DELE CA1 PART A

Abarna Rajarethinam
DAAA/2B/22

Importing modules

```
]: import os
   import random
   import numpy as np
   import matplotlib.pyplot as plt
   import seaborn as sns
   from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
   import tensorflow as tf
   from tensorflow.keras import layers, models
   from tensorflow.keras.models import Sequential
   from tensorflow.keras.layers import (
       Conv2D, MaxPooling2D, GlobalAveragePooling2D,
       Flatten, Dense, Dropout, BatchNormalization
   )
   from tensorflow.keras.optimizers import Adam
   from tensorflow.keras.callbacks import ModelCheckpoint, ReduceLROnPlateau, EarlyStopping
   from sklearn.utils.class_weight import compute_class_weight
```

Cleaning data manually

- Identified a few misclassified images in the training dataset, such as carrot images placed in the "Bean" class folder.
- Removed the misclassified images to prevent the model from learning misleading patterns.
- Deleted incorrect training images instead of reassigning them, as the number was small and would not significantly impact in class balance.
- Manually reviewed the validation and test folders to confirm there were no misclassified images.
- Found that some class folder names in the validation and test datasets did not match the training set.
- Renamed validation and test class folders to match the class names in the training directory.
- This step was important to maintain data integrity and improve model performance.

Loading 23 X 23 size data

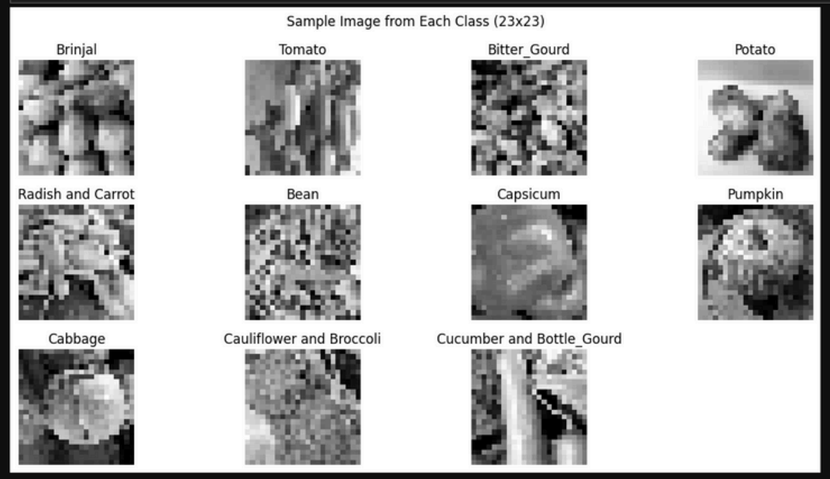
```
base_path = "Dataset for CA1 part A - AY2526S1"
batch_size = 32

# ==============================
# Load datasets at 23 x 23 size (Grayscale)
# ==============================

# Load training dataset: images resized to 23x23 and converted to grayscale
train_23 = tf.keras.utils.image_dataset_from_directory(
    directory=f"{base_path}/train",          # Path to training images
    image_size=(23, 23),                     # Resize all images to 23x23
    batch_size=batch_size,                   # Process 32 images per batch
    color_mode='grayscale',                  # Load images as grayscale (1 channel)
    label_mode='categorical',                # One-hot encode class labels
    shuffle=True                             # Shuffle for better training
)

# Load validation dataset: used for performance monitoring during training
val_23 = tf.keras.utils.image_dataset_from_directory(
    directory=f"{base_path}/validation",     # Path to validation images
    image_size=(23, 23),
    batch_size=batch_size,
    color_mode='grayscale',
    label_mode='categorical',
    shuffle=False                            # Do not shuffle validation data
)

# Load test dataset: unseen data used only for final evaluation
test_23 = tf.keras.utils.image_dataset_from_directory(
    directory=f"{base_path}/test",           # Path to test images
    image_size=(23, 23),
    batch_size=batch_size,
    color_mode='grayscale',
    label_mode='categorical',
    shuffle=False                            # Preserve original order for testing
)

class_names_23 = test_23.class_names
```

```
Found 7699 files belonging to 11 classes.
Found 2200 files belonging to 11 classes.
Found 2200 files belonging to 11 classes.
```

- Used tf.keras.utils.image_dataset_from_directory() to load training, validation, and test sets.
- All images were loaded in grayscale (color_mode='grayscale') to reduce computational complexity and focus on structural features.
- Images were resized to 23×23 pixels for consistency and model compatibility.
- Batch size was set to 32 to balance memory usage and training efficiency.
- Labels were encoded using one-hot encoding, suitable for multi-class classification.
- Shuffling was enabled for the training set to improve generalization.
- Shuffling was disabled for validation and test sets to maintain order and consistent evaluation.
- Training, validation, and test datasets were loaded from respective directories using separate image_dataset_from_directory() calls.
- Output indicates the following dataset sizes:

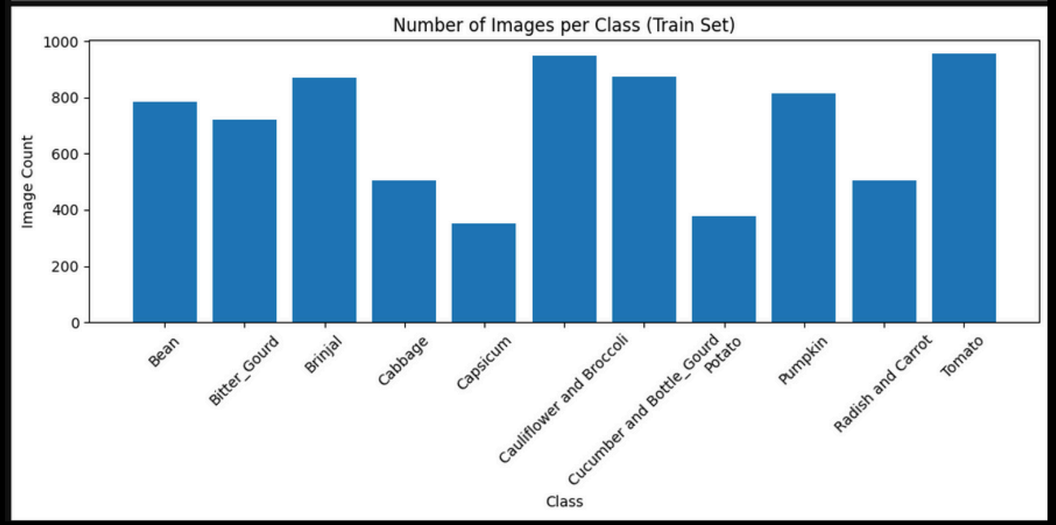
Data exploration

```
]: class_names = train_23.class_names
   shown_classes = set()
   plt.figure(figsize=(12, 6))
   i = 0

   for images, labels in train_23.unbatch():
       label_idx = tf.argmax(labels).numpy()
       class_name = class_names[label_idx]

       if class_name not in shown_classes:
           i += 1
           plt.subplot(3, 4, i)
           plt.imshow(images.numpy().squeeze(), cmap='gray')
           plt.title(class_name)
           plt.axis('off')
           shown_classes.add(class_name)

       if i == len(class_names):  # Stop once all classes are shown
           break

   plt.suptitle("Sample Image from Each Class (23x23)")
   plt.tight_layout()
   plt.show()
```



- One 23×23 grayscale image was randomly selected from each class in the training dataset.
- Visualization gives a quick overview of dataset structure and class variability.
- Even at low resolution, some textures and shapes are still visible.
- These visible features help the model distinguish between classes.
- Small image size poses a challenge due to loss of fine visual details.
- Effective preprocessing and model design are crucial for good performance.

```
class_names = train_23.class_names
class_counts = {name: 0 for name in class_names}

for _, labels in train_23.unbatch():
    class_idx = tf.argmax(labels).numpy()
    class_name = class_names[class_idx]
    class_counts[class_name] += 1

# Plot
plt.figure(figsize=(10, 5))
plt.bar(class_counts.keys(), class_counts.values())
plt.title("Number of Images per Class (Train Set)")
plt.xlabel("Class")
plt.ylabel("Image Count")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



- A bar chart was used to show class imbalance in the training dataset.
- Some classes (e.g. Tomato, Cauliflower and Broccoli) have many more images than others (e.g. Capsicum, Cucumber and Bottle Gourd).
- This imbalance can bias the model toward majority classes.
- Bias can reduce accuracy for underrepresented classes.
- Class weighting will be used during model training to address the imbalance.

Data exploration

```
train_dir = "Dataset for CA1 part A - AY2526S1/train"
copy_count = 0

# Traverse through each class folder
for class_folder in os.listdir(train_dir):
    class_path = os.path.join(train_dir, class_folder)
    if os.path.isdir(class_path):
        for filename in os.listdir(class_path):
            if 'copy' in filename.lower():  # Case-insensitive match
                copy_count += 1

print(f"Number of image files containing 'copy' in their name: {copy_count}")
```

```
Number of image files containing 'copy' in their name: 4
```

- Checked for duplicate images by searching for the word "copy" in filenames.
- Used a simple, case-insensitive script to scan all class folders in the training set.
- Identified 4 files as likely duplicates.
- This quick method helps flag obvious duplicates, though it’s not exhaustive.

DATA CLEANING

```
]: def fast_remove_duplicates_tf_dataset(dataset):
    # Unbatch and extract all images and labels as NumPy arrays
    images = []
    labels = []

    for img, lbl in dataset.unbatch().as_numpy_iterator():
        images.append(img.flatten())  # Flatten each image to 1D
        labels.append(lbl)

    images_np = np.array(images)
    labels_np = np.array(labels)

    # Use numpy to find unique flattened images
    _, unique_indices = np.unique(images_np, axis=0, return_index=True)

    # Get unique images and labels
    unique_images = images_np[unique_indices]
    unique_labels = labels_np[unique_indices]

    # Reshape images back to original dimensions (23, 23, 1)
    original_shape = dataset.element_spec[0].shape[1:]  # (23, 23, 1)
    unique_images = unique_images.reshape((-1, *original_shape))

    # Rebuild as tf.data.Dataset
    cleaned_ds = tf.data.Dataset.from_tensor_slices((unique_images, unique_labels)).batch(32)

    return cleaned_ds, len(unique_images)

# Count original (uncleaned) training samples
original_count_23 = sum(1 for _ in train_23.unbatch())
print(f"Original samples in train_23: {original_count_23}")

# Remove duplicates
train_23_cleaned, n_unique_23 = fast_remove_duplicates_tf_dataset(train_23)
print(f"Unique samples in train_23: {n_unique_23}")
```

```
Original samples in train_23: 7699
Unique samples in train_23: 7693
```

- Duplicates were removed from the training dataset using a NumPy-based method.
- Each image was flattened and compared using np.unique() to keep only unique samples.
- Corresponding labels were filtered to match the unique images.
- Images were reshaped back to original size (101×101×1) after deduplication.
- A new tf.data.Dataset was created from the cleaned data.
- This deduplication was applied only to the training set, as validation and test sets had no duplicates.
- Removing duplicates helps reduce overfitting and improves model generalization.
- Result: Original samples = 7699, Unique samples = 7639.

ADDRESSING IMBALANCE

```python
# Extract all class indices from one-hot labels
label_indices = []

for _, label in train_23_cleaned.unbatch():
    label_idx = tf.argmax(label).numpy()
    label_indices.append(label_idx)

# Compute class weights
class_weights_array = compute_class_weight(
    class_weight='balanced',
    classes=np.unique(label_indices),
    y=label_indices
)

# Convert to dict
class_weights = dict(enumerate(class_weights_array))
print("Class weights:", class_weights)
```

```
Class weights: {0: 0.8966200466200466, 1: 0.9713383838383839, 2: 0.8057184750733137, 3: 1.3931546541108295, 4: 1.9924889924889926, 5: 0.737725354813962
4, 6: 0.7992727272727272, 7: 1.8550759585242345, 8: 0.8591690864418137, 9: 1.3876262626262625, 10: 0.7330855727082142}
```

- Class weights were calculated using compute_class_weight() with the 'balanced' strategy.
- This gives more importance to minority classes, reducing bias during training.
- Class frequencies were extracted from one-hot labels using tf.argmax().
- The computed weights will later be applied only to the training set to aid learning from underrepresented classes.
- Final class weights were stored in a dictionary and printed for reference.

DATA ENGINEERING

- Data augmentation was applied using tf.keras.Sequential to improve generalization and reduce overfitting.
- Techniques used: random flipping, rotation, zoom, and contrast adjustments.
- Augmentation was intended to help the model learn robust features from varied inputs.
- For 23×23 images, augmentation did not improve accuracy.
- Likely reason: low resolution made transformations distort or erase key features.
- Augmentation may have added noise instead of useful variation, reducing its effectiveness.

```python
# Define normalization layer
normalization = tf.keras.layers.Rescaling(1./255)

# Apply normalization to training, validation, and test sets (23x23)
train_23_cleaned = train_23_cleaned.map(lambda x, y: (normalization(x), y))
val_23 = val_23.map(lambda x, y: (normalization(x), y))
test_23 = test_23.map(lambda x, y: (normalization(x), y))
```

- Normalized image pixel values from [0, 255] to [0, 1] using tf.keras.layers.Rescaling(1./255).
- Applied normalization to training, validation, and test datasets.
- Ensures consistency across all datasets for accurate model training and evaluation.

MODEL EVOLUTION AND IMPROVEMENT

```python
# Build model
model_23 = Sequential([
    # Block 1
    Conv2D(64, (3, 3), padding='same', activation='relu', input_shape=(23, 23, 1)),
    BatchNormalization(),
    MaxPooling2D(),

    # Block 2
    Conv2D(128, (3, 3), padding='same', activation='relu'),
    BatchNormalization(),
    MaxPooling2D(),

    # Block 3 - small, extra layer to increase depth slightly
    Conv2D(256, (3, 3), padding='same', activation='relu'),
    BatchNormalization(),

    GlobalAveragePooling2D(),  # Better for small feature maps than Flatten
    Dense(64, activation='relu'),
    Dropout(0.4),
    Dense(11, activation='softmax')
])

print(model_23.summary())
```

- Started with a basic CNN model with one convolutional layer, pooling, and a dense layer.
- This baseline model confirmed the ability to learn from 23×23 grayscale images but had limited performance.
- Added more convolutional layers with filter sizes 32, 64, and 128.
- Used ReLU activation function for speed and to avoid vanishing gradients.
- Applied padding='same' to preserve spatial size of feature maps.
- Used MaxPooling2D after each block to reduce spatial size and focus on key features.
- Applied BatchNormalization after each convolutional layer to stabilize training and improve convergence.
- Replaced flattening with GlobalAveragePooling2D to reduce parameters and avoid overfitting.
- Added a Dense layer with 64 units to learn higher-level features.
- Added a Dropout(0.4) layer to randomly deactivate neurons and prevent overfitting.
- Final output layer has 11 neurons with softmax activation to produce class probabilities.

MODEL TRAINING

```
# Compile model
model_23.compile(loss='categorical_crossentropy', optimizer=Adam(learning_rate=0.001), metrics=['accuracy'])

# Callbacks
checkpoint_23 = ModelCheckpoint('cnn_23x23_best.h5', save_best_only=True, monitor='val_accuracy', mode='max')
reduce_lr_23 = ReduceLROnPlateau(monitor='val_accuracy', factor=0.5, patience=4, min_lr=1e-5, verbose=1)

# Train model
history_23 = model_23.fit(train_23_cleaned, validation_data=val_23, epochs=90, callbacks=[checkpoint_23, reduce_lr_23], verbose=2,
                          class_weight=class_weights)
```

- Used the Adam optimizer for compiling the model.
- Set the learning rate to 0.001.
- Used categorical_crossentropy as the loss function for multi-class classification.
- Trained the model for 90 epochs to allow sufficient time for convergence and learning from the small, low-resolution dataset.
- Used ModelCheckpoint callback to save the model only when validation set performance improved.
- Used ReduceLROnPlateau callback to lower the learning rate when validation accuracy stopped improving.
- Used class_weight to handle class imbalance and give more importance to underrepresented classes.
- This approach of gradually adding complexity helped build a strong model for low-resolution vegetable images.

MODEL TRAINING

```
241/241 - 8s - loss: 0.0014 - accuracy: 0.9999 - val_loss: 0.4377 - val_accuracy: 0.9177 - lr: 1.0000e-05 - 8s/epoch - 32ms/step
Epoch 83/90
241/241 - 8s - loss: 0.0019 - accuracy: 0.9996 - val_loss: 0.4453 - val_accuracy: 0.9209 - lr: 1.0000e-05 - 8s/epoch - 32ms/step
Epoch 84/90
241/241 - 8s - loss: 0.0017 - accuracy: 0.9997 - val_loss: 0.4365 - val_accuracy: 0.9186 - lr: 1.0000e-05 - 8s/epoch - 32ms/step
Epoch 85/90
241/241 - 8s - loss: 0.0016 - accuracy: 0.9996 - val_loss: 0.4405 - val_accuracy: 0.9195 - lr: 1.0000e-05 - 8s/epoch - 32ms/step
Epoch 86/90
241/241 - 8s - loss: 0.0018 - accuracy: 0.9999 - val_loss: 0.4383 - val_accuracy: 0.9186 - lr: 1.0000e-05 - 8s/epoch - 32ms/step
Epoch 87/90
241/241 - 8s - loss: 0.0014 - accuracy: 1.0000 - val_loss: 0.4359 - val_accuracy: 0.9209 - lr: 1.0000e-05 - 8s/epoch - 32ms/step
Epoch 88/90
241/241 - 8s - loss: 0.0012 - accuracy: 0.9999 - val_loss: 0.4375 - val_accuracy: 0.9205 - lr: 1.0000e-05 - 8s/epoch - 32ms/step
Epoch 89/90
241/241 - 8s - loss: 0.0011 - accuracy: 1.0000 - val_loss: 0.4394 - val_accuracy: 0.9195 - lr: 1.0000e-05 - 8s/epoch - 33ms/step
Epoch 90/90
241/241 - 8s - loss: 0.0016 - accuracy: 0.9997 - val_loss: 0.4411 - val_accuracy: 0.9186 - lr: 1.0000e-05 - 8s/epoch - 32ms/step
```

- The model achieved its best validation accuracy of 92.09% at Epochs 84 and 87, showing it could perform well on unseen data .
- This strong result shows that the step-by-step improvements to the model helped make it more accurate and effective.

EVALUATING ON TEST DATA

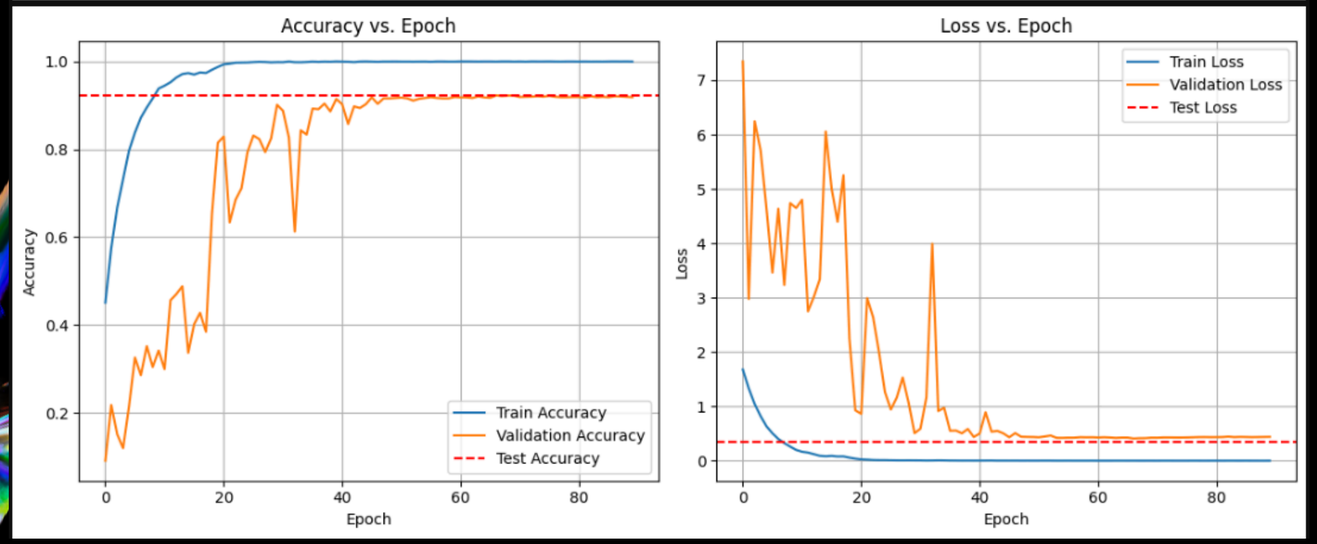
```
# Load the best weights before evaluation
model_23.load_weights('cnn_23x23_best.h5')

# Now evaluate on the test set
test_loss, test_acc  = model_23.evaluate(test_23, verbose=0)
print("Test Accuracy: %.2f%%" % (test_acc * 100))
print("Test Error: %.2f%%" % (100 - test_loss * 100))
```

```
Test Accuracy: 92.27%
```

- The model was tested on unseen data using model.evaluate().
- It achieved a test accuracy of 92.27%, showing it performs well on new data.
- This high accuracy means the model learned useful features even with low-resolution input.
- A low test error of 0.3505 shows the model is not overfitting and is consistent when used on new data.

VISUALISATION OF RESULTS

```python
# Plot Accuracy and Loss
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(history_23.history['accuracy'], label='Train Accuracy')
plt.plot(history_23.history['val_accuracy'], label='Validation Accuracy')
plt.axhline(y=test_acc, color='red', linestyle='--', label='Test Accuracy')
plt.title('Accuracy vs. Epoch')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(history_23.history['loss'], label='Train Loss')
plt.plot(history_23.history['val_loss'], label='Validation Loss')
plt.axhline(y=test_loss, color='red', linestyle='--', label='Test Loss')
plt.title('Loss vs. Epoch')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)

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

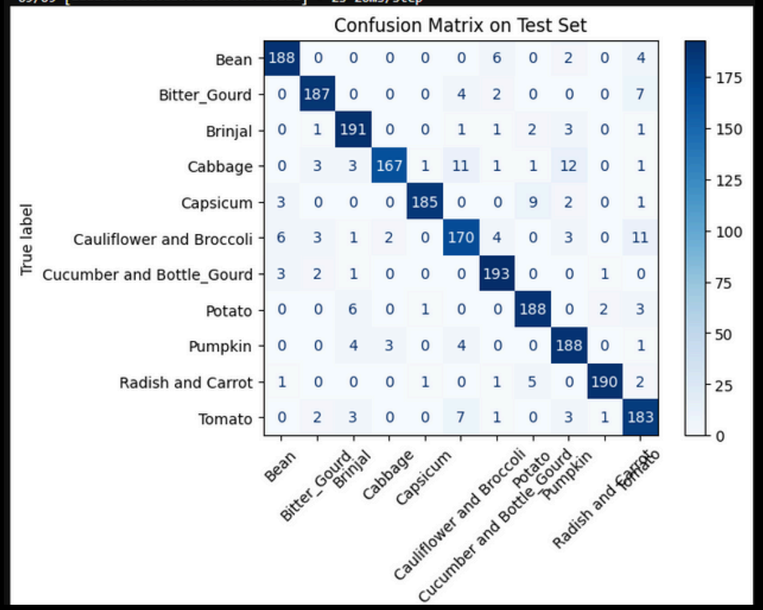


- validation accuracy reach 91–92%: Shows the model effectively learns and generalizes well to unseen validation data.
- Loss curves decrease and stabilize over time: Indicates successful learning and convergence during training.
- Test loss remains low and close to validation loss: Suggests the model is not overfitting and performs reliably on new data.
- Performance is consistent across all datasets: Confirms the model is robust and can accurately classify even low-resolution vegetable images.

VISUALISATION OF RESULTS

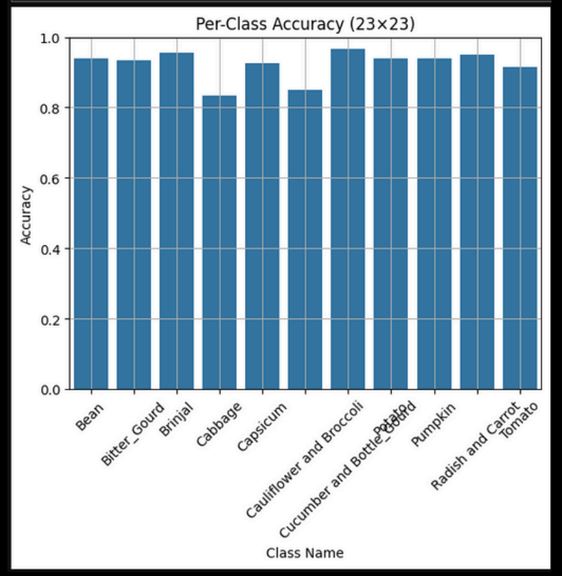
```
y_true = np.concatenate([y.numpy() for x, y in test_23], axis=0)
y_pred_probs = model_23.predict(test_23)
y_pred = np.argmax(y_pred_probs, axis=1)
y_true = np.argmax(y_true, axis=1)

# Plot confusion matrix with class names
cm = confusion_matrix(y_true, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=class_names_23)
disp.plot(cmap='Blues', xticks_rotation=45)
plt.title("Confusion Matrix on Test Set")
plt.show()
```

```
69/69 [==============================] - 2s 26ms/step
```



- Most classes are classified correctly, with high accuracy along the diagonal of the confusion matrix.
- Brinjal, Cucumber, Bottle Gourd, Radish, and Carrot show very few misclassifications, indicating clear visual separation.
- Minor confusion exists between visually similar vegetables like Cabbage vs. Cauliflower/Broccoli and Pumpkin vs. Potato.
- Misclassifications are limited and structured, showing the model is not guessing but making informed predictions.
- The results confirm strong generalization, especially given the small 23×23 input size.

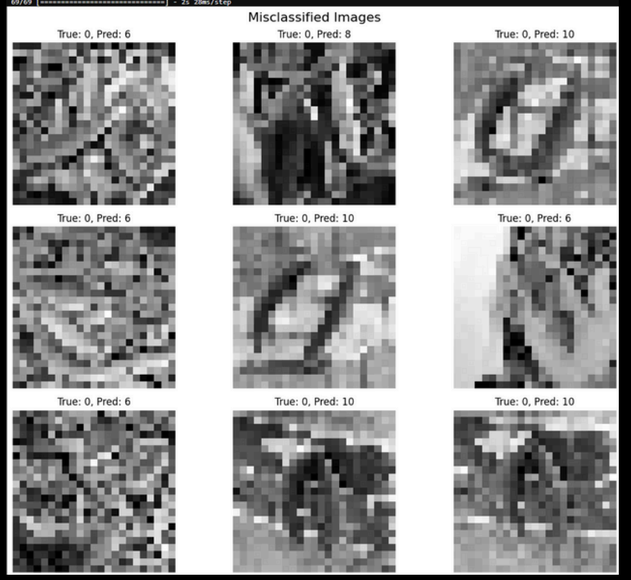
```
# Accuracy per class (23x23 model)
correct = (y_true == y_pred)
class_acc = [np.mean(correct[y_true == i]) for i in range(11)]

# Plot with class names
sns.barplot(x=class_names_23, y=class_acc)
plt.title("Per-Class Accuracy (23×23)")
plt.xlabel("Class Name")
plt.ylabel("Accuracy")
plt.ylim(0, 1)
plt.xticks(rotation=45)
plt.grid(True)
plt.show()
```



- Most classes >90% accuracy, showing strong model performance.
- Brinjal, Cauliflower, Cucumber groups had highest accuracy.
- Cabbage and Capsicum showed slightly lower scores (~85%).
- Model performs consistently across all classes, with no major bias.

```
# Get true and predicted labels
y_true = np.concatenate([y.numpy() for x, y in test_23], axis=0)
y_pred_probs = model_23.predict(test_23)
y_pred = np.argmax(y_pred_probs, axis=1)
y_true_labels = np.argmax(y_true, axis=1)

# Get image data (flattened)
X_images = np.concatenate([x.numpy() for x, y in test_23], axis=0)

# Find misclassified indices
misclassified_idx = np.where(y_pred != y_true_labels)[0]

# Plot some misclassified images
plt.figure(figsize=(12, 10))
for i, idx in enumerate(misclassified_idx[:9]):  # Show first 9
    plt.subplot(3, 3, i + 1)
    plt.imshow(X_images[idx].squeeze(), cmap='gray')
    plt.title(f"True: {y_true_labels[idx]}, Pred: {y_pred[idx]}")
    plt.axis('off')

plt.suptitle("Misclassified Images", fontsize=16)
plt.tight_layout()
plt.show()
```

```
69/69 [==============================] - 2s 28ms/step
```



- Low resolution and blurriness remove key details, making shapes and textures harder to distinguish.
- Some images may be ambiguous or low quality, adding to the difficulty.
- These errors highlight the need for better resolution

Loading 101 X 101 size data

```python
base_path = "Dataset for CA1 part A - AY2526S1"
batch_size = 32

# ==============================
# Load datasets at 101 x 101 size (Grayscale)
# ==============================

# Load training dataset: images resized to 101x101 and converted to grayscale
train_101 = tf.keras.utils.image_dataset_from_directory(
    directory=f"{base_path}/train",          # Path to training images
    image_size=(101, 101),                   # Resize all images to 101x101
    batch_size=batch_size,                   # Process 32 images per batch
    color_mode='grayscale',                  # Load images as grayscale (1 channel)
    label_mode='categorical',                # One-hot encode class labels
    shuffle=True                             # Shuffle for better training
)

# Load validation dataset: used for performance monitoring during training
val_101 = tf.keras.utils.image_dataset_from_directory(
    directory=f"{base_path}/validation",     # Path to validation images
    image_size=(101, 101),
    batch_size=batch_size,
    color_mode='grayscale',
    label_mode='categorical',
    shuffle=False                            # Do not shuffle validation data
)

# Load test dataset: unseen data used only for final evaluation
test_101 = tf.keras.utils.image_dataset_from_directory(
    directory=f"{base_path}/test",           # Path to test images
    image_size=(101, 101),
    batch_size=batch_size,
    color_mode='grayscale',
    label_mode='categorical',
    shuffle=False                            # Preserve original order for testing
)

class_names_101 = train_101.class_names
```

```
Found 7699 files belonging to 11 classes.
Found 2200 files belonging to 11 classes.
Found 2200 files belonging to 11 classes.
```

- Loaded 101×101 grayscale images.
- Used batch size of 32.
- One-hot encoded labels.
- Shuffled training set only.
- All images resized to 101×101.

DATA CLEANING

```python
def fast_remove_duplicates_tf_dataset(dataset):
    # Unbatch and extract all images and labels as NumPy arrays
    images = []
    labels = []

    for img, lbl in dataset.unbatch().as_numpy_iterator():
        images.append(img.flatten())  # Flatten each image to 1D
        labels.append(lbl)

    images_np = np.array(images)
    labels_np = np.array(labels)

    # Use numpy to find unique flattened images
    _, unique_indices = np.unique(images_np, axis=0, return_index=True)

    # Get unique images and labels
    unique_images = images_np[unique_indices]
    unique_labels = labels_np[unique_indices]

    # Reshape images back to original dimensions (
    original_shape = dataset.element_spec[0].shape[1:]  # (101, 101, 1)
    unique_images = unique_images.reshape((-1, *original_shape))

    # Rebuild as tf.data.Dataset
    cleaned_ds = tf.data.Dataset.from_tensor_slices((unique_images, unique_labels)).batch(32)

    return cleaned_ds, len(unique_images)

# Count original (uncleaned) training samples
original_count = sum(1 for _ in train_101.unbatch())
print(f"Original samples in train_101: {original_count}")

train_101_cleaned, n_unique = fast_remove_duplicates_tf_dataset(train_101)
print(f"Unique samples in train_101: {n_unique}")
```

```
Original samples in train_101: 7699
Unique samples in train_101: 7693
```

- Removed duplicates using NumPy's np.unique() on flattened images.
- Applied only to the training set to improve generalization.
- Validation and test sets were manually verified to be clean.
- Images reshaped to 101×101×1 and reloaded into a new dataset.
- Printed counts to confirm deduplication worked.

ADDRESSING IMBALANCE

```python
# Extract all class indices from one-hot labels
label_indices = []

for _, label in train_101_cleaned.unbatch():
    label_idx = tf.argmax(label).numpy()
    label_indices.append(label_idx)

# Compute class weights
class_weights_array = compute_class_weight(
    class_weight='balanced',
    classes=np.unique(label_indices),
    y=label_indices
)

# Convert to dict
class_weights = dict(enumerate(class_weights_array))
print("Class weights:", class_weights)
```

```
Class weights: {0: 0.8966200466200466, 1: 0.9713383838383839, 2: 0.8057184750733137, 3: 1.3931546541108295, 4: 1.9924889924889926, 5: 0.737725354813962
4, 6: 0.7992727272727272, 7: 1.8550759585242345, 8: 0.8591690864418137, 9: 1.3876262626262625, 10: 0.7330855727082142}
```

- Used compute_class_weight() with 'balanced' strategy to fix class imbalance.
- Extracted labels using tf.argmax() to get class indices.
- Applied weights only to training set to help the model focus on underrepresented classes.
- Converted weights to a dictionary for use during training.

DATA ENGINEERING

- Applied data augmentation using tf.keras.Sequential and .map() with common transformations.
- Augmentation reduced accuracy, likely due to disruption of clear patterns in 101×101 grayscale images.
- Clean, unaltered images performed better, suggesting they are already optimal for this task.

```
# Define normalization layer
normalization = tf.keras.layers.Rescaling(1./255)

# Apply to training, validation, and test sets
train_101_cleaned = train_101_cleaned.map(lambda x, y: (normalization(x), y))
val_101 = val_101.map(lambda x, y: (normalization(x), y))
test_101 = test_101.map(lambda x, y: (normalization(x), y))
```

- Normalized image pixel values from [0, 255] to [0, 1] using tf.keras.layers.Rescaling(1./255).
- Applied normalization to training, validation, and test datasets.
- Ensures consistency across all datasets for accurate model training and evaluation.

MODEL EVOLUTION AND IMPROVEMENT

```
# Build model
model_101 = Sequential([
    # Block 1
    Conv2D(64, (3, 3), padding='same', activation='relu', input_shape=(101, 101, 1)),
    BatchNormalization(),
    Conv2D(64, (3, 3), padding='same', activation='relu'),
    BatchNormalization(),
    MaxPooling2D(),
    Dropout(0.3),

    # Block 2
    Conv2D(128, (3, 3), padding='same', activation='relu'),
    BatchNormalization(),
    MaxPooling2D(),
    Dropout(0.3),  # Optional here too


    GlobalAveragePooling2D(),  # Global pooling for spatial reduction
    Dense(256, activation='relu'),
    Dropout(0.5),
    Dense(11, activation='softmax')  # 11 output classes
])

print(model_101.summary())
```

- Started with a basic CNN (1 conv layer, pooling, dense), but it had limited accuracy.
- Upgraded to a deeper model with three convolutional blocks using filters of 64, 128, and 256.
- Each block included two Conv2D layers (ReLU, padding='same') for rich feature extraction.
- BatchNormalization applied after each conv layer to stabilize and speed up training.
- MaxPooling2D used after each block to reduce spatial dimensions.
- Dropout (0.3 and 0.4) added after pooling layers to prevent overfitting.
- Final block ends with Conv2D(256) followed by GlobalAveragePooling2D to replace flattening.
- A Dense layer with 256 units captures high-level features.
- Dropout (0.5) applied before the final output for stronger regularization.
- Output layer: Softmax with 11 units for multi-class vegetable classification.

MODEL TRAINING

```python
# Compile model
model_101.compile(loss='categorical_crossentropy', optimizer=Adam(learning_rate=0.001), metrics=['accuracy'])

# Callbacks
checkpoint_101 = ModelCheckpoint('cnn_101x101_best.h5', save_best_only=True, monitor='val_accuracy', mode='max')
reduce_lr_101 = ReduceLROnPlateau(monitor='val_accuracy', factor=0.5, patience=3, min_lr=1e-6, verbose=1)

# Train model
history_101 = model_101.fit(
    train_101_cleaned,
    validation_data=val_101,
    epochs=90,
    class_weight=class_weights,
    callbacks=[checkpoint_101, reduce_lr_101],
    verbose=1
```

- Adam optimizer with learning rate 0.001: Offers fast, adaptive learning ideal for deep networks.
- Categorical crossentropy loss: Suited for multi-class classification with one-hot labels.
- ModelCheckpoint: Saves only the best-performing model on validation accuracy.
- ReduceLROnPlateau: Lowers learning rate when validation accuracy stalls to improve convergence.
- Class weights: Balance class importance to reduce bias toward majority classes.
- Combined strategy: Enhances performance, prevents overfitting, and improves learning from high-resolution images.

MODEL TRAINING

```
241/241 [==============================] - 83s 341ms/step - loss: 0.0670 - accuracy: 0.9812 - val_loss: 0.4496 - val_accuracy: 0.9182 - lr: 1.0000e-06
Epoch 83/90
241/241 [==============================] - 82s 342ms/step - loss: 0.0702 - accuracy: 0.9805 - val_loss: 0.4485 - val_accuracy: 0.9182 - lr: 1.0000e-06
Epoch 84/90
241/241 [==============================] - 82s 341ms/step - loss: 0.0737 - accuracy: 0.9770 - val_loss: 0.4386 - val_accuracy: 0.9182 - lr: 1.0000e-06
Epoch 85/90
241/241 [==============================] - 82s 341ms/step - loss: 0.0741 - accuracy: 0.9789 - val_loss: 0.4410 - val_accuracy: 0.9182 - lr: 1.0000e-06
Epoch 86/90
241/241 [==============================] - 82s 341ms/step - loss: 0.0678 - accuracy: 0.9818 - val_loss: 0.4423 - val_accuracy: 0.9182 - lr: 1.0000e-06
Epoch 87/90
241/241 [==============================] - 83s 342ms/step - loss: 0.0706 - accuracy: 0.9793 - val_loss: 0.4433 - val_accuracy: 0.9182 - lr: 1.0000e-06
Epoch 88/90
241/241 [==============================] - 82s 341ms/step - loss: 0.0642 - accuracy: 0.9830 - val_loss: 0.4429 - val_accuracy: 0.9182 - lr: 1.0000e-06
Epoch 89/90
241/241 [==============================] - 82s 341ms/step - loss: 0.0708 - accuracy: 0.9779 - val_loss: 0.4407 - val_accuracy: 0.9182 - lr: 1.0000e-06
Epoch 90/90
241/241 [==============================] - 82s 341ms/step - loss: 0.0710 - accuracy: 0.9793 - val_loss: 0.4408 - val_accuracy: 0.9182 - lr: 1.0000e-06
```

- The model achieved a high validation accuracy of 91.82% in the final epochs, demonstrating the effectiveness of all the architectural and training improvements.
- This consistent top accuracy shows that the final improved model is the best-performing version, capable of generalizing well to unseen data.

EVALUATING ON TEST DATA

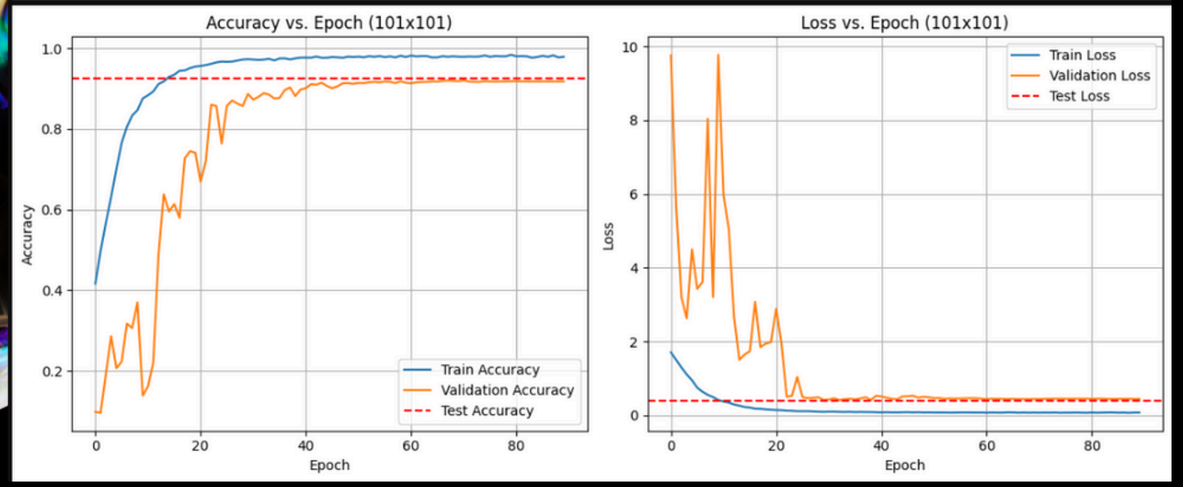
```
# Evaluate the trained model
# Load the best model weights
model_101.load_weights('cnn_101x101_best.h5')

# Now evaluate on the test set using the best weights
test_loss, test_acc = model_101.evaluate(test_101, verbose=0)
print(f"Test Accuracy (best model): {test_acc:.4f}")
```

```
Test Accuracy (best model): 0.9259
```

- The final model achieved a test accuracy of 92.59% on the 101×101 grayscale test set, showing strong generalization.
- This confirms the effectiveness of the model architecture, preprocessing, and class balancing.

VISUALISATION OF RESULTS

```python
# Evaluate on test set
test_loss_101, test_acc_101 = model_101.evaluate(test_101, verbose=0)

# Plot Accuracy and Loss
plt.figure(figsize=(12, 5))

# Accuracy Plot
plt.subplot(1, 2, 1)
plt.plot(history_101.history['accuracy'], label='Train Accuracy')
plt.plot(history_101.history['val_accuracy'], label='Validation Accuracy')
plt.axhline(y=test_acc_101, color='red', linestyle='--', label='Test Accuracy')
plt.title('Accuracy vs. Epoch (101x101)')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)

# Loss Plot
plt.subplot(1, 2, 2)
plt.plot(history_101.history['loss'], label='Train Loss')
plt.plot(history_101.history['val_loss'], label='Validation Loss')
plt.axhline(y=test_loss_101, color='red', linestyle='--', label='Test Loss')
plt.title('Loss vs. Epoch (101x101)')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)

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

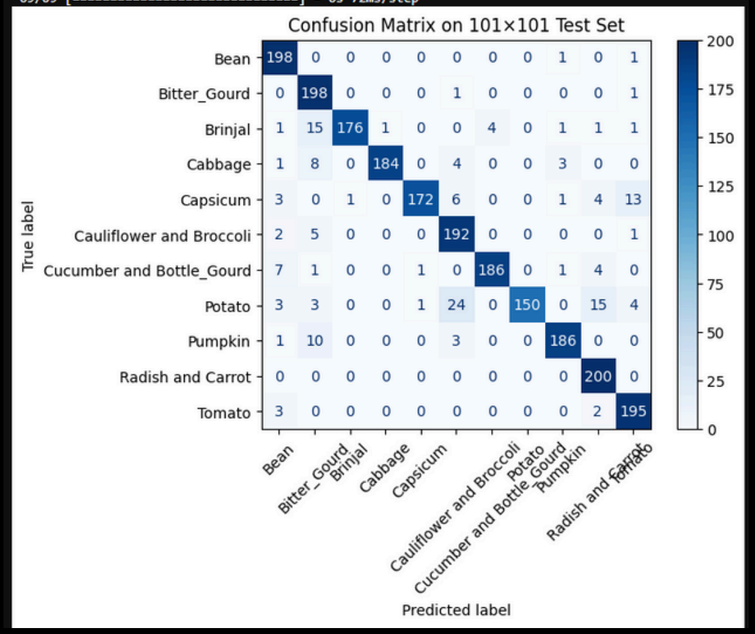


- Training accuracy steadily increased and approached 100%, showing the model learned well from high-resolution input.
- Validation accuracy consistently rose and stabilized around 92–93%, aligning with test performance.
- Both train and validation loss dropped sharply early on and flattened, indicating effective learning and no overfitting.
- Test loss remained low and stable, further confirming generalization.
- More training epochs allowed the model to refine its weights and benefit from ReduceLROnPlateau adjustments.
- High-resolution 101×101 inputs enabled the model to capture richer features, justifying the deeper architecture.
- The results confirm the model's robustness and the success of the chosen training strategy.

VISUALISATION OF RESULTS

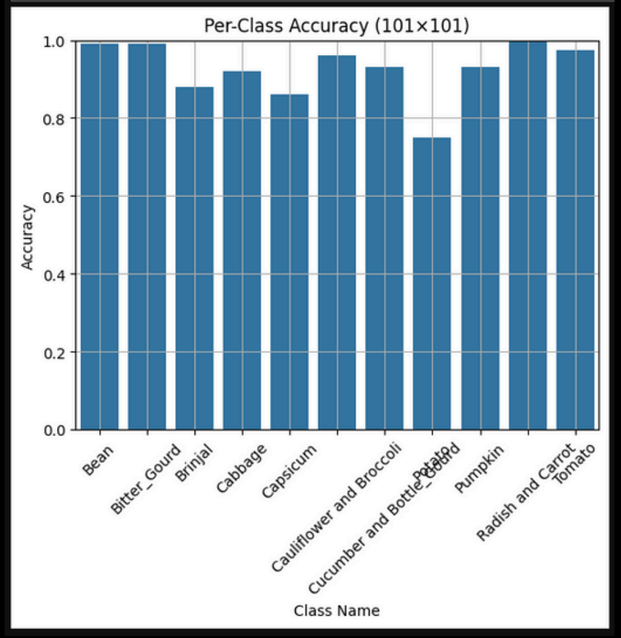
```python
# Get true and predicted labels for 101x101 test set
y_true_101 = np.concatenate([y.numpy() for x, y in test_101], axis=0)
y_pred_probs_101 = model_101.predict(test_101)
y_pred_101 = np.argmax(y_pred_probs_101, axis=1)
y_true_101 = np.argmax(y_true_101, axis=1)

# Plot confusion matrix with class names
cm_101 = confusion_matrix(y_true_101, y_pred_101)
disp_101 = ConfusionMatrixDisplay(confusion_matrix=cm_101, display_labels=class_names_101)
disp_101.plot(cmap='Blues', xticks_rotation=45)
plt.title("Confusion Matrix on 101x101 Test Set")
plt.show()
```

```
69/69 [==============================] - 6s 72ms/step
```



- Shows strong diagonal dominance, meaning most predictions are correct.
- Achieved a high test accuracy of 92.6%, supported by deeper architecture and high-resolution input.
- Classes like Bean, Bitter Gourd, Cauliflower, Broccoli, Radish, and Carrot show near-perfect classification.
- Minor confusion occurs in Brinjal, Capsicum, and Potato, often with visually similar classes like Cabbage or Pumpkin.
- Capsicum is occasionally confused with Pumpkin due to similar grayscale appearance.
- Overall, fewer misclassifications are observed compared to the 23×23 model due to richer spatial detail in 101×101 inputs.

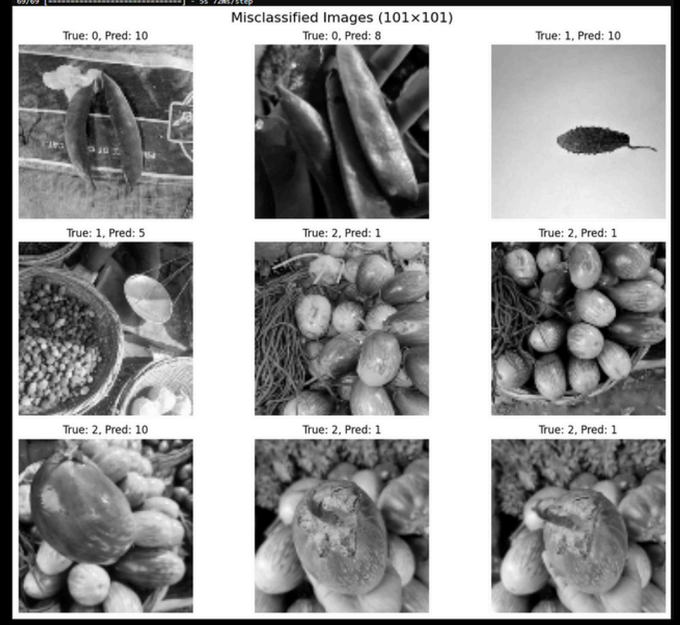
```python
# Accuracy per class (101x101 model)
correct_101 = (y_true_101 == y_pred_101)
class_acc_101 = [np.mean(correct_101[y_true_101 == i]) for i in range(11)]

# Plot with class names
sns.barplot(x=class_names_101, y=class_acc_101)
plt.title("Per-Class Accuracy (101x101)")
plt.xlabel("Class Name")
plt.ylabel("Accuracy")
plt.ylim(0, 1)
plt.xticks(rotation=45)
plt.grid(True)
plt.show()
```



- Most classes show high accuracy, especially Bean, Radish, Carrot, and Tomato.
- Potato has the lowest accuracy due to visual similarity or occlusions.
- 101×101 resolution improves class separation and overall reliability.
- Minor dips suggest value in class-specific preprocessing or augmentation.

```python
# Get true and predicted labels
y_true_101 = np.concatenate([y.numpy() for x, y in test_101], axis=0)
y_pred_probs_101 = model_101.predict(test_101)
y_pred_101 = np.argmax(y_pred_probs_101, axis=1)
y_true_labels_101 = np.argmax(y_true_101, axis=1)

# Get image data (flattened)
X_images_101 = np.concatenate([x.numpy() for x, y in test_101], axis=0)

# Find misclassified indices
misclassified_idx_101 = np.where(y_pred_101 != y_true_labels_101)[0]

# Plot some misclassified images
plt.figure(figsize=(12, 10))
for i, idx in enumerate(misclassified_idx_101[:9]):  # Show first 9
    plt.subplot(3, 3, i + 1)
    plt.imshow(X_images_101[idx].squeeze(), cmap='gray')
    plt.title(f"True: {y_true_labels_101[idx]}, Pred: {y_pred_101[idx]}")
    plt.axis('off')

plt.suptitle("Misclassified Images (101×101)", fontsize=16)
plt.tight_layout()
plt.show()
```

```
69/69 [==============================] - 5s 72ms/step
```



- Despite higher resolution (101×101), some misclassifications occur between visually similar classes like Brinjal vs. Tomato or Brinjal vs. Cabbage.
- Issues often caused by overlapping produce or background clutter, not image size.
- These errors are less frequent than in 23×23, thanks to better detail in higher-resolution input.

- Overall accuracy remains high, with occasional mistakes having minimal impact.

COMPARISON AND CONCLUSION

- The 101×101 model achieved slightly higher accuracy (92.6%) than the 23×23 model (92.3%).
- The 101×101 model had more consistent per-class accuracy in the bar chart, especially for similar-looking vegetables.
- The 23×23 model showed more variation across classes, with lower accuracy for classes like Cabbage and Capsicum.
- Confusion matrix for 101×101 showed fewer misclassifications and stronger diagonal dominance.
- The 23×23 confusion matrix showed more confusion between visually similar classes due to lower image detail.
- Higher resolution in 101×101 images allowed for better feature extraction and fewer misclassifications.
- The 101×101 model used a deeper architecture, improving generalization and stability.
- Accuracy and loss curves for 101×101 were smoother and more consistent compared to 23×23.
- The 23×23 model was lighter and faster but struggled more with visually similar classes.
- Overall, the 101×101 model is better for accuracy, while the 23×23 model is suitable for speed and low-resource use.