A Machine Learning Approach to Classifying Billiards Balls

October 9, 2024

1 Introduction

This report is about recognizing billiard balls from images. Our student organization has an interest for software that can analyze billiard games from video footage. Currently, our billiard statistics software only registers the outcomes, without insights into the gameplay itself. A system capable of recognizing the balls on the table could generate various player statistics, such as the success rate of long shots or games lost due to potting the eight ball prematurely. As a first step towards developing this system, this project focuses specifically on labeling individual billiards balls from clear, square-shaped images.

This report is structured as follows: First, the problem is defined, including an overview of the dataset and the machine learning approach. The methodology section details the data preprocessing, model selection, and the training process. This is followed by a discussion of the results, including evaluation metrics and insights gained from the experiments. Finally, conclusions and potential areas for improvement are outlined.

2 Problem Formulation

Our dataset of billiards balls contains 960 high-quality 3552x3552 pixel, 24-bit RGB images of billiards balls, 60 for each of the 16 different categories. Each image only depicts one ball. The data is therefore categorical with 16 different classes, one representing each type of ball.

We have cropped and resized the images to 64 by 64 pixels. We have also scaled the color 8-bit color channels, whose values range from 0 to 256, to floating-point values between 0 and 1. After preprocessing the image dataset, each image will constitute a 3×4096 feature matrix of floating-point values. A machine learning model, specifically a SVM classifier, is trained using this labeled dataset. This means we have a supervised machine learning task, with the goal of accurately predicting the labels of input images.

3 Method

A large literature of image classification projects exist. In this report we implement two approaches for image classification: a support vector machine classifier (SVC) and a convolutional neural network (CNN). For the SVC approach we have chosen to largely follow the method described in [1]. The CNN approach largely follows tensorflow's computer vision tutorial, which we also used for the dataset loading and preprocessing [2].

3.1 Dataset

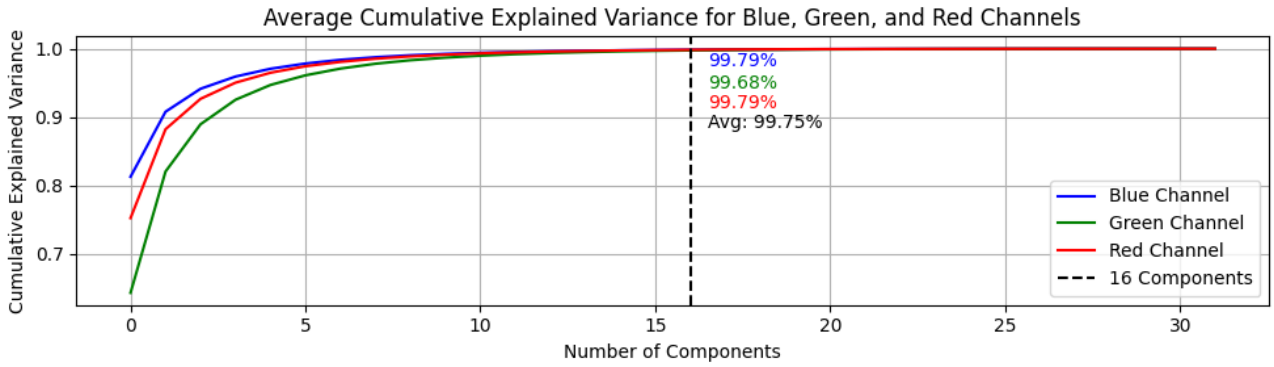
The dataset of billiards ball images was created specifically for the goal of predicting the correct labels of billiards balls at our table location. We took 60 photos of each of the 16 balls from various angles, resulting in 960 total images. The images taken were 3552 by 3552 pixel jpg images with a bit depth of 24. The dataset was split into training, validation and test datasets. The final training dataset had 720 images, validation dataset 80 images and test dataset 160 images, corresponding roughly to a 75-8-17 split. We chose this split to have most of the data be used in training, while setting aside separate datasets for validation to prevent overfitting, and testing to evaluate the final performance of the model. The reasoning for a commonly used 75-10-15 split is explained in [3]. The training, validation and test datasets were randomly separated using scikit-learn's train_test_split-function.

3.2 Data Preprocessing and Feature Engineering

Preprocessing

Our initial dataset consists of unnecessarily large RGB images. Our preprocessing has three steps: Cropping the images, resizing them to 64x64 pixels and scaling the 8-bit integer RGB values down to a single floating-point number between 0 and 1. This is done to reduce the size of inputs to the model from an integer matrix of size $3 \times (3552^2)$, to a more manageable 3×4096 floating-point values. The color value scaling was done because machine learning methods generally work better with small input values. [4]. Now each feature is a 3 by 4096 matrix of floating-point numbers between 0 and 1. The dimensions of the matrix represent the red, green and blue color channels of the image. The features are labeled with a number between 0 and 15, representing the ground truth, which is the number of the billiards ball in the image. The cue ball is number 0.

Principal Component Analysis



With the Support Vector Classifier (explained in section 3.3), the data was additionally preprocessed with Principal Component Analysis (PCA) in order to reduce input feature size [5]. We chose the number of PCA components to be 16, based on our analysis of cumulative explained variance of the number of components for each color channel over the training dataset. 16 PCA components explained 99,75% of variance in the images. This reduced the size of the features by 75% from $64 \times 64 \times 3$ to $64 \times 16 \times 3$. The PCA was implemented using the PCA class of scikit-learn's decomposition module [6].

Data Augmentation

We used data augmentation when training our CNN model (explained in section 3.3). We chose to perform the augmentation since our dataset is comparatively small and not very diverse. We augmented the training dataset by applying random rotations, image flips and brightness adjustments to the preprocessed training dataset images. The augmentation was performed using keras' layers utility [7]. We also found that augmentation greatly reduced model overfitting to the training dataset.

3.3 Model

SVM Classifier

For this task of recognizing billiard balls from images, we have selected the Support Vector Classifier (SVC) as our initial ML model. SVC is ideal for handling high-dimensional data, like images, by finding a hyperplane that maximises the margin between different classes. In this case, the classes are different billiard balls. SVC works by identifying the optimal weight vector and bias term to separate these classes. If the data isn't linearly separable, kernel functions such as the Radial Basis Function (RBF) can be used to map the data to a higher-dimensional space, allowing more accurate classification. [1], [8]

$$h(x) = sign(wx + b) \tag{1}$$

In this case, $h(x)$ assigns a class label to the input data point x, returning +1 when $wx + b$ is greater than or equal to 0, and -1 when it is less than 0.

SVC was chosen because of its ability to efficiently manage image-based tasks with numerous pixel features, and its effectiveness in separating classes through margin maximization. It's also robust against overfitting and performs well with smaller datasets like the couple thousand images used in this project, making it a practical and reliable choice. [1], [8]

Convolutional Neural Network

For this project, we also implemented a Convolutional Neural Network (CNN), another powerful machine learning model designed specifically for image-based tasks. CNNs are highly effective in recognizing patterns within images by taking advantage of their unique architecture, which is well-suited for extracting spatial hierarchies from visual data [9].

A CNN works by passing the input image through several layers, including convolutional layers, pooling layers, and fully connected layers. The convolutional layers apply filters that move across the image to detect local features such as edges or textures. These features are then aggregated in the pooling layers, which reduce the spatial dimensions of the data, preserving important information while minimizing computational complexity. The final fully connected layers perform the actual classification by combining the learned features into predictions for each class [9].

We tested three different CNN models. The first model had three convolutional layers with 16, 32, and 64 filters (3x3 kernels, ReLU activation), each followed by max pooling. The output was flattened and passed through a 128-unit dense layer with ReLU activation, and then a final dense layer with 16 outputs for classification. The second model, model_l2, was identical to the first but applied L2 kernel regularization to the 128-unit dense layer. The third model, model_batch, introduced batch normalization between each layer. We used regularization and normalization to reduce overfitting. As outlined in the results, the three models performed very similarly, all achieving over 95% test accuracy.

3.4 Loss Function

SVM Hinge Loss

For this project, we have selected the hinge loss function as our initial loss function, which is commonly used with Support Vector Machines (SVMs). Hinge loss encourages the model to maximize the margin between different classes, penalizing predictions that fall too close to the decision boundary or are misclassified. This ensures that the SVC model not only classifies the billiard balls correctly but does so with confidence, reducing the likelihood of errors. [1], [10]

$$L(y, f(x)) = max(0, 1 - y \cdot f(x)) \tag{2}$$

Here, y denotes the actual class label (+1 or -1), while $f(x)$ refers to the classifier's decision function for the given data point x.

Hinge loss is computationally efficient and aligns perfectly with SVC's margin-maximizing objective, making it an ideal choice. By penalizing near-boundary predictions, hinge loss ensures that the model maintains clear and accurate boundaries between billiard ball classes, improving overall classification performance. [1], [10]

CNN Sparse Multiclass Cross Entropy

For our Convolutional Neural Network (CNN), we used the Sparse Multiclass Cross-Entropy Loss function to optimize the model's performance in the classification of billiard balls. This loss function is specifically designed for multiclass classification tasks where each image corresponds to exactly one class, making it well-suited for our project of recognizing a single billiard ball from each image [11].

The Sparse Multiclass Cross-Entropy Loss works by comparing the true class label with the predicted probability for that class. The function takes the negative log of the predicted probability assigned to the true class, and then averages this value across the dataset. Mathematically, it is expressed as:

$$L(y_{\text{true}}, y_{\text{pred}}) = -\sum(y_{\text{true}} \times \log(y_{\text{pred}})) \tag{3}$$

Here, y_{true} represents the true label (an integer value between 0 and the number of classes), while y_{pred} is the predicted probability for that class. This formulation ensures that the model is penalized more when it assigns a lower probability to the correct class[11].

One advantage of the sparse version of cross-entropy loss is its computational efficiency, as it only requires the probability for the true class rather than for all possible classes. However, it is somewhat less robust to class imbalance than other loss functions, which could be considered in future iterations of the project. Overall, Sparse Multiclass Cross-Entropy Loss allowed us to efficiently train the CNN while maintaining a high level of prediction accuracy.

4 Results

The results of this project demonstrate the performance of both the Support Vector Classifier (SVC) and Convolutional Neural Network (CNN) in recognizing billiard balls from images. We present the evaluation metrics, including accuracy, training time, and prediction time, for each model configuration.

Support Vector Classifier

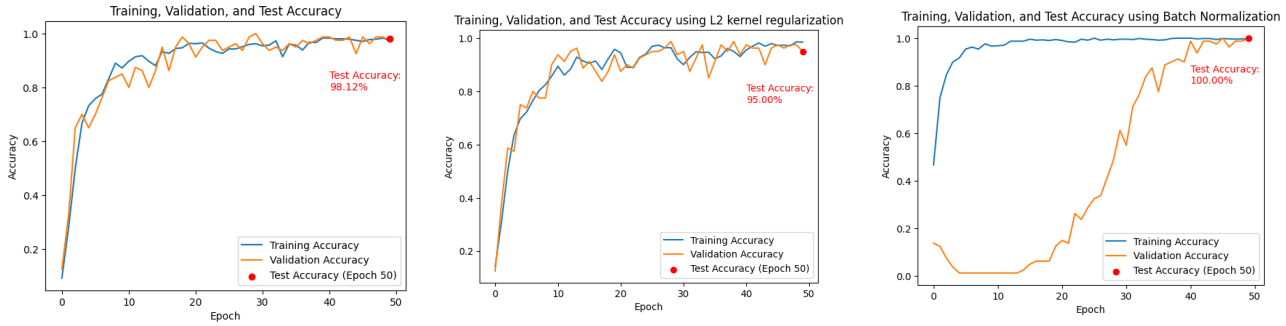
The SVC was evaluated with and without Principal Component Analysis (PCA) to assess the trade-offs between accuracy and computational efficiency. As we can see from the results below, the training time increased by a factor of 26 and prediction time by a factor of 52. While accuracy improves without PCA, the computational cost in terms of time is drastically higher. The decision to use PCA thus depends on the project's priorities, whether focusing on accuracy or efficiency.

Dataset	With PCA (16 components)	Without PCA	Change
Train	91.53%	89.03%	-2.5%
Validation	80.00%	85.00%	+5%
Test	70.00%	81.25%	+11.25%
Training time*	11 seconds	4 minutes 47 seconds	×26
Prediction time*	3 seconds	2 minutes 36 seconds	×52

Table 1: SVC prediction accuracy and computation times with and without PCA.

* Prediction time reported her means the time to predict labels for the testing dataset of 160 images. The training time with PCA includes the time for performing PCA on the training dataset. The reason for the difference in relative computation times with and without PCA between training and prediction is because the prediction time doesn't include the time used to perform PCA on the test dataset.

Convolutional Neural Network



The graph above shows the performance of the three CNN models over their training period of 50 epochs each. Model_batch performed the best, but all models achieved over 95% accuracy. When running the training multiple times, the results weren't stable, but landed somewhere above 95%. Further statistical analysis of the difference in model performance could be done, but we have chosen to not include that in this project. We cannot currently say that there is any significant difference between the three CNN models tested.

5 Conclusion

In this report, we explored two machine learning models—Support Vector Classifier (SVC) and Convolutional Neural Network (CNN)—to classify billiard balls from images. The SVC provided a computationally efficient method for classification, although its accuracy was lower compared to the CNN models. The CNN models, in contrast, achieved higher classification accuracy, surpassing 95% on the test dataset.

Despite the promising results, there are areas for improvement. The relatively small size and lack of diversity in the dataset limited the models' generalizability. Future work could explore expanding the dataset with additional variations, such as different lighting conditions and more complex backgrounds. The scope of the machine learning problem could also be widened to include image segmentation and object detection to find coordinates of multiple billiards balls from an image.

Overall, this project successfully demonstrated the potential of machine learning for automating the recognition of billiard balls. The work presented here can serve as a first step toward more advanced applications.

6 Aknowledgements

Use of generative AI

Parts of this text were generated using ChatGPT, but all prompts were crafted and texts edited and reviewed the authors. Additionally, ChatGPT was used for assistance in data processing and coding.

References

[1] unknown, "A Machine Learning Approach to Classifying Bangla Handwritten Characters," Sep. 2023.

[2] "Computer vision with TensorFlow — TensorFlow Core," Accessed: Sep. 20, 2024. [Online]. Available: https://www.tensorflow.org/tutorials/images.

[3] V. R. Joseph, "Optimal Ratio for Data Splitting," *Statistical Analysis and Data Mining: The ASA Data Science Journal*, vol. 15, no. 4, pp. 531–538, Aug. 2022, ISSN: 1932-1864, 1932-1872. DOI: 10.1002/sam.11583. arXiv: 2202.03326 [cs, stat]. Accessed: Sep. 19, 2024. [Online]. Available: http://arxiv.org/abs/2202.03326.

[4] "Importance of Feature Scaling," scikit-learn, Accessed: Sep. 20, 2024. [Online]. Available: https://scikit-learn/stable/auto_examples/preprocessing/plot_scaling_importance.html.

[5] S. Wold and K. E. P. Geladi, "Principal Component Analysis,"

[6] "PCA," scikit-learn, Accessed: Oct. 9, 2024. [Online]. Available: https://scikit-learn/stable/modules/generated/sklearn.decomposition.PCA.html.

[7] "Module: Tf.keras.layers — TensorFlow v2.16.1," TensorFlow, Accessed: Oct. 9, 2024. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/layers.

[8] W. S. Noble, "What is a support vector machine?" *Nature Biotechnology*, vol. 24, no. 12, pp. 1565–1567, Dec. 2006, ISSN: 1087-0156, 1546-1696. DOI: 10.1038/nbt1206-1565. Accessed: Sep. 20, 2024. [Online]. Available: https://www.nature.com/articles/nbt1206-1565.

[9] K. O'Shea and R. Nash. "An Introduction to Convolutional Neural Networks." arXiv: 1511.08458 [cs], Accessed: Oct. 8, 2024. [Online]. Available: http://arxiv.org/abs/1511.08458, pre-published.

[10] P. L. Bartlett and M. H. Wegkamp, "Classification with a Reject Option using a Hinge Loss," Aug. 2008.

[11] C. Jeeva. "Loss Functions in Neural Networks," Scaler Topics, Accessed: Oct. 8, 2024. [Online]. Available: https://www.scaler.com/topics/loss-functions-in-neural-networks/.

A The python code used in the project

ml

October 9, 2024

[ ]:
```
!pip install Pillow opencv-python numpy tensorflow
```

1 Crop and resize images

The images are 3552x3552 pixels and contain a lot of empty space at the edges. Here we crop the images to 2048x2048 toward the center. Then they are resized to 64x64.

[ ]:
```
from PIL import Image
import os
from pprint import pprint

dirs = [ (f'.\\data\\raw_data\\{num}', f'.\\data\\processed_64\\{num}') for num␣
 ↪in range(16) ]

pprint(dirs)
for (in_dir, out_dir) in dirs:
    os.makedirs(out_dir, exist_ok=True)

original_size = 3552
crop_size = 2048
target_size = 64
max_files = 60 # We want the same number of images of each ball
```

[ ]:
```
left = (original_size - crop_size) // 2
top = (original_size - crop_size) // 2
right = (original_size + crop_size) // 2
bottom = (original_size + crop_size) // 2

for (in_dir, out_dir) in dirs:
    files = os.listdir(in_dir)
    print(f"Processing {in_dir} ...")

    counter = 0
    for filename in files[:max_files]:
        if filename.lower().endswith(".jpg"):
            counter += 1
            # Open the image
```

```
            img_path = os.path.join(in_dir, filename)
            output_path = os.path.join(out_dir, filename)

            Image.open(img_path
                ).crop((left, top, right, bottom)
                ).resize((target_size, target_size), Image.Resampling.LANCZOS
                ).save(output_path)

            # print(f"Cropped and saved: {output_path} ({counter} of␣
 ↪{max_files})")
```

2 Split the data into training, validation and testing datasets

We have 960 images (60 of each class). We will split these into

720 training images (75 %)
80 validation images (8.33 %)
160 testing images (16.67 %)

[1]:
```
import numpy as np
import keras
from sklearn.model_selection import train_test_split
import tensorflow as tf

data_dir = './data/processed_64'
batch_size = 25
img_width = 64
img_height = 64

dataset = keras.utils.image_dataset_from_directory(
    data_dir,
    image_size=(img_height, img_width),
    batch_size=batch_size,
    label_mode='int'
)

class_names = dataset.class_names # ['0', '1', '10', '11', '12', '13', '14',␣
 ↪'15', '2', '3', '4', '5', '6', '7', '8', '9']

image_batches = []
label_batches = []

for images, labels in dataset:
    image_batches.append(images)
    label_batches.append(labels)

X = np.concatenate(image_batches)
```

```
y = np.concatenate(label_batches)

print("Splitting data into training, validation and testing datasets...")

X_train, X_rest, y_train, y_rest = train_test_split(X, y, test_size = 0.25,␣
 ↪random_state = 0)

X_val, X_test, y_val, y_test = train_test_split(X_rest, y_rest, test_size = (2/
 ↪3), random_state = 0)

print("X_train", X_train.shape)
print("y_train", y_train.shape)
print("X_val", X_val.shape)
print("y_val", y_val.shape)
print("X_test", X_test.shape)
print("y_test", y_test.shape)
```

```
Found 960 files belonging to 16 classes.
Splitting data into training, validation and testing datasets…
X_train (720, 64, 64, 3)
y_train (720,)
X_val (80, 64, 64, 3)
y_val (80,)
X_test (160, 64, 64, 3)
y_test (160,)
```

3 PCA decomposition

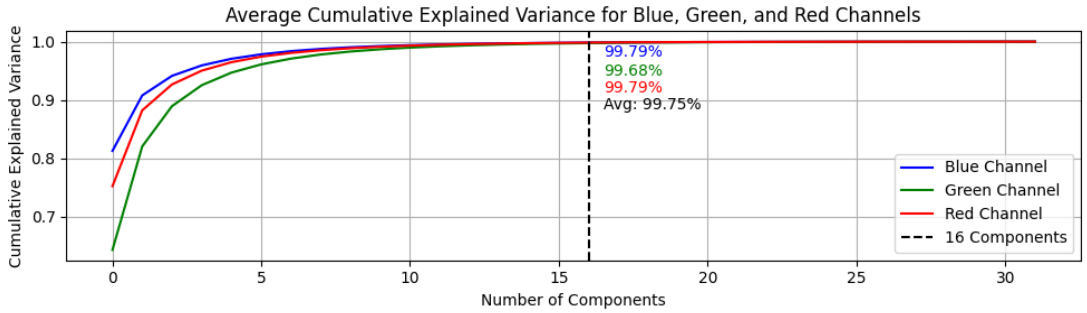
We use PCA decomposition to decrease the number of features. First we calculate the cumulative explained variance of PCA components for each of the image color channels, averaged over the training dataset. We choose the number of PCA components based on the average to preserve as much of the explained variance as possible, while reducing the number of components.

[2]:
```
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import cv2
import numpy as np

# Initialize variables for cumulative explained variance ratios
embed_size = 32
cumulative_variances = [np.zeros(embed_size),np.zeros(embed_size),np.
 ↪zeros(embed_size)]

# Loop over all images in the X_train dataset
counter = 0
for img in X_train:
    # print(f"Processing {counter} of {len(X_train)}")
```

```
    counter = counter + 1

    img = img.astype("uint8")
    blue, green, red = cv2.split(img)
    blue = blue / 255
    green = green / 255
    red = red / 255

    pca_b, pca_g, pca_r = [PCA(n_components=embed_size) for i in range(3)]

    reduced_blue = pca_b.fit(blue)
    reduced_green = pca_g.fit(green)
    reduced_red = pca_r.fit(red)

    cumulative_variances[0] += pca_b.explained_variance_ratio_
    cumulative_variances[1] += pca_g.explained_variance_ratio_
    cumulative_variances[2] += pca_r.explained_variance_ratio_

# Average the explained variance ratios over the entire dataset
avg_variance_blue = cumulative_variances[0] / len(X_train)
avg_variance_green = cumulative_variances[1] / len(X_train)
avg_variance_red = cumulative_variances[2] / len(X_train)
```

[9]:
```
# Calculate the total explained variance for the first 16 components
selected_embed_size = 16

# Find the variance explained by the first 16 components for each channel
variance_blue = np.sum(avg_variance_blue[:selected_embed_size])
variance_green = np.sum(avg_variance_green[:selected_embed_size])
variance_red = np.sum(avg_variance_red[:selected_embed_size])

# Convert to percentage
variance_blue_percentage = variance_blue * 100
variance_green_percentage = variance_green * 100
variance_red_percentage = variance_red * 100
variance_avg_percentage = ((variance_blue + variance_green + variance_red) / 3)␣
 ↪* 100

# Plot the cumulative explained variance for all channels on the same plot
plt.figure(figsize=(10, 3))
plt.title("Average Cumulative Explained Variance for Blue, Green, and Red␣
 ↪Channels")
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance')

# Plot the averaged cumulative explained variance for each channel
plt.plot(np.cumsum(avg_variance_blue), label='Blue Channel', color='blue')
```

```
plt.plot(np.cumsum(avg_variance_green), label='Green Channel', color='green')
plt.plot(np.cumsum(avg_variance_red), label='Red Channel', color='red')

# Add a vertical line at 16 components
plt.axvline(x=16, color='black', linestyle='--', label='16 Components')

# Annotate the variances at x=16
plt.text(16 + 0.5, np.cumsum(avg_variance_blue)[15] - 0.03,␣
 ↪f'{variance_blue_percentage:.2f}%',
         color='blue', fontsize=10, verticalalignment='bottom')
plt.text(16 + 0.5, np.cumsum(avg_variance_green)[15] - 0.06,␣
 ↪f'{variance_green_percentage:.2f}%',
         color='green', fontsize=10, verticalalignment='bottom')
plt.text(16 + 0.5, np.cumsum(avg_variance_red)[15] - 0.09,␣
 ↪f'{variance_red_percentage:.2f}%',
         color='red', fontsize=10, verticalalignment='bottom')
plt.text(16 + 0.5, np.cumsum(avg_variance_red)[15] - 0.12, f'Avg:␣
 ↪{variance_avg_percentage:.2f}%',
         color='black', fontsize=10, verticalalignment='bottom')

# Add a legend
plt.legend(loc='lower right')

# Show the plot
plt.grid(True)
plt.tight_layout()
plt.show()
```



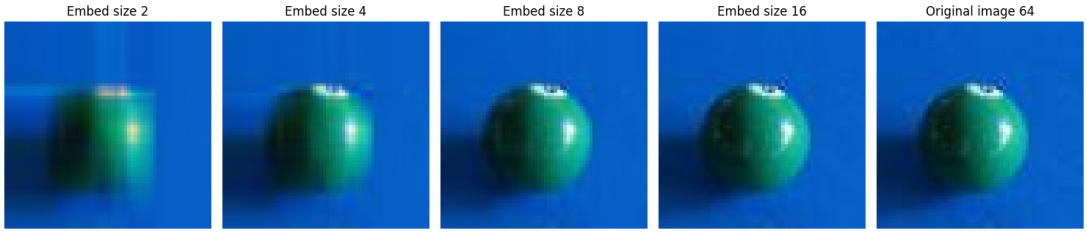
From the plot above we see that 16 PCA components should be sufficient for our use case. Here we visualize how different number of PCA components reduce the image quality.

[49]:
```
from random import randrange
embed_sizes=[2, 4, 8, 16]

fig, axs = plt.subplots(1, len(embed_sizes) + 1, figsize=(15, 15))
```

```
image = X_train[randrange(0, len(X_train))]

for i, selected_embed_size in enumerate(embed_sizes):
    # Select a random image from X_train
    img = image.astype("uint8")
    blue, green, red = cv2.split(img)
    blue = blue / 255
    green = green / 255
    red = red / 255

    # PCA for each channel
    pca_b = PCA(n_components=selected_embed_size)
    trans_pca_b = pca_b.fit(blue).transform(blue)

    pca_g = PCA(n_components=selected_embed_size)
    trans_pca_g = pca_g.fit(green).transform(green)

    pca_r = PCA(n_components=selected_embed_size)
    trans_pca_r = pca_r.fit(red).transform(red)

    # Inverse transformation to get reduced images
    b_arr = pca_b.inverse_transform(trans_pca_b)
    g_arr = pca_g.inverse_transform(trans_pca_g)
    r_arr = pca_r.inverse_transform(trans_pca_r)

    img_reduced = cv2.merge((b_arr, g_arr, r_arr))

    # Plot original and reduced images
    axs[i].imshow(img_reduced)
    axs[i].set_title(f"Embed size {selected_embed_size}")
    axs[i].axis('off')  # Turn off axis

axs[4].imshow(image.astype("uint8"))
axs[4].set_title(f"Original image 64")
axs[4].axis('off')  # Turn off axis

# Show the plots
plt.tight_layout()
plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
```

floats or [0..255] for integers).



4 SVC

[62]:
```
from sklearn.svm import SVC
from sklearn.decomposition import PCA
selected_embed_size = 16

def prepare_X(X, n_components=selected_embed_size, flatten = True):
    n_samples, h, w, channels = X.shape
    # 720, 64, 64, 3

    # Split into color channels
    blue_channel = X[:, :, :, 0]
    green_channel = X[:, :, :, 1]
    red_channel = X[:, :, :, 2]

    print("b, g, r shapes", np.shape(blue_channel), np.shape(green_channel), np.
 ↪shape(red_channel))

    pca_b, pca_g, pca_r = [PCA(n_components=n_components) for _ in range(3)]

    transformed_blue = [pca_b.fit_transform(im) for im in blue_channel]
    transformed_green = [pca_g.fit_transform(im) for im in green_channel]
    transformed_red = [pca_r.fit_transform(im) for im in red_channel]
    # Shape is now (720, 64, 16) for each channel

    print("transformed shapes", np.shape(transformed_blue), np.
 ↪shape(transformed_green), np.shape(transformed_red))

    X_pca = np.stack((transformed_blue, transformed_green, transformed_red),␣
 ↪axis=-1)  # Shape: (720, 64, 16, 3)
    print("xpca_shape", X_pca.shape)

    if (flatten):   return X_pca.reshape(n_samples, -1)
    else:           return X_pca
```

[ ]:
```
print("preparing X_train", X_train.shape)
X_train_pca = prepare_X(X_train)
print(X_train_pca.shape)

print("preparing X_val", X_val.shape)
X_val_pca = prepare_X(X_val)
print(X_val_pca.shape)

print("preparing X_test", X_test.shape)
X_test_pca = prepare_X(X_test)
print(X_test_pca.shape)

model = SVC(probability=True).fit(X_train_pca , y_train)
```

[6]:
```
from sklearn.svm import SVC

X_tr = X_train.reshape(720, -1)
X_va = X_val.reshape(80, -1)
X_te = X_test.reshape(160, -1)

model = SVC(probability=True).fit(X_tr, y_train)
```

[15]:
```
from sklearn import metrics

# X_train_pca = X_tr
# X_val_pca = X_va
# X_test_pca = X_te

y_pred_train = model.predict(X_train_pca)
y_pred_train_probability = model.predict_proba(X_train_pca)

y_pred_val = model.predict(X_val_pca)
y_pred_val_probability = model.predict_proba(X_val_pca)

y_pred_test = model.predict(X_test_pca)
y_pred_test_probability = model.predict_proba(X_test_pca)

train_log_loss = metrics.log_loss(y_train, y_pred_train_probability)
train_hinge_loss = metrics.hinge_loss(y_train, y_pred_train_probability)
train_acc = metrics.accuracy_score(y_train, y_pred_train)
train_pr = metrics.precision_score(y_train, y_pred_train, average='macro')
train_re = metrics.recall_score(y_train, y_pred_train, average='macro')
train_f1 = metrics.f1_score(y_train, y_pred_train, average='macro')

valid_log_loss = metrics.log_loss(y_val, y_pred_val_probability)
valid_hinge_loss = metrics.hinge_loss(y_val, y_pred_val_probability)
valid_acc = metrics.accuracy_score(y_val, y_pred_val)
```

```
valid_pr = metrics.precision_score(y_val, y_pred_val, average='macro')
valid_re = metrics.recall_score(y_val, y_pred_val, average='macro')
valid_f1 = metrics.f1_score(y_val, y_pred_val, average='macro')

test_log_loss = metrics.log_loss(y_test, y_pred_test_probability)
test_hinge_loss = metrics.hinge_loss(y_test, y_pred_test_probability)
test_acc = metrics.accuracy_score(y_test, y_pred_test)
test_pr = metrics.precision_score(y_test, y_pred_test, average='macro')
test_re = metrics.recall_score(y_test, y_pred_test, average='macro')
test_f1 = metrics.f1_score(y_test, y_pred_test, average='macro')
```

[ ]:
```
print(f"Train Log Loss = {train_log_loss:.4f}")
print(f"Train Hinge Loss = {train_hinge_loss:.4f}")
print(f"Train Accuracy = {train_acc * 100:.2f}%")
print(f"Train Precision = {train_pr:.4f}")
print(f"Train Recall = {train_re:.4f}")
print(f"Train F1-Score = {train_f1:.4f}")
print('-' * 20)

print(f"Valid Log Loss = {valid_log_loss:.4f}")
print(f"Valid Hinge Loss = {valid_hinge_loss:.4f}")
print(f"Valid Accuracy = {valid_acc * 100:.2f}%")
print(f"Valid Precision = {valid_pr:.4f}")
print(f"Valid Recall = {valid_re:.4f}")
print(f"Valid F1-Score = {valid_f1:.4f}")
print('-' * 20)

print(f"Test Log Loss = {test_log_loss:.4f}")
print(f"Test Hinge Loss = {test_hinge_loss:.4f}")
print(f"Test Accuracy = {test_acc * 100:.2f}%")
print(f"Test Precision = {test_pr:.4f}")
print(f"Test Recall = {test_re:.4f}")
print(f"Test F1-Score = {test_f1:.4f}")
```

5 SVC with PCA results

Metric	Train	Valid	Test
Log Loss	0.5153	0.6615	0.7971
Hinge Loss	0.4882	0.5621	0.6451
Accuracy	91.53%	80.00%	70.00%
Precision	0.9252	0.8156	0.7180
Recall	0.9155	0.7646	0.7040
F1-Score	0.9169	0.7684	0.6985

pca preparation of datasets + model training time: 11 s prediction time: 3 s

6 SVC without PCA results

Metric	Train	Valid	Test
Log Loss	0.4375	0.5468	0.5665
Hinge Loss	0.4456	0.5346	0.5264
Accuracy	89.03%	85.00%	81.25%
Precision	0.8966	0.8337	0.8205
Recall	0.8909	0.8262	0.8086
F1-Score	0.8887	0.8071	0.7997

model training time: 4 min 47 s prediction time 2 min 36 s

7 SVC with vs. without PCA: Performance Summary

7.1 Accuracy Comparison

Dataset	With PCA	Without PCA	Change
Train	91.53%	89.03%	-2.50%
Validation	80.00%	85.00%	+5.00%
Test	70.00%	81.25%	+11.25%

- **Train Accuracy** drops by 2.5% without PCA.
- **Validation Accuracy** improves by 5% without PCA.
- **Test Accuracy** shows the most significant improvement, increasing by 11.25% without PCA.

7.2 Training and Prediction Time

Task	With PCA	Without PCA	Difference
Training Time	11 seconds	4 min 47 sec	~26x longer
Prediction Time	3 seconds	2 min 36 sec	~52x longer

- **Training Time** without PCA takes over 26 times longer.
- **Prediction Time** without PCA is over 50 times longer than with PCA.

7.3 Conclusion

While the **accuracy improves** significantly when **PCA is not used**, the **computational cost** in terms of training and prediction time is much higher. The decision to use PCA depends on whether the focus is on achieving higher accuracy or on minimizing training and prediction times.

8 Data augmentation

[2]:
```
from keras import layers
import tensorflow as tf
import numpy as np

train_ds = tf.data.Dataset.from_tensor_slices((X_train, y_train))
val_ds = tf.data.Dataset.from_tensor_slices((X_val, y_val))
test_ds = tf.data.Dataset.from_tensor_slices((X_test, y_test))

normalization_layer = keras.layers.Rescaling(1./255) # Rescale RGB values from␣
 ↪0..255 to floats in 0..1

augmentation_layer = keras.Sequential([
    layers.RandomFlip("horizontal_and_vertical"),
    layers.RandomRotation(0.2),
    layers.RandomBrightness(0.05)
])

batch_size = 50
AUTOTUNE = tf.data.AUTOTUNE

train_ds = train_ds.map(
    lambda x, y: (augmentation_layer(x), y)).map(
    lambda x, y: (normalization_layer(x), y)
    ).shuffle(buffer_size=len(X_train), seed=0, reshuffle_each_iteration=True
    ).batch(batch_size
    ).prefetch(buffer_size=AUTOTUNE)

val_ds = val_ds.map(
    lambda x, y: (normalization_layer(x), y)
    ).batch(batch_size
    ).prefetch(buffer_size=AUTOTUNE)

test_ds = test_ds.map(
    lambda x, y: (normalization_layer(x), y)
    ).batch(batch_size
    ).prefetch(buffer_size=AUTOTUNE)
```

9 Create and train the CNN

[6]:
```
num_classes = 16
model = keras.Sequential([
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(32, 3, padding='same', activation='relu'),
```

```
  layers.MaxPooling2D(),
  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(num_classes)
])

model.compile(optimizer="adam", loss=keras.losses.
  ↪SparseCategoricalCrossentropy(from_logits=True), metrics=["accuracy"])
```

[7]:
```
epochs = 50
history = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs = epochs
)
```

```
Epoch 1/50
15/15             2s 23ms/step -
accuracy: 0.1235 - loss: 2.7284 - val_accuracy: 0.1375 - val_loss: 2.5143
Epoch 2/50
15/15             1s 17ms/step -
accuracy: 0.3716 - loss: 2.1513 - val_accuracy: 0.3875 - val_loss: 1.7844
Epoch 3/50
15/15             1s 18ms/step -
accuracy: 0.5547 - loss: 1.3331 - val_accuracy: 0.7250 - val_loss: 1.0101
Epoch 4/50
15/15             1s 16ms/step -
accuracy: 0.6712 - loss: 0.8971 - val_accuracy: 0.6625 - val_loss: 0.9120
Epoch 5/50
15/15             1s 15ms/step -
accuracy: 0.7569 - loss: 0.6954 - val_accuracy: 0.8500 - val_loss: 0.5979
Epoch 6/50
15/15             1s 15ms/step -
accuracy: 0.8077 - loss: 0.5322 - val_accuracy: 0.8000 - val_loss: 0.5852
Epoch 7/50
15/15             1s 16ms/step -
accuracy: 0.8047 - loss: 0.4973 - val_accuracy: 0.8250 - val_loss: 0.4920
Epoch 8/50
15/15             1s 15ms/step -
accuracy: 0.8697 - loss: 0.3993 - val_accuracy: 0.9000 - val_loss: 0.3261
Epoch 9/50
15/15             1s 16ms/step -
accuracy: 0.9077 - loss: 0.2981 - val_accuracy: 0.8625 - val_loss: 0.3368
Epoch 10/50
15/15             1s 16ms/step -
accuracy: 0.9073 - loss: 0.2605 - val_accuracy: 0.9250 - val_loss: 0.2660
```

```
Epoch 11/50
15/15             1s 16ms/step -
accuracy: 0.9087 - loss: 0.2378 - val_accuracy: 0.9000 - val_loss: 0.2759
Epoch 12/50
15/15             1s 16ms/step -
accuracy: 0.8918 - loss: 0.2826 - val_accuracy: 0.8875 - val_loss: 0.2271
Epoch 13/50
15/15             1s 17ms/step -
accuracy: 0.9202 - loss: 0.2364 - val_accuracy: 0.8750 - val_loss: 0.3095
Epoch 14/50
15/15             1s 19ms/step -
accuracy: 0.9038 - loss: 0.2607 - val_accuracy: 0.8875 - val_loss: 0.2538
Epoch 15/50
15/15             1s 17ms/step -
accuracy: 0.9191 - loss: 0.2420 - val_accuracy: 0.8750 - val_loss: 0.3462
Epoch 16/50
15/15             1s 16ms/step -
accuracy: 0.9361 - loss: 0.1854 - val_accuracy: 0.8875 - val_loss: 0.2552
Epoch 17/50
15/15             1s 16ms/step -
accuracy: 0.9601 - loss: 0.1421 - val_accuracy: 0.9875 - val_loss: 0.1096
Epoch 18/50
15/15             1s 16ms/step -
accuracy: 0.9708 - loss: 0.0973 - val_accuracy: 0.9875 - val_loss: 0.0775
Epoch 19/50
15/15             1s 16ms/step -
accuracy: 0.9479 - loss: 0.1578 - val_accuracy: 0.8375 - val_loss: 0.2600
Epoch 20/50
15/15             1s 16ms/step -
accuracy: 0.8780 - loss: 0.3725 - val_accuracy: 0.9625 - val_loss: 0.1470
Epoch 21/50
15/15             1s 16ms/step -
accuracy: 0.9631 - loss: 0.1285 - val_accuracy: 0.9125 - val_loss: 0.1672
Epoch 22/50
15/15             1s 17ms/step -
accuracy: 0.9832 - loss: 0.0824 - val_accuracy: 0.9875 - val_loss: 0.0818
Epoch 23/50
15/15             1s 17ms/step -
accuracy: 0.9670 - loss: 0.1034 - val_accuracy: 0.9375 - val_loss: 0.1399
Epoch 24/50
15/15             1s 16ms/step -
accuracy: 0.9773 - loss: 0.0997 - val_accuracy: 0.9875 - val_loss: 0.0540
Epoch 25/50
15/15             1s 17ms/step -
accuracy: 0.9604 - loss: 0.1005 - val_accuracy: 0.9750 - val_loss: 0.0653
Epoch 26/50
15/15             1s 16ms/step -
accuracy: 0.9837 - loss: 0.0661 - val_accuracy: 1.0000 - val_loss: 0.0387
```

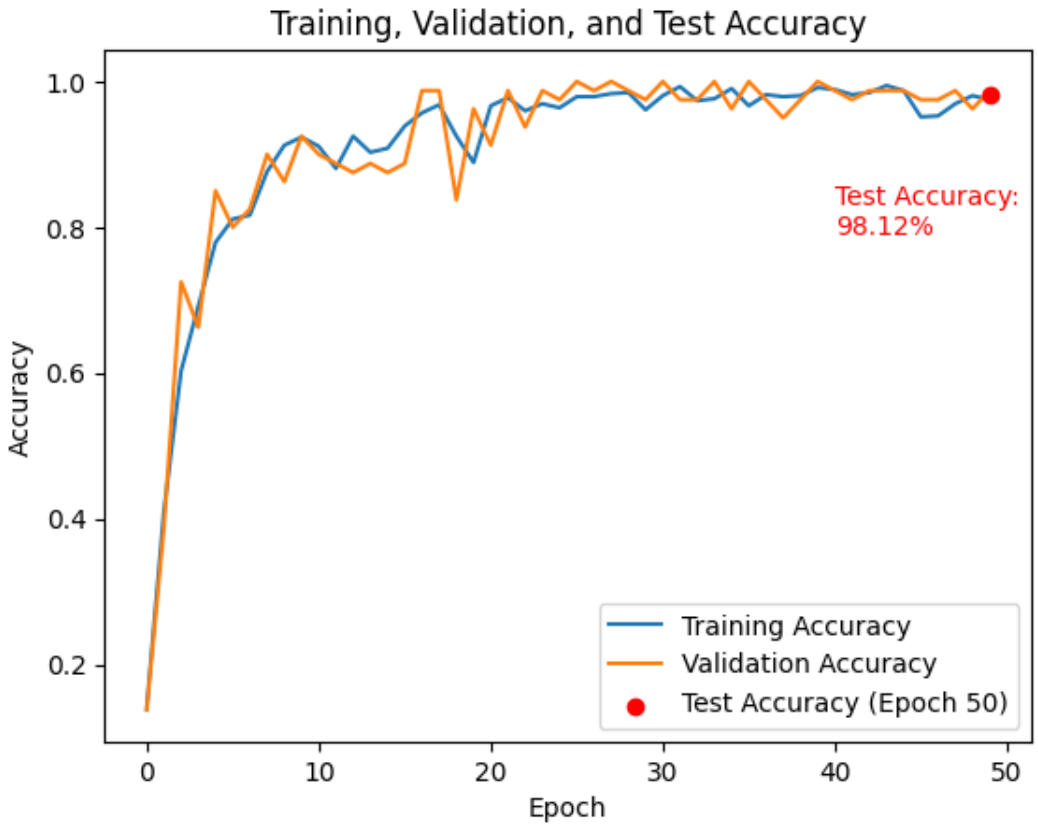
```
Epoch 27/50
15/15             1s 16ms/step -
accuracy: 0.9727 - loss: 0.0715 - val_accuracy: 0.9875 - val_loss: 0.0470
Epoch 28/50
15/15             1s 17ms/step -
accuracy: 0.9892 - loss: 0.0473 - val_accuracy: 1.0000 - val_loss: 0.0238
Epoch 29/50
15/15             1s 16ms/step -
accuracy: 0.9847 - loss: 0.0437 - val_accuracy: 0.9875 - val_loss: 0.0559
Epoch 30/50
15/15             1s 17ms/step -
accuracy: 0.9532 - loss: 0.1160 - val_accuracy: 0.9750 - val_loss: 0.0408
Epoch 31/50
15/15             1s 16ms/step -
accuracy: 0.9750 - loss: 0.0577 - val_accuracy: 1.0000 - val_loss: 0.0269
Epoch 32/50
15/15             1s 16ms/step -
accuracy: 0.9937 - loss: 0.0403 - val_accuracy: 0.9750 - val_loss: 0.0785
Epoch 33/50
15/15             1s 16ms/step -
accuracy: 0.9759 - loss: 0.0580 - val_accuracy: 0.9750 - val_loss: 0.0542
Epoch 34/50
15/15             1s 16ms/step -
accuracy: 0.9763 - loss: 0.0662 - val_accuracy: 1.0000 - val_loss: 0.0294
Epoch 35/50
15/15             1s 16ms/step -
accuracy: 0.9906 - loss: 0.0369 - val_accuracy: 0.9625 - val_loss: 0.0872
Epoch 36/50
15/15             1s 17ms/step -
accuracy: 0.9656 - loss: 0.0950 - val_accuracy: 1.0000 - val_loss: 0.0225
Epoch 37/50
15/15             1s 17ms/step -
accuracy: 0.9859 - loss: 0.0534 - val_accuracy: 0.9750 - val_loss: 0.0523
Epoch 38/50
15/15             1s 16ms/step -
accuracy: 0.9819 - loss: 0.0483 - val_accuracy: 0.9500 - val_loss: 0.1329
Epoch 39/50
15/15             1s 16ms/step -
accuracy: 0.9820 - loss: 0.0510 - val_accuracy: 0.9750 - val_loss: 0.0503
Epoch 40/50
15/15             1s 19ms/step -
accuracy: 0.9944 - loss: 0.0286 - val_accuracy: 1.0000 - val_loss: 0.0227
Epoch 41/50
15/15             1s 18ms/step -
accuracy: 0.9919 - loss: 0.0346 - val_accuracy: 0.9875 - val_loss: 0.0484
Epoch 42/50
15/15             1s 18ms/step -
accuracy: 0.9828 - loss: 0.0571 - val_accuracy: 0.9750 - val_loss: 0.0614
```

```
Epoch 43/50
15/15             1s 19ms/step -
accuracy: 0.9825 - loss: 0.0467 - val_accuracy: 0.9875 - val_loss: 0.0283
Epoch 44/50
15/15             1s 18ms/step -
accuracy: 0.9929 - loss: 0.0240 - val_accuracy: 0.9875 - val_loss: 0.0472
Epoch 45/50
15/15             1s 16ms/step -
accuracy: 0.9896 - loss: 0.0247 - val_accuracy: 0.9875 - val_loss: 0.0383
Epoch 46/50
15/15             1s 18ms/step -
accuracy: 0.9683 - loss: 0.0776 - val_accuracy: 0.9750 - val_loss: 0.1516
Epoch 47/50
15/15             1s 17ms/step -
accuracy: 0.9445 - loss: 0.1452 - val_accuracy: 0.9750 - val_loss: 0.0846
Epoch 48/50
15/15             1s 17ms/step -
accuracy: 0.9606 - loss: 0.1060 - val_accuracy: 0.9875 - val_loss: 0.0479
Epoch 49/50
15/15             1s 16ms/step -
accuracy: 0.9793 - loss: 0.0703 - val_accuracy: 0.9625 - val_loss: 0.0844
Epoch 50/50
15/15             1s 17ms/step -
accuracy: 0.9778 - loss: 0.0572 - val_accuracy: 0.9875 - val_loss: 0.0453
```

[8]:
```python
import matplotlib.pyplot as plt

# Assuming 'history' object after model.fit and test data available
epochs = 50

# Evaluate the model on test data to get the test accuracy
test_loss, test_accuracy = model.evaluate(test_ds)

# Plot training and validation accuracy
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')

# Plot test accuracy at the final epoch (epoch 50)
plt.scatter(epochs - 1, test_accuracy, color='red', label='Test Accuracy (Epoch␣
↪50)', zorder=5)

# Add text annotation for test accuracy
plt.text(epochs - 10, test_accuracy - 0.2, f'Test Accuracy:\n{(test_accuracy *␣
↪100):.2f}%',
         horizontalalignment='left', verticalalignment='bottom', color='red')

# Add labels and title
```

```
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training, Validation, and Test Accuracy')

# Add legend and show the plot
plt.legend()
plt.show()
```

```
1/4             0s 16ms/step -
accuracy: 0.9800 - loss: 0.06754/4
            0s 6ms/step - accuracy: 0.9825 -
loss: 0.0559
```



10 Adding L2 regularization to the model to try to prevent overfitting

[9]:
```
from keras import regularizers

num_classes = 16
model_L2 = keras.Sequential([
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Flatten(),
  layers.Dense(128, activation='relu', kernel_regularizer=regularizers.l2(0.
 ↪001)),
  layers.Dense(num_classes)
])

model_L2.compile(optimizer="adam", loss=keras.losses.
 ↪SparseCategoricalCrossentropy(from_logits=True), metrics=["accuracy"])
```

[10]:
```
epochs = 50
history_L2 = model_L2.fit(
    train_ds,
    validation_data=val_ds,
    epochs = epochs
)
```

```
Epoch 1/50
15/15             2s 26ms/step -
accuracy: 0.0650 - loss: 2.9707 - val_accuracy: 0.1250 - val_loss: 2.6439
Epoch 2/50
15/15             1s 16ms/step -
accuracy: 0.2094 - loss: 2.4423 - val_accuracy: 0.1625 - val_loss: 2.2064
Epoch 3/50
15/15             1s 16ms/step -
accuracy: 0.3573 - loss: 1.8872 - val_accuracy: 0.5000 - val_loss: 1.6313
Epoch 4/50
15/15             1s 17ms/step -
accuracy: 0.5313 - loss: 1.4588 - val_accuracy: 0.5750 - val_loss: 1.1576
Epoch 5/50
15/15             1s 16ms/step -
accuracy: 0.5964 - loss: 1.1782 - val_accuracy: 0.6375 - val_loss: 1.1033
Epoch 6/50
15/15             1s 17ms/step -
accuracy: 0.6395 - loss: 1.0663 - val_accuracy: 0.6000 - val_loss: 0.9915
```

```
Epoch 7/50
15/15             1s 17ms/step -
accuracy: 0.7188 - loss: 0.8770 - val_accuracy: 0.8875 - val_loss: 0.5986
Epoch 8/50
15/15             1s 20ms/step -
accuracy: 0.7931 - loss: 0.6599 - val_accuracy: 0.8875 - val_loss: 0.4808
Epoch 9/50
15/15             1s 17ms/step -
accuracy: 0.8486 - loss: 0.5485 - val_accuracy: 0.8125 - val_loss: 0.6124
Epoch 10/50
15/15             1s 17ms/step -
accuracy: 0.8430 - loss: 0.5046 - val_accuracy: 0.9000 - val_loss: 0.4288
Epoch 11/50
15/15             1s 16ms/step -
accuracy: 0.8770 - loss: 0.4782 - val_accuracy: 0.8750 - val_loss: 0.4278
Epoch 12/50
15/15             1s 17ms/step -
accuracy: 0.8516 - loss: 0.5175 - val_accuracy: 0.9250 - val_loss: 0.3125
Epoch 13/50
15/15             1s 18ms/step -
accuracy: 0.9349 - loss: 0.3540 - val_accuracy: 0.9250 - val_loss: 0.3698
Epoch 14/50
15/15             1s 17ms/step -
accuracy: 0.8842 - loss: 0.3880 - val_accuracy: 0.9250 - val_loss: 0.3150
Epoch 15/50
15/15             1s 16ms/step -
accuracy: 0.9402 - loss: 0.3332 - val_accuracy: 0.8875 - val_loss: 0.3446
Epoch 16/50
15/15             1s 16ms/step -
accuracy: 0.9408 - loss: 0.2978 - val_accuracy: 0.9625 - val_loss: 0.2258
Epoch 17/50
15/15             1s 17ms/step -
accuracy: 0.9398 - loss: 0.3179 - val_accuracy: 0.9500 - val_loss: 0.2611
Epoch 18/50
15/15             1s 17ms/step -
accuracy: 0.9278 - loss: 0.2903 - val_accuracy: 0.9000 - val_loss: 0.3701
Epoch 19/50
15/15             1s 16ms/step -
accuracy: 0.9240 - loss: 0.2980 - val_accuracy: 0.9250 - val_loss: 0.3147
Epoch 20/50
15/15             1s 17ms/step -
accuracy: 0.9478 - loss: 0.2855 - val_accuracy: 0.9375 - val_loss: 0.2410
Epoch 21/50
15/15             1s 18ms/step -
accuracy: 0.8971 - loss: 0.4136 - val_accuracy: 0.9125 - val_loss: 0.3958
Epoch 22/50
15/15             1s 17ms/step -
accuracy: 0.9107 - loss: 0.3472 - val_accuracy: 0.9000 - val_loss: 0.3881
```

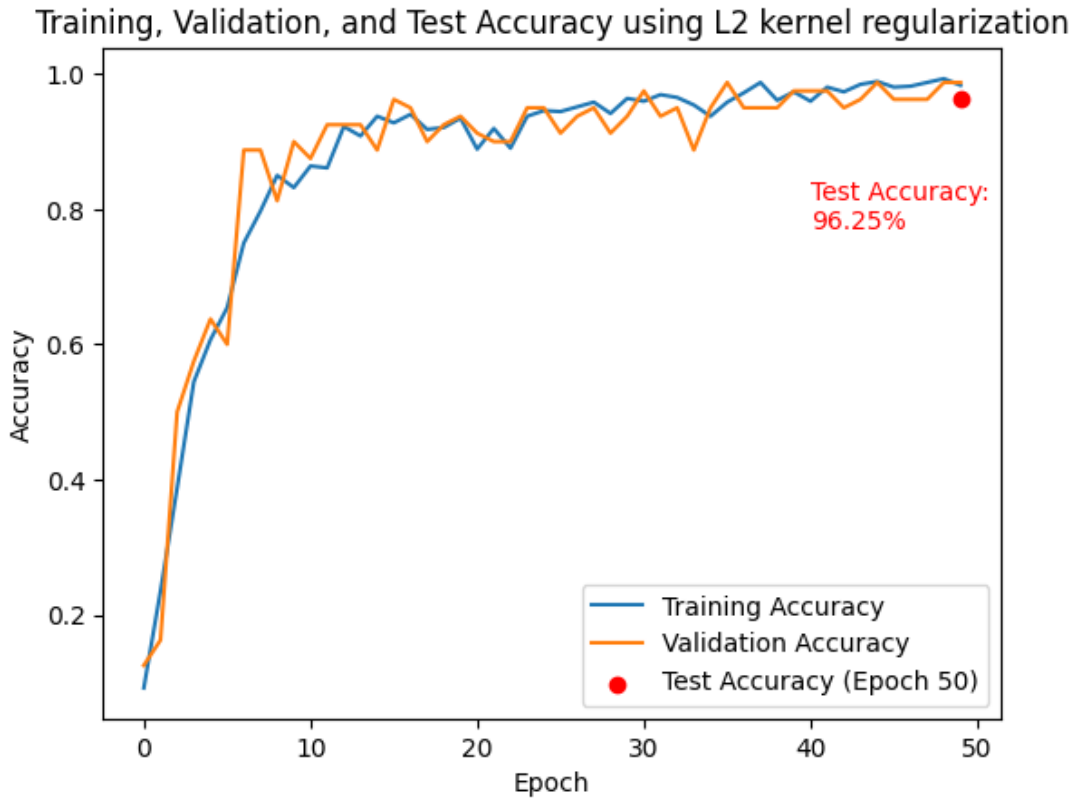
```
Epoch 23/50
15/15             1s 19ms/step -
accuracy: 0.8858 - loss: 0.3981 - val_accuracy: 0.9000 - val_loss: 0.3705
Epoch 24/50
15/15             1s 18ms/step -
accuracy: 0.9367 - loss: 0.2705 - val_accuracy: 0.9500 - val_loss: 0.3036
Epoch 25/50
15/15             1s 16ms/step -
accuracy: 0.9475 - loss: 0.2587 - val_accuracy: 0.9500 - val_loss: 0.2832
Epoch 26/50
15/15             1s 16ms/step -
accuracy: 0.9290 - loss: 0.2839 - val_accuracy: 0.9125 - val_loss: 0.4208
Epoch 27/50
15/15             1s 17ms/step -
accuracy: 0.9572 - loss: 0.2253 - val_accuracy: 0.9375 - val_loss: 0.2642
Epoch 28/50
15/15             1s 16ms/step -
accuracy: 0.9653 - loss: 0.2444 - val_accuracy: 0.9500 - val_loss: 0.2077
Epoch 29/50
15/15             1s 16ms/step -
accuracy: 0.9392 - loss: 0.2758 - val_accuracy: 0.9125 - val_loss: 0.2831
Epoch 30/50
15/15             1s 17ms/step -
accuracy: 0.9555 - loss: 0.2444 - val_accuracy: 0.9375 - val_loss: 0.3064
Epoch 31/50
15/15             1s 20ms/step -
accuracy: 0.9533 - loss: 0.2341 - val_accuracy: 0.9750 - val_loss: 0.1992
Epoch 32/50
15/15             1s 18ms/step -
accuracy: 0.9641 - loss: 0.2108 - val_accuracy: 0.9375 - val_loss: 0.2917
Epoch 33/50
15/15             1s 16ms/step -
accuracy: 0.9639 - loss: 0.1983 - val_accuracy: 0.9500 - val_loss: 0.2578
Epoch 34/50
15/15             1s 16ms/step -
accuracy: 0.9547 - loss: 0.2472 - val_accuracy: 0.8875 - val_loss: 0.4373
Epoch 35/50
15/15             1s 17ms/step -
accuracy: 0.9208 - loss: 0.2901 - val_accuracy: 0.9500 - val_loss: 0.2122
Epoch 36/50
15/15             1s 17ms/step -
accuracy: 0.9483 - loss: 0.2400 - val_accuracy: 0.9875 - val_loss: 0.1684
Epoch 37/50
15/15             1s 17ms/step -
accuracy: 0.9747 - loss: 0.1752 - val_accuracy: 0.9500 - val_loss: 0.2031
Epoch 38/50
15/15             1s 17ms/step -
accuracy: 0.9895 - loss: 0.1449 - val_accuracy: 0.9500 - val_loss: 0.2713
```

```
Epoch 39/50
15/15             1s 16ms/step -
accuracy: 0.9607 - loss: 0.2026 - val_accuracy: 0.9500 - val_loss: 0.2028
Epoch 40/50
15/15             1s 17ms/step -
accuracy: 0.9814 - loss: 0.1619 - val_accuracy: 0.9750 - val_loss: 0.1484
Epoch 41/50
15/15             1s 17ms/step -
accuracy: 0.9592 - loss: 0.2201 - val_accuracy: 0.9750 - val_loss: 0.1658
Epoch 42/50
15/15             1s 17ms/step -
accuracy: 0.9863 - loss: 0.1547 - val_accuracy: 0.9750 - val_loss: 0.1801
Epoch 43/50
15/15             1s 16ms/step -
accuracy: 0.9802 - loss: 0.1651 - val_accuracy: 0.9500 - val_loss: 0.1834
Epoch 44/50
15/15             1s 16ms/step -
accuracy: 0.9823 - loss: 0.1441 - val_accuracy: 0.9625 - val_loss: 0.1614
Epoch 45/50
15/15             1s 17ms/step -
accuracy: 0.9929 - loss: 0.1247 - val_accuracy: 0.9875 - val_loss: 0.1573
Epoch 46/50
15/15             1s 17ms/step -
accuracy: 0.9791 - loss: 0.1527 - val_accuracy: 0.9625 - val_loss: 0.2238
Epoch 47/50
15/15             1s 17ms/step -
accuracy: 0.9779 - loss: 0.1581 - val_accuracy: 0.9625 - val_loss: 0.1616
Epoch 48/50
15/15             1s 17ms/step -
accuracy: 0.9944 - loss: 0.1170 - val_accuracy: 0.9625 - val_loss: 0.1655
Epoch 49/50
15/15             1s 17ms/step -
accuracy: 0.9910 - loss: 0.1211 - val_accuracy: 0.9875 - val_loss: 0.1104
Epoch 50/50
15/15             1s 17ms/step -
accuracy: 0.9830 - loss: 0.1272 - val_accuracy: 0.9875 - val_loss: 0.1168
```

[11]:
```python
import matplotlib.pyplot as plt

# Assuming 'history' object after model.fit and test data available
epochs = 50

# Evaluate the model on test data to get the test accuracy
test_loss_l2, test_accuracy_l2 = model_L2.evaluate(test_ds)

# Plot training and validation accuracy
plt.plot(history_L2.history['accuracy'], label='Training Accuracy')
```

```
plt.plot(history_L2.history['val_accuracy'], label='Validation Accuracy')

# Plot test accuracy at the final epoch (epoch 50)
plt.scatter(epochs - 1, test_accuracy_l2, color='red', label='Test Accuracy␣
 ↪(Epoch 50)', zorder=5)

# Add text annotation for test accuracy
plt.text(epochs - 10, test_accuracy_l2 - 0.2, f'Test Accuracy:
 ↪\n{(test_accuracy_l2 * 100):.2f}%',
         horizontalalignment='left', verticalalignment='bottom', color='red')

# Add labels and title
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training, Validation, and Test Accuracy using L2 kernel␣
 ↪regularization')

# Add legend and show the plot
plt.legend()
plt.show()
```

```
1/4             0s 15ms/step -
accuracy: 0.9800 - loss: 0.14794/4
            0s 6ms/step - accuracy: 0.9690 -
loss: 0.2326
```



11 Adding Batch Normalization to the model to try to prevent overfitting

[21]:
```
num_classes = 16
model_batch = keras.Sequential([
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.BatchNormalization(),
  layers.MaxPooling2D(),
  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.BatchNormalization(),
  layers.MaxPooling2D(),
  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.BatchNormalization(),
  layers.MaxPooling2D(),
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.BatchNormalization(),
  layers.Dense(num_classes)
])
```

```
model_batch.compile(optimizer="adam", loss=keras.losses.
  ↪SparseCategoricalCrossentropy(from_logits=True), metrics=["accuracy"])
```

[22]:
```
epochs = 50
history_batch = model_batch.fit(
    train_ds,
    validation_data=val_ds,
    epochs = epochs
)
```

```
Epoch 1/50
15/15             3s 43ms/step -
accuracy: 0.3130 - loss: 2.4719 - val_accuracy: 0.0500 - val_loss: 2.8450
Epoch 2/50
15/15             1s 35ms/step -
accuracy: 0.6514 - loss: 1.1189 - val_accuracy: 0.0125 - val_loss: 2.8552
Epoch 3/50
15/15             1s 34ms/step -
accuracy: 0.7046 - loss: 0.9060 - val_accuracy: 0.1625 - val_loss: 3.0148
Epoch 4/50
15/15             1s 34ms/step -
accuracy: 0.8300 - loss: 0.6033 - val_accuracy: 0.1250 - val_loss: 3.5207
Epoch 5/50
15/15             1s 34ms/step -
accuracy: 0.9122 - loss: 0.4357 - val_accuracy: 0.1250 - val_loss: 4.3173
Epoch 6/50
15/15             1s 34ms/step -
accuracy: 0.9173 - loss: 0.3618 - val_accuracy: 0.1250 - val_loss: 5.0622
Epoch 7/50
15/15             1s 34ms/step -
accuracy: 0.9202 - loss: 0.3126 - val_accuracy: 0.1250 - val_loss: 5.8901
Epoch 8/50
15/15             1s 35ms/step -
accuracy: 0.9515 - loss: 0.2427 - val_accuracy: 0.1250 - val_loss: 6.7825
Epoch 9/50
15/15             1s 35ms/step -
accuracy: 0.9630 - loss: 0.1837 - val_accuracy: 0.1250 - val_loss: 7.0187
Epoch 10/50
15/15             1s 34ms/step -
accuracy: 0.9722 - loss: 0.1737 - val_accuracy: 0.1250 - val_loss: 7.5858
Epoch 11/50
15/15             1s 34ms/step -
accuracy: 0.9806 - loss: 0.1527 - val_accuracy: 0.1250 - val_loss: 8.0481
Epoch 12/50
15/15             1s 34ms/step -
accuracy: 0.9899 - loss: 0.1030 - val_accuracy: 0.1375 - val_loss: 8.1207
Epoch 13/50
```

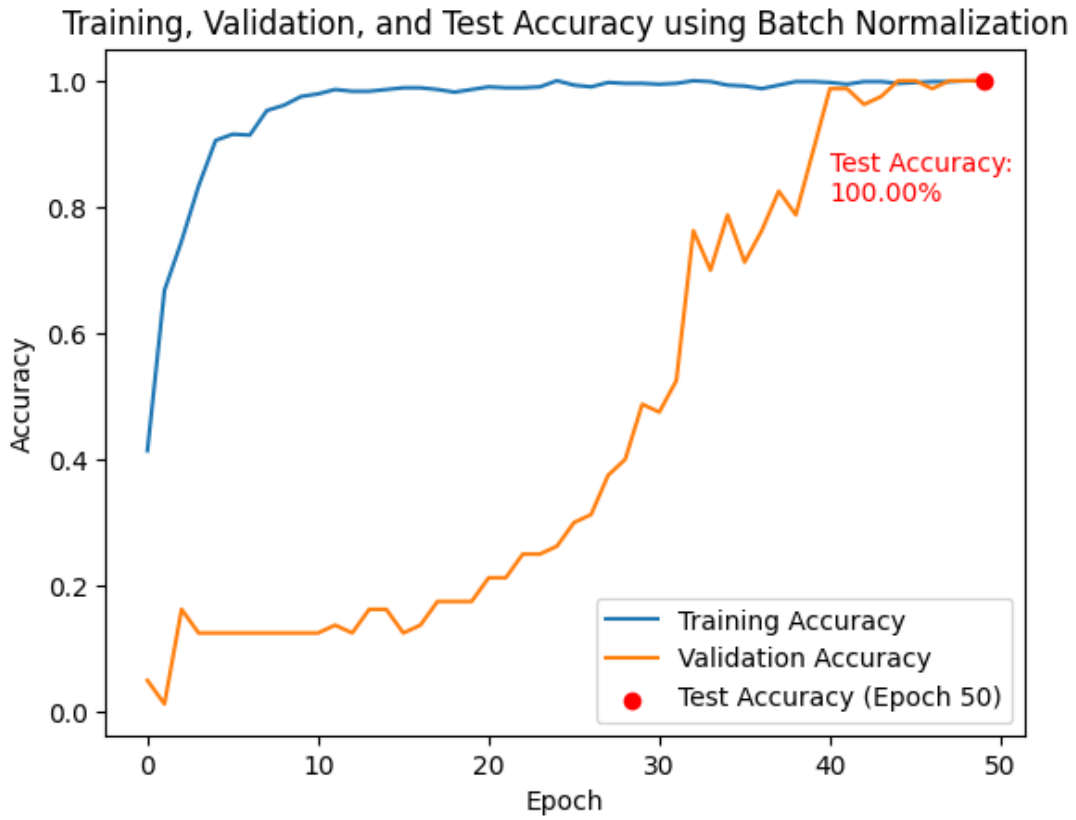
```
15/15            1s 35ms/step -
accuracy: 0.9870 - loss: 0.1083 - val_accuracy: 0.1250 - val_loss: 8.2933
Epoch 14/50
15/15            1s 34ms/step -
accuracy: 0.9826 - loss: 0.0943 - val_accuracy: 0.1625 - val_loss: 7.9654
Epoch 15/50
15/15            1s 34ms/step -
accuracy: 0.9875 - loss: 0.0955 - val_accuracy: 0.1625 - val_loss: 7.9178
Epoch 16/50
15/15            1s 35ms/step -
accuracy: 0.9852 - loss: 0.1047 - val_accuracy: 0.1250 - val_loss: 8.1536
Epoch 17/50
15/15            1s 37ms/step -
accuracy: 0.9947 - loss: 0.0537 - val_accuracy: 0.1375 - val_loss: 7.8112
Epoch 18/50
15/15            1s 38ms/step -
accuracy: 0.9870 - loss: 0.0710 - val_accuracy: 0.1750 - val_loss: 7.4111
Epoch 19/50
15/15            1s 35ms/step -
accuracy: 0.9786 - loss: 0.0995 - val_accuracy: 0.1750 - val_loss: 6.5521
Epoch 20/50
15/15            1s 35ms/step -
accuracy: 0.9891 - loss: 0.0710 - val_accuracy: 0.1750 - val_loss: 7.2180
Epoch 21/50
15/15            1s 34ms/step -
accuracy: 0.9885 - loss: 0.0682 - val_accuracy: 0.2125 - val_loss: 6.3595
Epoch 22/50
15/15            1s 35ms/step -
accuracy: 0.9888 - loss: 0.0554 - val_accuracy: 0.2125 - val_loss: 6.2618
Epoch 23/50
15/15            1s 34ms/step -
accuracy: 0.9909 - loss: 0.0568 - val_accuracy: 0.2500 - val_loss: 5.1737
Epoch 24/50
15/15            1s 35ms/step -
accuracy: 0.9903 - loss: 0.0577 - val_accuracy: 0.2500 - val_loss: 5.2447
Epoch 25/50
15/15            1s 35ms/step -
accuracy: 1.0000 - loss: 0.0332 - val_accuracy: 0.2625 - val_loss: 4.6390
Epoch 26/50
15/15            1s 34ms/step -
accuracy: 0.9967 - loss: 0.0324 - val_accuracy: 0.3000 - val_loss: 3.8891
Epoch 27/50
15/15            1s 37ms/step -
accuracy: 0.9955 - loss: 0.0326 - val_accuracy: 0.3125 - val_loss: 3.9129
Epoch 28/50
15/15            1s 34ms/step -
accuracy: 0.9982 - loss: 0.0359 - val_accuracy: 0.3750 - val_loss: 2.7422
Epoch 29/50
```

```
15/15            1s 34ms/step -
accuracy: 0.9966 - loss: 0.0310 - val_accuracy: 0.4000 - val_loss: 2.4669
Epoch 30/50
15/15            1s 35ms/step -
accuracy: 0.9940 - loss: 0.0265 - val_accuracy: 0.4875 - val_loss: 2.0088
Epoch 31/50
15/15            1s 34ms/step -
accuracy: 0.9963 - loss: 0.0271 - val_accuracy: 0.4750 - val_loss: 1.8986
Epoch 32/50
15/15            1s 34ms/step -
accuracy: 0.9948 - loss: 0.0268 - val_accuracy: 0.5250 - val_loss: 1.5479
Epoch 33/50
15/15            1s 34ms/step -
accuracy: 1.0000 - loss: 0.0226 - val_accuracy: 0.7625 - val_loss: 0.6974
Epoch 34/50
15/15            1s 33ms/step -
accuracy: 0.9992 - loss: 0.0201 - val_accuracy: 0.7000 - val_loss: 0.6231
Epoch 35/50
15/15            1s 37ms/step -
accuracy: 0.9939 - loss: 0.0370 - val_accuracy: 0.7875 - val_loss: 0.4523
Epoch 36/50
15/15            1s 36ms/step -
accuracy: 0.9978 - loss: 0.0241 - val_accuracy: 0.7125 - val_loss: 0.6620
Epoch 37/50
15/15            1s 35ms/step -
accuracy: 0.9892 - loss: 0.0346 - val_accuracy: 0.7625 - val_loss: 0.6588
Epoch 38/50
15/15            1s 34ms/step -
accuracy: 0.9937 - loss: 0.0347 - val_accuracy: 0.8250 - val_loss: 0.3642
Epoch 39/50
15/15            1s 34ms/step -
accuracy: 0.9995 - loss: 0.0244 - val_accuracy: 0.7875 - val_loss: 0.5144
Epoch 40/50
15/15            1s 35ms/step -
accuracy: 0.9976 - loss: 0.0184 - val_accuracy: 0.8875 - val_loss: 0.2247
Epoch 41/50
15/15            1s 35ms/step -
accuracy: 0.9981 - loss: 0.0143 - val_accuracy: 0.9875 - val_loss: 0.1029
Epoch 42/50
15/15            1s 35ms/step -
accuracy: 0.9933 - loss: 0.0261 - val_accuracy: 0.9875 - val_loss: 0.0571
Epoch 43/50
15/15            1s 33ms/step -
accuracy: 0.9958 - loss: 0.0203 - val_accuracy: 0.9625 - val_loss: 0.0770
Epoch 44/50
15/15            1s 37ms/step -
accuracy: 0.9993 - loss: 0.0132 - val_accuracy: 0.9750 - val_loss: 0.1015
Epoch 45/50
```

```
15/15             1s 35ms/step -
accuracy: 0.9956 - loss: 0.0249 - val_accuracy: 1.0000 - val_loss: 0.0336
Epoch 46/50
15/15             1s 34ms/step -
accuracy: 0.9986 - loss: 0.0155 - val_accuracy: 1.0000 - val_loss: 0.0198
Epoch 47/50
15/15             1s 36ms/step -
accuracy: 0.9984 - loss: 0.0175 - val_accuracy: 0.9875 - val_loss: 0.0331
Epoch 48/50
15/15             1s 35ms/step -
accuracy: 0.9992 - loss: 0.0127 - val_accuracy: 1.0000 - val_loss: 0.0285
Epoch 49/50
15/15             1s 34ms/step -
accuracy: 1.0000 - loss: 0.0131 - val_accuracy: 1.0000 - val_loss: 0.0146
Epoch 50/50
15/15             1s 33ms/step -
accuracy: 1.0000 - loss: 0.0094 - val_accuracy: 1.0000 - val_loss: 0.0124
```

[23]:
```python
import matplotlib.pyplot as plt

# Assuming 'history' object after model.fit and test data available
epochs = 50

# Evaluate the model on test data to get the test accuracy
test_loss_batch, test_accuracy_batch = model_batch.evaluate(test_ds)

# Plot training and validation accuracy
plt.plot(history_batch.history['accuracy'], label='Training Accuracy')
plt.plot(history_batch.history['val_accuracy'], label='Validation Accuracy')

# Plot test accuracy at the final epoch (epoch 50)
plt.scatter(epochs - 1, test_accuracy_batch, color='red', label='Test Accuracy␣
↪(Epoch 50)', zorder=5)

# Add text annotation for test accuracy
plt.text(epochs - 10, test_accuracy_batch - 0.2, f'Test Accuracy:
↪\n{(test_accuracy_batch * 100):.2f}%',
         horizontalalignment='left', verticalalignment='bottom', color='red')

# Add labels and title
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training, Validation, and Test Accuracy using Batch Normalization')

# Add legend and show the plot
plt.legend()
plt.show()
```

```
4/4                 0s 7ms/step -
accuracy: 1.0000 - loss: 0.0131
```



[ ]: