, for each patch of the feature map, the maximum value is taken. This process reduces computation, controls overfitting, and maintains the detected features' essential characteristics.

Flatten Layer: After feature extraction using convolutional and pooling layers, the two-dimensional feature maps need to be flattened into a one-dimensional vector, making them suitable for input into the fully connected layers.

Flatten Layer: After feature extraction using convolutional and pooling layers, the two-dimensional feature maps need to be flattened into a one-dimensional vector, making them suitable for input into the fully connected layers.

Activation Functions:

- **ReLU (Rectified Linear Unit)**: Used in the convolutional and dense layers, the ReLU activation function transforms each pixel value to the maximum of 0 and the input. Mathematically: $f(x) = max(0, x)$

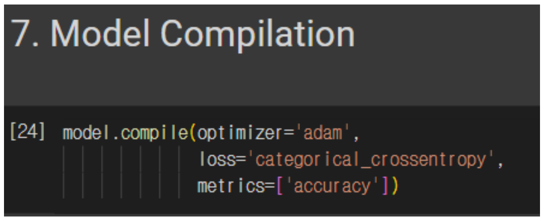
Softmax: Used in the output layer, Softmax calculates the probabilities distribution of the event over 'n' different events. In general way of saying, this function will calculate the probabilities of each target class over all possible target classes, and the range will 0 to 1.

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

$for\ j = 1, \ldots, K$

By stacking these layers in a sequence, the neural network learns to detect features in the early layers and uses these detected features in the deeper layers for classification. The choice and order of these layers, their parameters, and the number of neurons play a crucial role in determining the network's effectiveness in learning patterns from the data.

Model Compilation


7. Model Compilation

```
[24] model.compile(optimizer='adam',
                   loss='categorical_crossentropy',
                   metrics=['accuracy'])
```

Once the architecture of the neural network is defined, the next step is to compile the model. Compilation configures the model for training and prepares it to be optimized.

Key Components of Model Compilation:

1. **Optimizer**: The optimizer is an algorithm that adjusts the weights of the network to minimize the loss function. Different optimizers use various techniques to navigate the weight space effectively.
   - **Adam**: Used in this project, Adam (Adaptive Moment Estimation) is a popular optimization algorithm that combines the strengths of two other optimizers: AdaGrad and RMSProp (Kingma & Ba, 2014). It computes adaptive learning rates for each parameter by considering the moving average of the gradients and the squared gradients. The general idea is to adjust the learning rate during training to converge faster.
2. **Loss Function**: The loss function, or objective function, quantifies how far off our predictions are from the actual values. It provides feedback to the optimizer to adjust the model's weights during training.
   - **Categorical Cross-Entropy**: Used for multi-class classification problems, this loss function calculates the difference between the true labels and the predicted probabilities. It penalizes predictions that are both incorrect and confident. Mathematically, for a single sample, it is defined as:

$$-\sum_{c=1}^{M} y_{o,c} \log(p_{o,c})$$

where $y_{o,c}$ is a binary indicator of whether class c is the correct classification for observation o, and $p_{o,c}$ is the predicted probability observation o is of class c.

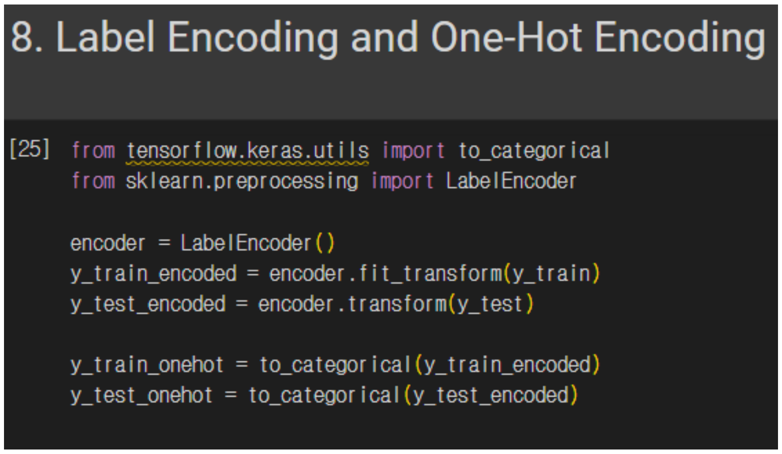
Metrics: Metrics are used to monitor the training and testing steps. They are not used when training the model but are a measure of evaluation.

- **Accuracy**: It's the ratio of the number of correct predictions to the total number of predictions. For multi-class classification, it's often used as the evaluation metric. Given as:

$$Accuracy = \frac{Number\ of\ correct\ predictions}{Total\ predictions\ made}$$

During the compilation process, the neural network internalizes these settings, making it ready for the training phase. The choice of optimizer, loss function, and metrics depends on the nature of the problem and the architecture of the network.

Data Encoding



8. Label Encoding and One-Hot Encoding

```
[25] from tensorflow.keras.utils import to_categorical
     from sklearn.preprocessing import LabelEncoder

     encoder = LabelEncoder()
     y_train_encoded = encoder.fit_transform(y_train)
     y_test_encoded = encoder.transform(y_test)

     y_train_onehot = to_categorical(y_train_encoded)
     y_test_onehot = to_categorical(y_test_encoded)
```

For a neural network to process labels in a classification task, those labels need to be in a format that the network can understand. Often, raw labels are textual or categorical (like 'fighter jet', 'helicopter', etc.). Neural networks, being mathematical constructs, don't understand such textual data directly. They require numerical input. Thus, before feeding labels into the network, they are often encoded or transformed into a numerical format.

Key Components of Data Encoding:

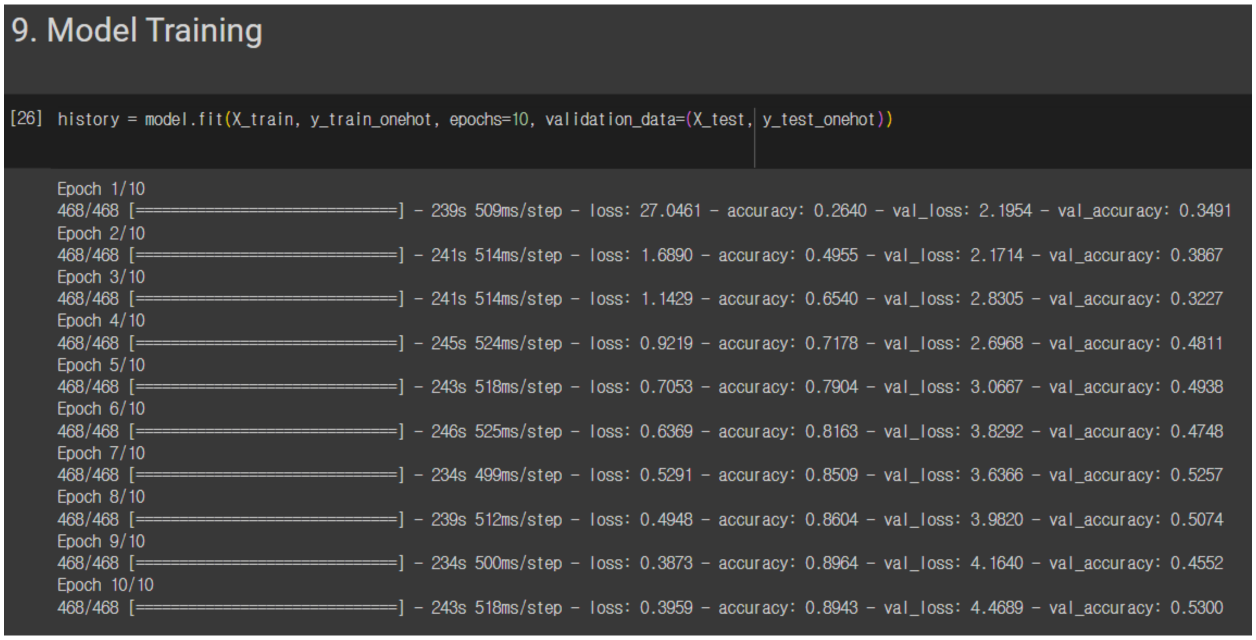
1. **Label Encoding**: Label Encoding involves converting each value in a column to a number. It's a technique to transform non-numerical labels into numerical labels (or nominal categorical variables). For instance, 'fighter jet' might become 0, 'helicopter' might become 1, and so on.
2. **One-Hot Encoding**: One-Hot Encoding is used after label encoding. It involves converting the label encoded data into a binary matrix representation. In this binary matrix, the number of columns corresponds to the number of categories or classes in the data. For a given sample, only the column corresponding to its category will have a 1, and all other columns will have 0s. For example, using our previous example where 'fighter jet' is 0 and 'helicopter' is 1:
    - 'fighter jet' would be represented as [1, 0]
    - 'helicopter' would be represented as [0, 1]

In this project:

- **LabelEncoder** from the **sklearn.preprocessing** module was used to perform label encoding. It's a utility class to help normalize labels such that they contain only values between 0 and **n_classes-1**.
- After label encoding, the labels were further transformed into one-hot encoded format using the **to_categorical** function from TensorFlow's Keras API. This function returns a matrix representation of the input, effectively creating a one-hot encoded version of the input data.

This encoding process is crucial as it prepares the dataset in a format that the neural network can work with, ensuring that the training process is efficient and that the network can make meaningful predictions.

Model Training



9. Model Training

```
[26] history = model.fit(X_train, y_train_onehot, epochs=10, validation_data=(X_test, y_test_onehot))

     Epoch 1/10
     468/468 [==============================] - 239s 509ms/step - loss: 27.0461 - accuracy: 0.2640 - val_loss: 2.1954 - val_accuracy: 0.3491
     Epoch 2/10
     468/468 [==============================] - 241s 514ms/step - loss: 1.6890 - accuracy: 0.4955 - val_loss: 2.1714 - val_accuracy: 0.3867
     Epoch 3/10
     468/468 [==============================] - 241s 514ms/step - loss: 1.1429 - accuracy: 0.6540 - val_loss: 2.8305 - val_accuracy: 0.3227
     Epoch 4/10
     468/468 [==============================] - 245s 524ms/step - loss: 0.9219 - accuracy: 0.7178 - val_loss: 2.6968 - val_accuracy: 0.4811
     Epoch 5/10
     468/468 [==============================] - 243s 518ms/step - loss: 0.7053 - accuracy: 0.7904 - val_loss: 3.0667 - val_accuracy: 0.4938
     Epoch 6/10
     468/468 [==============================] - 246s 525ms/step - loss: 0.6369 - accuracy: 0.8163 - val_loss: 3.8292 - val_accuracy: 0.4748
     Epoch 7/10
     468/468 [==============================] - 234s 499ms/step - loss: 0.5291 - accuracy: 0.8509 - val_loss: 3.6366 - val_accuracy: 0.5257
     Epoch 8/10
     468/468 [==============================] - 239s 512ms/step - loss: 0.4948 - accuracy: 0.8604 - val_loss: 3.9820 - val_accuracy: 0.5074
     Epoch 9/10
     468/468 [==============================] - 234s 500ms/step - loss: 0.3873 - accuracy: 0.8964 - val_loss: 4.1640 - val_accuracy: 0.4552
     Epoch 10/10
     468/468 [==============================] - 243s 518ms/step - loss: 0.3959 - accuracy: 0.8943 - val_loss: 4.4689 - val_accuracy: 0.5300
```

The training process of a neural network involves feeding it data and adjusting the network's weights based on the error of its predictions. This iterative adjustment process is what allows the network to learn from the data.

Key Concepts in Model Training:

1. **Epoch**: An epoch refers to one complete forward and backward pass of all the training examples. The number of epochs is the number of times the learning algorithm will work through the entire training dataset. For example, if I train a model for 10 epochs, it means the entire dataset will be passed forward and backward through the neural network ten times.

2. **Batch Size**: This is the number of training examples used in one iteration. For instance, let's say I have 1000 training samples and I set a batch size of 200. The algorithm would take the first 200 samples from the training dataset and train the network. Next, it would take the second 200 samples and train the network again. I do this until I have propagated all samples through the network.

3. **Loss Function**: It quantifies how well the predicted output of the network matches the actual output. For our classification problem, the categorical cross-entropy loss was used. This is a common choice for classification problems. The loss provides a measure that I aim to minimize during training.

4. **Optimizer**: An optimizer is an algorithm or method used to adjust the attributes of the neural network, such as weights and learning rate, to minimize the loss. In this project, the 'adam'

optimizer was used. Adam is a popular optimization algorithm in deep learning. It combines the best properties of the AdaGrad and RMSProp algorithms to provide an optimization algorithm that can handle sparse gradients on noisy problems.

5. **Validation Data**: While the model is trained using the training data, I also pass it validation data to test its predictions. This data isn't used to adjust the model's weights. Instead, it provides an unbiased evaluation of the model's performance as it trains. It helps in detecting issues like overfitting, where the model performs exceptionally well on the training data but poorly on unseen data.

In this project, the model was trained using the **fit** method, which is a standard Keras function to train the model. The training data (**X_train** and **y_train_onehot**) was passed along with the number of epochs, and the validation data (**X_test** and **y_test_onehot**). The function returns a **history** object, which contains details about the training process, including metrics like loss and accuracy for both the training and validation sets over each epoch. This history is valuable as it allows for the visualization of the training process and helps in diagnosing issues or determining if the model is training as expected.

Evaluation and Visualization

After training, it's crucial to evaluate how well the model has learned to make predictions on unseen data. This evaluation can help understand the model's strengths and weaknesses and guide further refinement.

Key Concepts in Evaluation:

1. **Accuracy**: Accuracy is a metric that quantifies the number of correct predictions made by the model out of all predictions. While it's a straightforward metric, it might not always be the best choice, especially when dealing with imbalanced datasets.

2. **Loss**: The loss, or cost, function is a mathematical function that the model seeks to minimize during training. By examining the loss on the validation data, I can get a sense of how well the model is generalizing to new, unseen data.

3. **Learning Curves**: These are plots that show changes in learning performance over time in terms of experience. By visualizing the model's accuracy and loss for both training and validation data across epochs, I can get insights into overfitting (when the model performs well on the training data but poorly on the validation data) or underfitting (when the model performs poorly on both datasets).

4. **Histogram of Prediction Probabilities**: This visualization provides insights into the confidence of the model's predictions. For a well-trained model, I expect most predictions to be made with high confidence. If many predictions fall in the middle range (e.g., 0.4-0.6 in binary classification), it might indicate that the model is uncertain about those predictions.

5. **Confusion Matrix**: For classification tasks, a confusion matrix provides a summary of prediction results on a classification problem. The number of correct and incorrect predictions is

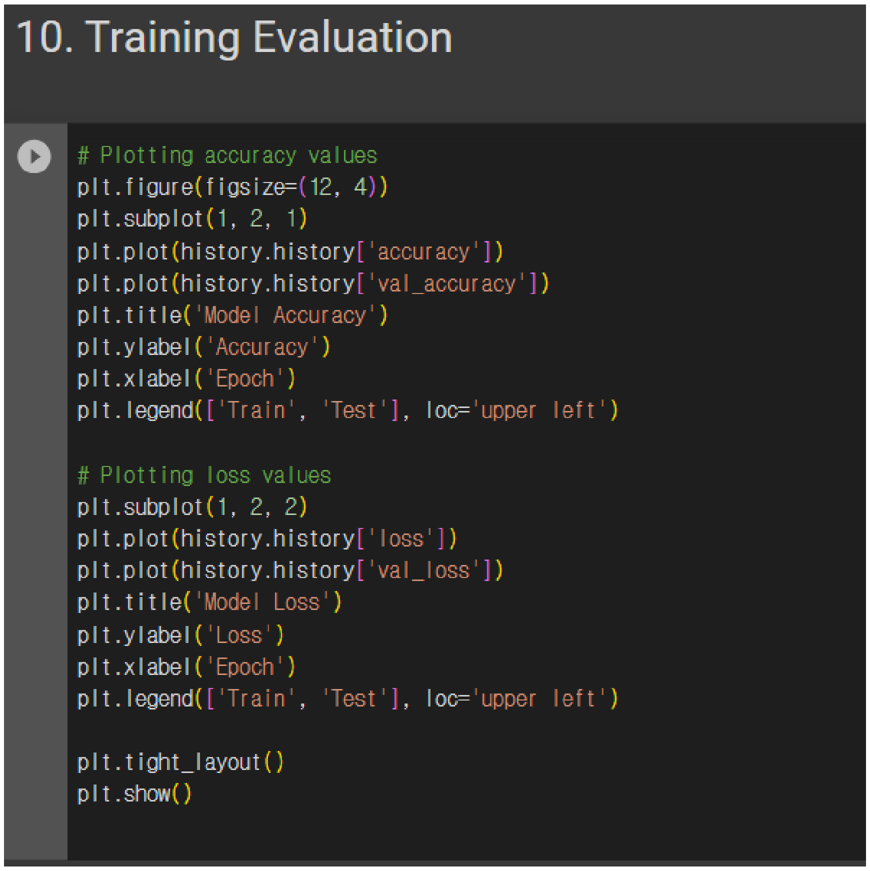
summarized with count values and broken down by each class. It gives insights into which classes are most often confused by the model.

Visualization:

Visualizations play a pivotal role in understanding and interpreting machine learning models. They provide insights into the model's behavior, highlight potential issues, and can guide further refinement.

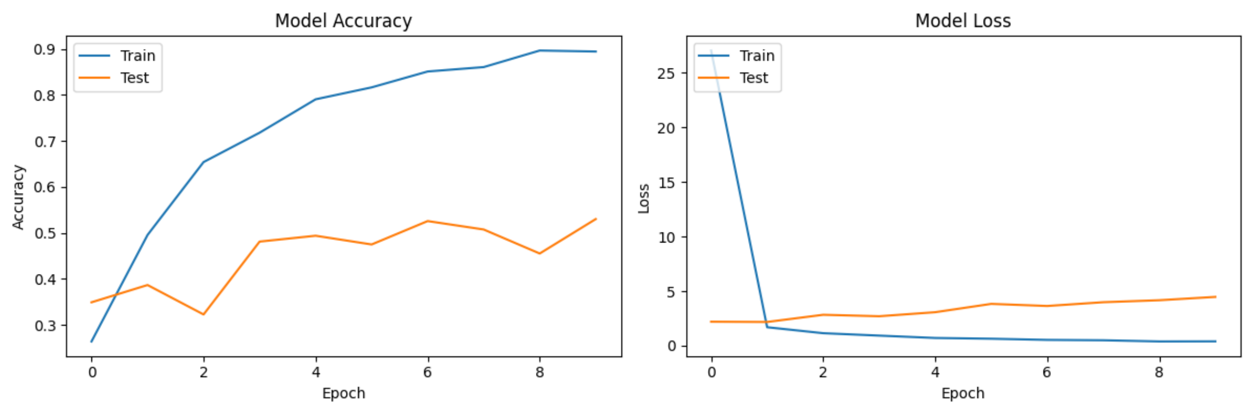
For this project:

- **Learning Curves**: By plotting the accuracy and loss values for both training and validation sets over epochs, I can observe how the model's performance changes over time. If the training loss continues to decrease but the validation loss starts to increase, this could be a sign of overfitting.

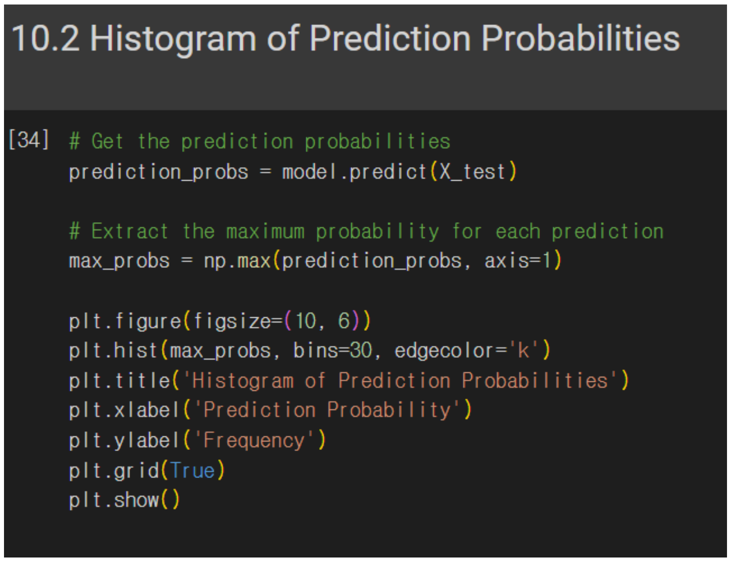


10. Training Evaluation

```python
# Plotting accuracy values
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')

# Plotting loss values
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')

plt.tight_layout()
plt.show()
```

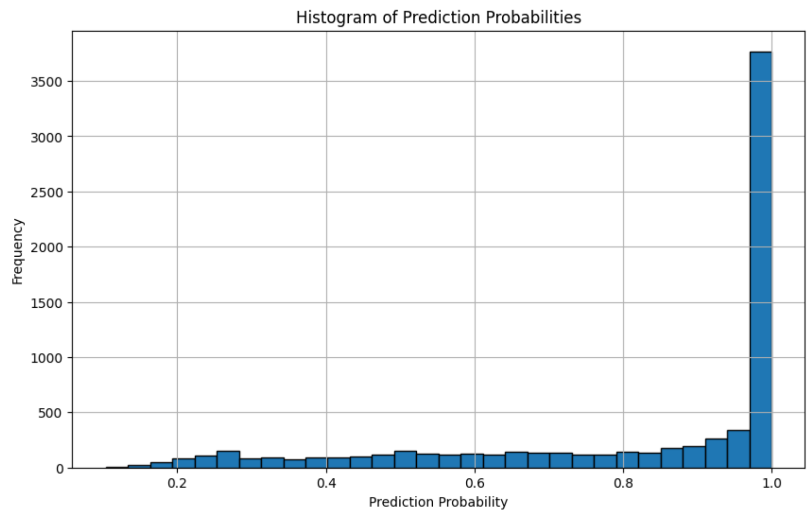


- **Histogram of Prediction Probabilities**: By visualizing the distribution of the model's prediction probabilities, I can gauge its confidence in making predictions. A model that outputs extreme probabilities (close to 0 or 1 for binary classification) is more confident in its predictions than one that outputs probabilities closer to 0.5.



10.2 Histogram of Prediction Probabilities

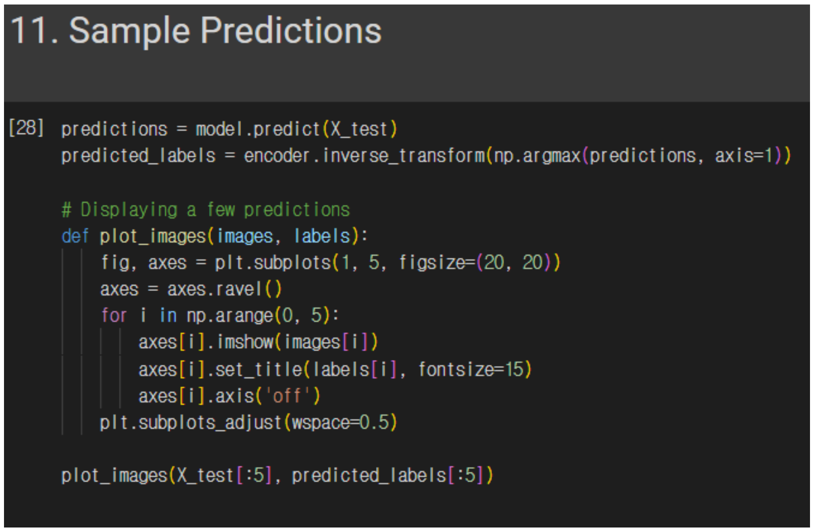
```
[34]  # Get the prediction probabilities
      prediction_probs = model.predict(X_test)

      # Extract the maximum probability for each prediction
      max_probs = np.max(prediction_probs, axis=1)

      plt.figure(figsize=(10, 6))
      plt.hist(max_probs, bins=30, edgecolor='k')
      plt.title('Histogram of Prediction Probabilities')
      plt.xlabel('Prediction Probability')
      plt.ylabel('Frequency')
      plt.grid(True)
      plt.show()
```



s

Including these visualizations in the journal provides a transparent and comprehensive view of the model's performance and behavior, ensuring that the results and conclusions drawn are backed by empirical evidence.

Prediction



11. Sample Predictions

```
[28] predictions = model.predict(X_test)
     predicted_labels = encoder.inverse_transform(np.argmax(predictions, axis=1))

     # Displaying a few predictions
     def plot_images(images, labels):
         fig, axes = plt.subplots(1, 5, figsize=(20, 20))
         axes = axes.ravel()
         for i in np.arange(0, 5):
             axes[i].imshow(images[i])
             axes[i].set_title(labels[i], fontsize=15)
             axes[i].axis('off')
         plt.subplots_adjust(wspace=0.5)

     plot_images(X_test[:5], predicted_labels[:5])
```

Once a model has been trained and evaluated, the next step is to utilize it for making predictions on new, unseen data. This is often the primary goal of building a machine learning model: to make informed decisions based on data it hasn't seen during training.

Prediction

1. **Input Data Preprocessing**: Just as I preprocessed our training data before feeding it into the model, any new data I wish to predict on must undergo the same preprocessing steps. This ensures that the data is in a format the model expects. For image data, this typically involves resizing the images to the expected input size, normalizing the pixel values, and potentially expanding the dimensions to represent a batch.

2. **Model Inference**: This is the actual prediction step. The preprocessed data is fed into the trained model, which outputs its predictions. For classification tasks, this usually takes the form of probabilities for each class.

3. **Post-processing**: Often, the raw output from the model needs some post-processing to be interpretable. In classification tasks, this could involve taking the class with the highest probability as the model's prediction. If label encoders were used during preprocessing, this step would also involve converting numerical predictions back to their original labels.

For this project:

After training the CNN on the dataset of military aircraft images, the model was used to predict the classes of a set of test images. The steps involved:

- Loading and preprocessing the test images.

- Using the **predict** method of the trained model to get the predicted class probabilities for each image.
- Decoding the predicted class labels from the highest probability predictions using the label encoder.
- Visualizing some of the predictions by displaying the test images alongside their predicted labels.

Predictions provide a practical demonstration of the model's capabilities. In real-world applications, this step is crucial as it translates the abstract capabilities of a trained model into actionable insights or decisions. For instance, in a military context, accurately identifying aircraft types from satellite images can aid in strategic decision-making.

Conclusion

This study embarked on an in-depth exploration of Convolutional Neural Networks (CNNs) for image classification. Each phase of the project, from setting up the environment to data preprocessing, model architecture design, and evaluation, brought its own set of challenges and insights.

One of the primary technical challenges faced was the limitation of computational resources on the basic version of Google Colab. The environment frequently ran out of memory, disrupting the workflow. This challenge emphasizes the importance of robust computational infrastructure when dealing with deep learning models and large datasets. Upgrading to Colab Pro proved to be a viable solution, offering enhanced RAM and processing capabilities.

Another point of reflection was the time taken for model training. With a training duration close to 50 minutes, it highlighted the intricacies involved in training a deep CNN and the patience required to achieve optimal results.

Despite these challenges, the project reaffirmed the prowess of CNNs in extracting nuanced features from images and recognizing intricate patterns. It's clear that while CNNs hold immense promise, utilizing them effectively comes with its set of hurdles.

As the field of machine learning continues to evolve and as datasets grow in both size and complexity, the role of advanced models like CNNs becomes even more paramount. This research journey, with its blend of challenges and achievements, offers a window into the ever-evolving landscape of image-based machine learning.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., ... & Ghemawat, S. (2016). TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. arXiv preprint arXiv:1603.04467.

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. Nature, 521(7553), 436-444. https://doi.org/10.1038/nature14539

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278-2324. https://doi.org/10.1109/5.726791

Appendix

Kaggle: Military Aircraft Recognition dataset. (n.d.). Www.kaggle.com. Retrieved October 15, 2023, from

https://www.kaggle.com/datasets/khlaifiabilel/military-aircraft-recognition-dataset

Colab Link: 13659848_Junghyun_Shin_A2.ipynb - Colaboratory (google.com)

Github Link: UTS/ at main · Junghyun-Shin/UTS (github.com)

ChatGPT: OpenAI. (2023). *ChatGPT* (September 25 Version) [Large language model].

https://chat.openai.com

https://chat.openai.com/share/de15db19-aaed-4794-a651-5f21155fc8c3