Final Project Analysis

1a)

The dataset we worked with is a dataset made of animal pictures. We divided the dataset into 70% training images, 20% validation images, and 10% test images.

- training images: 4200

- validation images: 1200

- test images: 600

The preprocessing for the dataset included normalization, data augmentation, and one-hot encoding. We decided not to use both normalization and standardization together because applying both to the same dataset can be redundant or counterproductive considering that they serve overlapping roles.

```
def normalize_images(images):
    return images / 255.0  # Normalize pixel values to [0, 1]
```

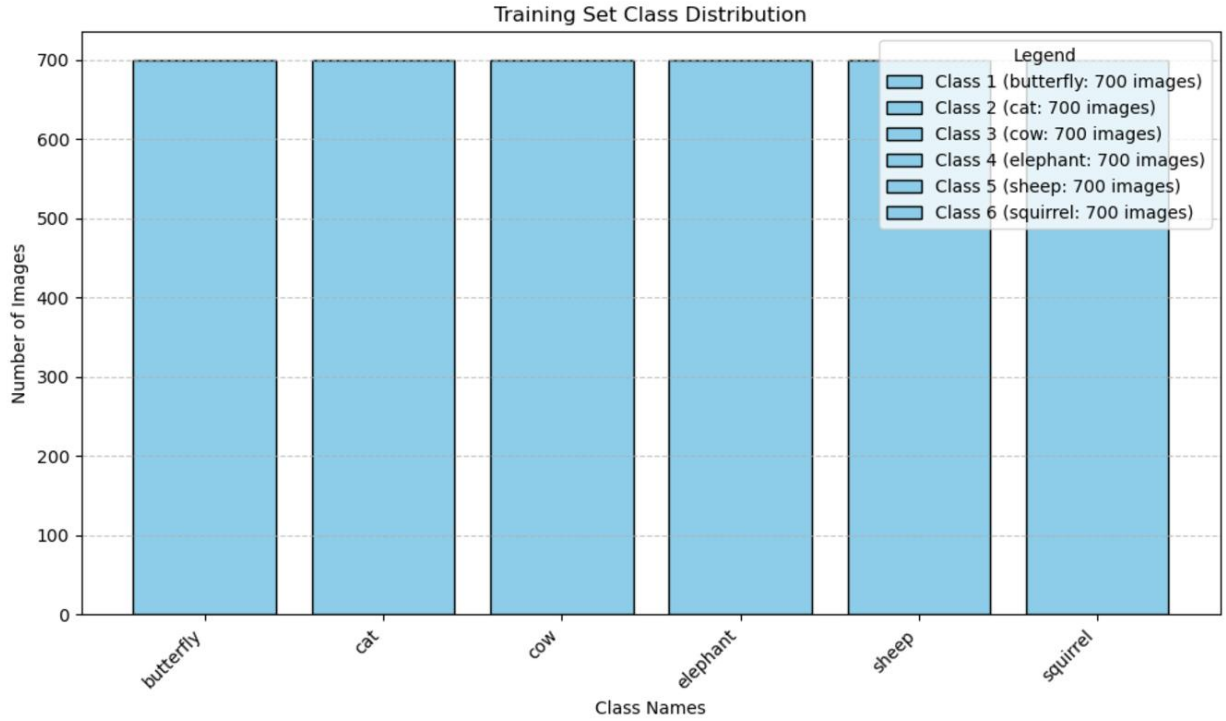
```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(
    rotation_range=10,
    width_shift_range=0.1,
    height_shift_range=0.1,
    shear_range=0.1,
    zoom_range=0.1,
    horizontal_flip=True,
    fill_mode='nearest',
)
datagen.fit(train_images_normalized)
```

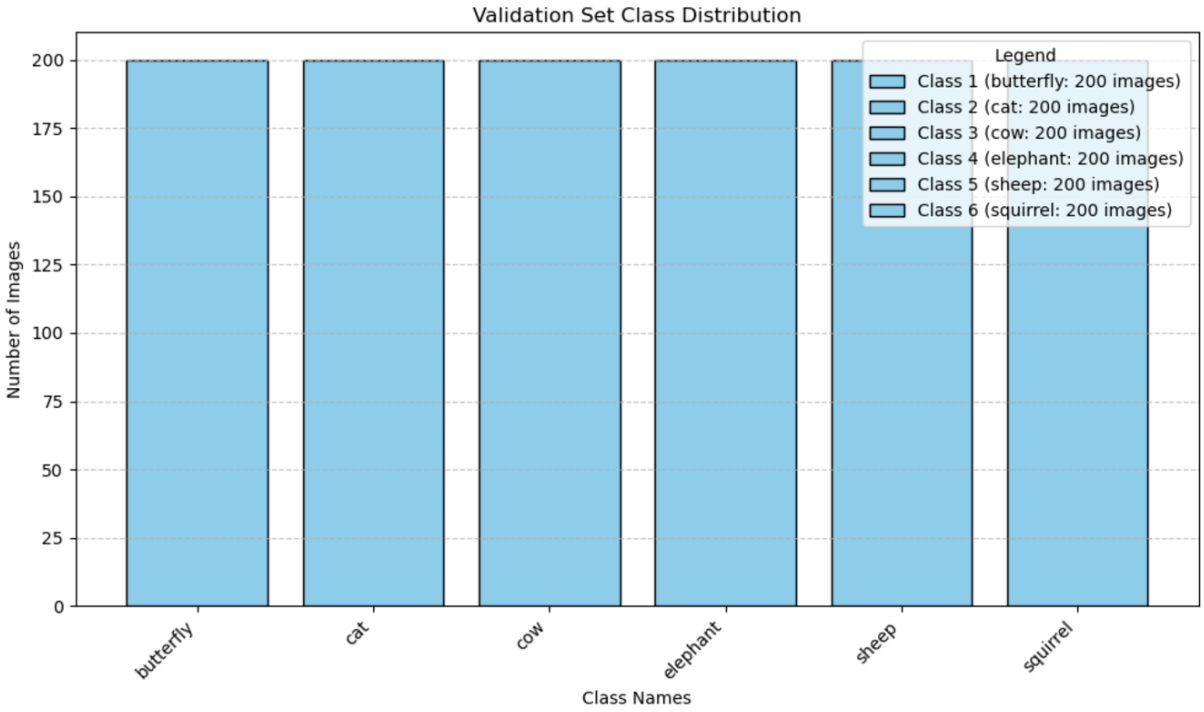
```
# Apply one-hot encoding to the adjusted labels
train_labels = to_categorical(train_labels, num_classes=6)
validation_labels = to_categorical(validation_labels, num_classes=6)
```

Our next step was to represent the classes in training and validation sets:

```python
import matplotlib.pyplot as plt
import numpy as np

def plot_class_distribution(class_names, class_counts, dataset_name):

    # Map counts based on class names
    counts = [class_counts.get(name, 0) for name in class_names]

    # Bar width and positions
    bar_positions = np.arange(len(class_names))

    # Plotting
    plt.figure(figsize=(10, 6))
    bars = plt.bar(bar_positions, counts, color='skyblue', edgecolor='black')

    # Add labels, title, and grid
    plt.xlabel("Class Names")
    plt.ylabel("Number of Images")
    plt.title(f"{dataset_name} Class Distribution")
    plt.xticks(bar_positions, class_names, rotation=45, ha="right")
    plt.tight_layout()
    plt.grid(axis='y', linestyle='--', alpha=0.7)

    # Adding legend
    legend_labels = [f"Class {i+1} ({name}: {counts[i]} images)" for i, name in enumerate(class_names)]
    plt.legend(bars, legend_labels, loc='upper right', title="Legend")

    # Show plot
    plt.show()
```

The result of the code are the following graphs:





We can notice that everything is very balanced. Each class of the training set has 700 images, and each class of the validation set has 200 images. Having a balanced dataset where each class has the same number of images is great because it ensures the model treats all classes equally. This prevents the model from getting biased toward more frequent classes and makes sure it learns the important features from every category. When the dataset is balanced, we will see more reliable results in terms of accuracy

because the model isn't focusing more on one class over another. It also means we don't have to worry about complex techniques to fix class imbalances.

1b)

The next step was to create the 3 models:

Model 1:

- A baseline CNN architecture using ReLU activation and the Adam optimizer with a batch size of 32.
- It features four convolutional layers with progressively increasing filters (32, 64, 128, 256), each followed by batch normalization and pooling (MaxPooling for spatial reduction).
- A Global Average Pooling layer is used to reduce dimensionality instead of flattening.
- A dense layer with 128 neurons, ReLU activation, and dropout (0.5) is added to prevent overfitting. The output layer uses a softmax activation to classify into 6 classes.
- Learning rate: 1e-4.

Model 1 serves as the baseline, designed with simplicity and standard practices in mind. It uses ReLU activation for efficient learning, the Adam optimizer for stable training, and a modest batch size of 32. Its architecture consists of four convolutional layers with progressively increasing filters (32 → 256), each followed by MaxPooling for spatial reduction. It uses Global Average Pooling instead of flattening to reduce dimensionality efficiently and includes a dense layer with 128 neurons and a 50% dropout rate to prevent overfitting.

```python
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.optimizers import Adam

# Define the CNN model
model_1 = models.Sequential()

# Add the first convolutional layer
model_1.add(layers.Conv2D(32, (3, 3), activation='relu', padding='same', input_shape=(32, 32, 3)))
model_1.add(layers.BatchNormalization())
model_1.add(layers.MaxPooling2D((2, 2)))

# Add the second convolutional layer
model_1.add(layers.Conv2D(64, (3, 3), activation='relu', padding='same'))
model_1.add(layers.BatchNormalization())
model_1.add(layers.MaxPooling2D((2, 2)))

# Add the third convolutional layer
model_1.add(layers.Conv2D(128, (3, 3), activation='relu', padding='same'))
model_1.add(layers.BatchNormalization())
model_1.add(layers.MaxPooling2D((2, 2)))

# Add the fourth convolutional layer
model_1.add(layers.Conv2D(256, (3, 3), activation='relu', padding='same'))
model_1.add(layers.BatchNormalization())
model_1.add(layers.GlobalAveragePooling2D())  # Use GAP instead of Flatten

# Add a fully connected (dense) layer
model_1.add(layers.Dense(128, activation='relu'))
model_1.add(layers.Dropout(0.5))  # Add dropout for regularization

# Add the output layer
model_1.add(layers.Dense(6, activation='softmax'))

# Compile the model with a tuned learning rate
model_1.compile(optimizer=Adam(learning_rate=1e-4),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Print model summary
model_1.summary()

# Train the model

history_1 = model_1.fit(
    datagen.flow(train_images_normalized, train_labels, batch_size=32),
    epochs=25,  # Increase epochs for better convergence
    validation_data=(validation_images_normalized, validation_labels),
    shuffle=True,
)
```

Model 2:

- An enhanced CNN model using LeakyReLU activation with L2 regularization and a higher learning rate (1e-3).
- Features three convolutional layers with larger filters (128, 256, 256) and kernels (5x5 for the first layer).
- Includes batch normalization, LeakyReLU activations, and MaxPooling layers for spatial reduction.
- Utilizes Global Average Pooling for dimensionality reduction and a dense layer with 512 neurons and dropout (0.3).
- Incorporates training callbacks for early stopping and learning rate adjustment.

Model 2 enhances the baseline by introducing more sophisticated techniques. It swaps ReLU for LeakyReLU activation to mitigate the dying ReLU problem and incorporates L2

regularization to reduce overfitting. Larger filters (128 → 256) and a 5x5 kernel in the first layer capture more spatial context, while a higher learning rate (1e-3) speeds up convergence. A dense layer with 512 neurons and a slightly lower dropout rate (30%) balances capacity and regularization. Additionally, training callbacks for early stopping and learning rate adjustments are introduced, making this model more adaptive and capable of handling increased complexity compared to Model 1.

```
model_2 = models.Sequential()

# First Convolutional Layer with larger filters and Adam optimizer
model_2.add(layers.Conv2D(128, (5, 5), activation='linear', kernel_regularizer=regularizers.l2(0.001), input_shape=(32, 32, 3)))
model_2.add(layers.LeakyReLU(alpha=0.1))
model_2.add(layers.BatchNormalization())
model_2.add(layers.MaxPooling2D((2, 2)))

# Second Convolutional Layer
model_2.add(layers.Conv2D(256, (3, 3), activation='linear', kernel_regularizer=regularizers.l2(0.001)))
model_2.add(layers.LeakyReLU(alpha=0.1))
model_2.add(layers.BatchNormalization())
model_2.add(layers.MaxPooling2D((2, 2)))

# Third Convolutional Layer with more filters
model_2.add(layers.Conv2D(256, (3, 3), activation='linear', kernel_regularizer=regularizers.l2(0.001)))
model_2.add(layers.LeakyReLU(alpha=0.1))
model_2.add(layers.BatchNormalization())
model_2.add(layers.MaxPooling2D((2, 2)))

# Add a global average pooling layer
model_2.add(layers.GlobalAveragePooling2D())

# Fully Connected Layer with increased neurons
model_2.add(layers.Dense(512, activation='linear', kernel_regularizer=regularizers.l2(0.001)))
model_2.add(layers.LeakyReLU(alpha=0.1))  # Leaky ReLU
model_2.add(layers.Dropout(0.3))  # Reduced dropout rate

# Output Layer
model_2.add(layers.Dense(6, activation='softmax'))  # Assuming 6 classes

# Compile the model with Adam optimizer
model_2.compile(optimizer=Adam(learning_rate=1e-3),  # Using Adam with a higher learning rate
                loss='categorical_crossentropy',
                metrics=['accuracy'])

# Print the model summary
model_2.summary()

# Define callbacks for better training control
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=2, min_lr=1e-6)

# Train the model
history_2 = model_2.fit(
    datagen.flow(train_images_normalized, train_labels, batch_size=32),
    epochs=25,
    validation_data=(validation_images_normalized, validation_labels),
    shuffle=True,
    callbacks=[early_stopping, reduce_lr]
)
```

Model 3:

- A deeper CNN architecture with 4 convolutional layers (filters: 64, 128, 256, 512), ReLU activation, and L2 regularization.

- Batch normalization is included after each convolutional layer, and the fourth layer omits pooling to retain more spatial details.
- A Global Average Pooling layer reduces the dimensionality before a dense layer with 256 neurons and dropout (0.6).
- Data augmentation (rotation, shift, flip, etc.) is employed to increase the diversity of training samples.
- Uses training callbacks for early stopping and learning rate reduction.

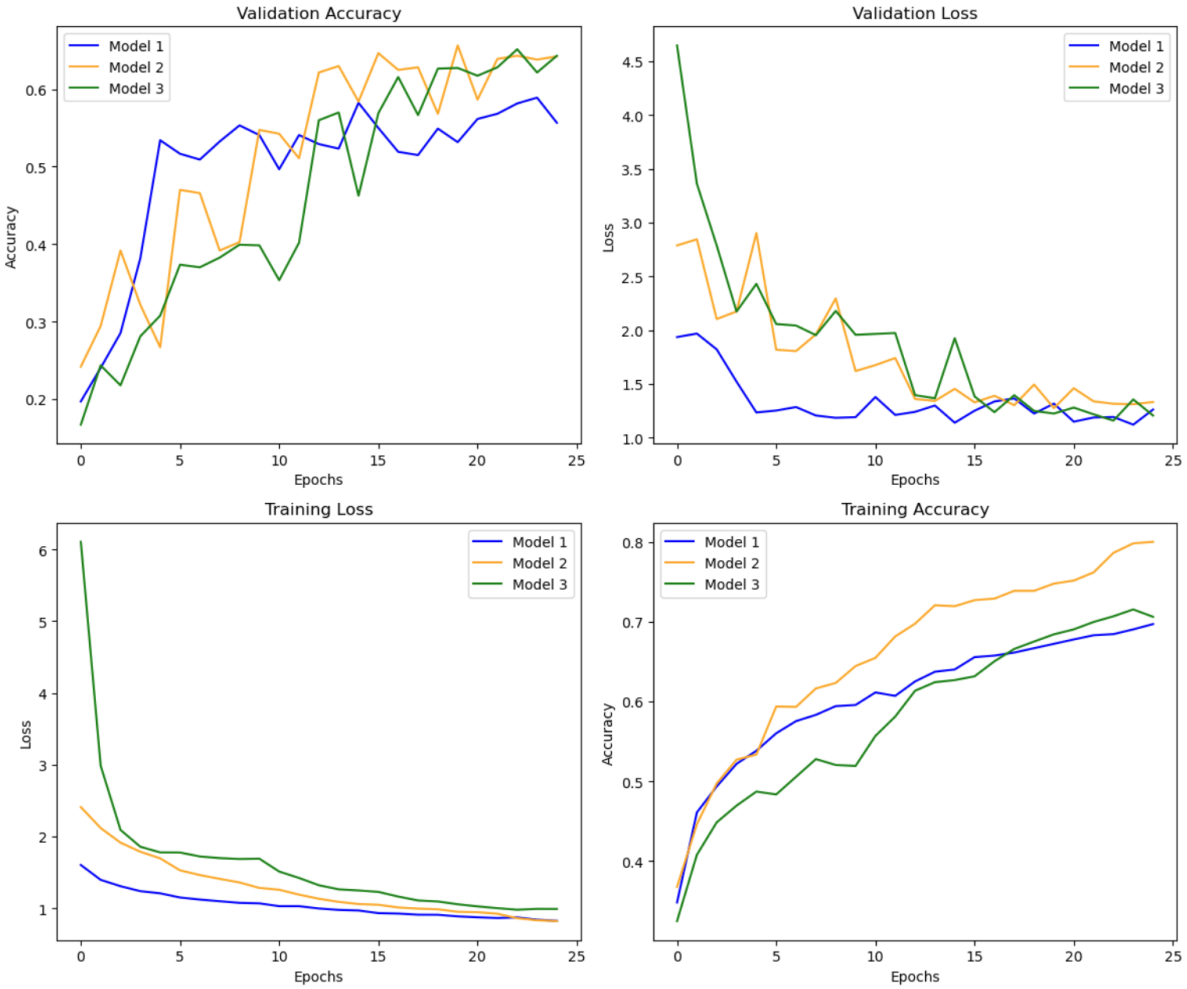
Model 3 explores a deeper architecture, with four convolutional layers scaling up to 512 filters, designed to capture more detailed hierarchical features. Unlike the others, it omits pooling in the final layer to retain more spatial information, which is critical for fine-grained feature learning. Data augmentation (e.g., rotations, flips, and shifts) is employed to improve generalization, while a dense layer with 256 neurons and a higher dropout rate (60%) combats overfitting. L2 regularization remains to manage the increased complexity, and training callbacks ensure dynamic adjustments during learning. Compared to Models 1 and 2, Model 3 is the most advanced, leveraging depth and augmentation to maximize feature extraction.

```python
import tensorflow as tf
from tensorflow.keras import layers, models, regularizers
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau

# Define Model 3 - Updated Architecture
model_3 = models.Sequential()

# First Convolutional Layer
model_3.add(layers.Conv2D(64, (3, 3), strides=(1, 1), activation='relu', kernel_regularizer=regularizers.l2(0.01), padding='same', input_shape=(32, 3
model_3.add(layers.BatchNormalization())
model_3.add(layers.MaxPooling2D((2, 2)))

# Second Convolutional Layer
model_3.add(layers.Conv2D(128, (3, 3), strides=(1, 1), activation='relu', kernel_regularizer=regularizers.l2(0.01), padding='same'))
model_3.add(layers.BatchNormalization())
model_3.add(layers.MaxPooling2D((2, 2)))

# Third Convolutional Layer
model_3.add(layers.Conv2D(256, (3, 3), strides=(1, 1), activation='relu', kernel_regularizer=regularizers.l2(0.01), padding='same'))
model_3.add(layers.BatchNormalization())
model_3.add(layers.MaxPooling2D((2, 2)))

# Fourth Convolutional Layer
model_3.add(layers.Conv2D(512, (3, 3), strides=(1, 1), activation='relu', kernel_regularizer=regularizers.l2(0.01), padding='same'))
model_3.add(layers.BatchNormalization())

# Global Average Pooling Layer
model_3.add(layers.GlobalAveragePooling2D())

# Fully Connected (Dense) Layer
model_3.add(layers.Dense(256, activation='relu'))
model_3.add(layers.Dropout(0.6))  # Increased dropout to 0.6

# Output Layer
model_3.add(layers.Dense(6, activation='softmax'))  # Assuming 6 classes

# Compile the model
model_3.compile(optimizer=Adam(learning_rate=1e-3),  # Higher initial learning rate
                loss='categorical_crossentropy',
                metrics=['accuracy'])

# Print model summary
model_3.summary()

# Define callbacks for better training control
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=2, min_lr=1e-6)

# Data Augmentation
datagen = tf.keras.preprocessing.image.ImageDataGenerator(
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)

# Train the model
history_3 = model_3.fit(
    datagen.flow(train_images_normalized, train_labels, batch_size=32),
    epochs=25,  # Set to 25 epochs
    validation_data=(validation_images_normalized, validation_labels),
    shuffle=True,
    callbacks=[early_stopping, reduce_lr]
)
```

The three models show variation in key hyperparameters and architectural decisions (e.g., activation functions, optimizers, number of convolutional layers, regularization techniques, and dropout rates).

Model 1 meets the baseline requirements with ReLU, Adam, and batch size 32, while **Models 2 and 3** explore more advanced techniques, including LeakyReLU, larger kernels, and deeper architectures.

All models incorporate batch normalization, shuffling, dropout, and regularization, ensuring robust training to prevent overfitting.



Over the course of 25 epochs, Model 1 exhibits a gradual improvement in both training and validation accuracy, though it demonstrates signs of overfitting. The training accuracy increases from 29.85% at epoch 1 to 69.87% at epoch 25, while the validation accuracy improves from 19.67% to 55.67%. While the training accuracy consistently increases, the validation accuracy plateaus after reaching its peak around epoch 24, indicating the model may not be generalizing well to unseen data. The training loss steadily decreases from 1.7064 to 0.8068, and the validation loss shows similar trends, reducing from 1.9344 to 1.2606, with some fluctuation in the later epochs. This suggests that while the model is improving on the training data, its ability to generalize is limited. The model could benefit from additional techniques like regularization, early stopping, or data augmentation to prevent overfitting and improve performance on the validation set.

Talking about model 2, the model shows steady improvements in both training accuracy and loss across epochs, with training accuracy increasing from 32% at Epoch 1 to around 80% by Epoch 25, and training loss decreasing from 2.61 to 0.82, indicating effective learning. The validation accuracy fluctuates but generally trends upward, reaching about

64% by the end of training, while the validation loss decreases, suggesting good generalization.

Last, model 3 the model shows substantial improvement over the 25 epochs, with training accuracy rising from 28.83% at Epoch 1 to 71.38% at Epoch 25, and training loss decreasing from 7.22 to 0.98, indicating progress in learning. Validation accuracy also increases from 16.67% at the start to 64.33% by the end, although some fluctuations are observed. The validation loss decreases from 4.65 to 1.20, suggesting better generalization. The learning rate decreases progressively, a typical strategy to refine the model's performance toward the end of training. Despite some variations in validation accuracy, the model demonstrates consistent improvement overall.