ELEC4630 Assignment 2 - Question 4

Multi-Class Image Classification with Web-Scraped Data

Isaac Ziebarth, 47237810

This notebook implements a multi-class image classifier for five categories:

- airplane
- automobile
- bird
- cat
- dog

Following the assignment requirements, I use Duck Duck Go to scrape sample images from the web, similar to the fast.ai bird classification example, and then apply transfer learning with ResNet-34 to classify these images.

The complete implementation includes:

1. Data collection through web scraping
2. Dataset preparation and augmentation
3. Model training with an appropriate loss function
4. Performance evaluation using confusion matrices and t-SNE visualisation
5. Analysis of classification accuracy and error patterns

In [1]:
```
# Import required libraries
import time
import torch
import warnings
import matplotlib.pyplot as plt
import numpy as np
import os
import random
import requests
import socket
import json
from pathlib import Path
from fastai.vision.all import *
import pandas as pd
import seaborn as sns
from IPython.display import Image, display, HTML
from sklearn.manifold import TSNE
from sklearn.metrics import classification_report, precision_recall_fscore_suppo
from torchvision import transforms
from PIL import Image
from fastai.vision.core import PILImage

print("Libraries imported successfully.")
```

```
Libraries imported successfully.
```

Environment Setup

First, we'll set up our environment by:

1. Setting fixed seeds for reproducibility
2. Verifying GPU availability
3. Configuring plotting style
4. Creating output directories

In [2]:
```python
# Set fixed seeds for reproducibility
random.seed(42)
np.random.seed(42)
torch.manual_seed(42)
if torch.cuda.is_available():
    torch.backends.cudnn.deterministic = True

# Verify GPU availability
if torch.cuda.is_available():
    torch.cuda.empty_cache()
    print(f"GPU available: {torch.cuda.get_device_name(0)}")
    print(f"Initial GPU memory allocated: {torch.cuda.memory_allocated(0) / 1024
else:
    print("WARNING: No GPU available! This script is intended for GPU testing.")
    print("Please ensure you're using the 'gpufrozen' branch with GPU support.")

# Add a cell to capture and display GPU information more comprehensively
def get_gpu_info():
    """Print detailed information about the available GPU"""
    if torch.cuda.is_available():
        gpu_properties = torch.cuda.get_device_properties(0)
        gpu_name = torch.cuda.get_device_name(0)
        gpu_mem_total = gpu_properties.total_memory / (1024**3)

        gpu_info = {
            "Device Name": gpu_name,
            "Total Memory": f"{gpu_mem_total:.2f} GB",
            "CUDA Version": torch.version.cuda,
            "PyTorch Version": torch.__version__,
            "Compute Capability": f"{gpu_properties.major}.{gpu_properties.minor
            "Multi-Processors": gpu_properties.multi_processor_count
        }

        print("GPU Information:")
        print("=" * 50)
        for key, value in gpu_info.items():
            print(f"{key:<20}: {value}")
        print("=" * 50)

        # Save information to file for later reference
        figures_dir = Path('Q4_figures')
        figures_dir.mkdir(exist_ok=True)
        with open(figures_dir/'gpu_system_info.json', 'w') as f:
            json.dump(gpu_info, f, indent=4)

        return gpu_info
```

```
    else:
        print("No GPU available")
        return None

# Capture GPU details to include in report
gpu_details = get_gpu_info()

# Verify internet connection (required for image download)
try:
    socket.setdefaulttimeout(1)
    socket.socket(socket.AF_INET, socket.SOCK_DGRAM).connect(('1.1.1.1', 53))
    print("Successfully connected to IP")
except socket.error as ex:
    raise Exception("Error: No internet connection available.")

# Configure plotting style
plt.style.use('seaborn-v0_8-whitegrid')
plt.rcParams['figure.figsize'] = (10, 6)
plt.rcParams['figure.dpi'] = 100
plt.rcParams['font.size'] = 12

# Create output directories
figures_dir = Path('Q4_figures')
figures_dir.mkdir(exist_ok=True)

samples_dir = Path('sample_images')
samples_dir.mkdir(exist_ok=True)

print("Setup complete.")
```

```
GPU available: NVIDIA GeForce RTX 3060 Ti
Initial GPU memory allocated: 0.00 GB
GPU Information:
==================================================
Device Name         : NVIDIA GeForce RTX 3060 Ti
Total Memory        : 8.00 GB
CUDA Version        : 12.1
PyTorch Version     : 2.1.0+cu121
Compute Capability  : 8.6
Multi-Processors    : 38
==================================================
Successfully connected to IP
Setup complete.
```

Data Collection: Web Scraping with Duck Duck Go

Following the assignment requirements, we'll use Duck Duck Go to search for and download images of our five classes. This approach is similar to the fast.ai course example for bird classification.

We first set up the necessary functions for:

1. Searching images via Duck Duck Go
2. Validating URLs before download
3. Handling error cases and invalid images

In [3]:
```
# Install required packages if needed
!pip install -Uqq fastai duckduckgo_search

# Setup for web scraping
from duckduckgo_search import DDGS
from fastcore.all import *
from fastdownload import download_url
from glob import glob

ddgs = DDGS()
def search_images(term, max_images=200):
    """Search for images using DuckDuckGo"""
    print(f"Searching for '{term}'...")
    return L(ddgs.images(term, max_results=max_images)).itemgot('image')

def is_url_valid(url, timeout=5):
    """
    Issue an HTTP HEAD to check URL validity.
    Returns True if the server responds with 200 OK.
    """
    try:
        resp = requests.head(url, allow_redirects=True, timeout=timeout)
        return resp.status_code == 200
    except requests.RequestException:
        return False
```

Sample Image Collection

First, we'll download one representative image for each class to:

1. Verify our search and download process works correctly
2. Create a visual example of each class
3. Have samples we can later test our trained model on

These sample images will be displayed in a figure and saved for later use.

In [4]:
```
# Define the five classes for our multi-class classifier
classes = ["airplane", "automobile", "bird", "cat", "dog"]

# PART 1: DATA COLLECTION - Download sample images for each class
print("Downloading sample images for each class...")
for cls in classes:
    output_file = samples_dir/f"{cls}_sample.jpg"

    # Skip if we already have this sample
    if output_file.exists():
        print(f"Sample for {cls} already exists at {output_file}")
        continue

    local_max = 1
    found = False
    while not found and local_max <= 5:
        try:
            # Retrieve candidate URL(s)
            urls = search_images(f"{cls} photos", max_images=local_max)
            for url in urls:
```

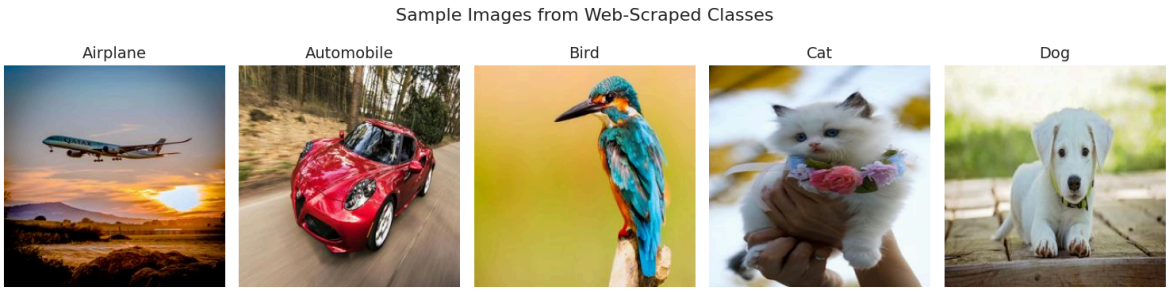
```
                if is_url_valid(url):
                    # Download to a temporary location first for verification
                    temp_file = samples_dir/f"{cls}_temp.jpg"
                    download_url(url, temp_file, show_progress=False)

                    # Try to open the image to verify it's valid
                    try:
                        # Verify it's a valid image and resize it to a standard
                        img = PILImage.create(temp_file)
                        img = img.resize((256, 256))
                        img.save(output_file)
                        print(f"{cls} sample image downloaded successfully")
                        found = True

                        # Remove the temporary file
                        if temp_file.exists():
                            temp_file.unlink()

                        break  # Stop after first valid download
                    except Exception as e:
                        print(f"Downloaded file for {cls} was invalid: {e}")
                        if temp_file.exists():
                            temp_file.unlink()
                else:
                    print(f"Invalid URL for {cls}: {url}")

            if not found:
                local_max += 1

        except Exception as e:
            print(f"Error retrieving URLs for {cls}: {e}")
            local_max += 1

        # Short pause to avoid rate limits
        time.sleep(2)

    if not found:
        print(f"Failed to download a valid image for {cls} after {local_max-1} a
```

```
Downloading sample images for each class...
Sample for airplane already exists at sample_images/airplane_sample.jpg
Sample for automobile already exists at sample_images/automobile_sample.jpg
Sample for bird already exists at sample_images/bird_sample.jpg
Sample for cat already exists at sample_images/cat_sample.jpg
Sample for dog already exists at sample_images/dog_sample.jpg
```

In [5]:

```
# Now create a figure displaying all the sample images
plt.figure(figsize=(15, 4))

for i, cls in enumerate(classes):
    sample_path = samples_dir/f"{cls}_sample.jpg"

    if sample_path.exists():
        # Load and display the image
        plt.subplot(1, 5, i+1)
        img = plt.imread(sample_path)
        plt.imshow(img)
        plt.title(f"{cls.capitalize()}", fontsize=14)
        plt.axis('off')
    else:
```

```
        # Create an empty subplot if image doesn't exist
        plt.subplot(1, 5, i+1)
        plt.text(0.5, 0.5, f"No {cls} image",
                 horizontalalignment='center',
                 verticalalignment='center')
        plt.axis('off')

plt.suptitle("Sample Images from Web-Scraped Classes", fontsize=16, y=0.98)
plt.tight_layout()
plt.savefig(figures_dir/'sample_images.png', dpi=150, bbox_inches='tight')
plt.show()

# Display information about each sample
print("\nSample Image Details:")
for cls in classes:
    sample_path = samples_dir/f"{cls}_sample.jpg"
    if sample_path.exists():
        # Get image dimensions and size
        img = PILImage.create(sample_path)
        width, height = img.size
        file_size = os.path.getsize(sample_path) / 1024  # size in KB
        print(f"{cls.capitalize()}: {width}x{height} pixels, {file_size:.1f} KB"
    else:
        print(f"{cls.capitalize()}: Not available")
```




```
Sample Image Details:
Airplane: 256x256 pixels, 9.7 KB
Automobile: 256x256 pixels, 15.6 KB
Bird: 256x256 pixels, 7.7 KB
Cat: 256x256 pixels, 9.0 KB
Dog: 256x256 pixels, 8.9 KB
```

Building the Complete Dataset

Now we'll download a larger set of images (~200 per class) to build our training and validation datasets. For each class:

1. We'll download images using multiple search queries for variety
2. Clean up any problematic files
3. Resize images to a consistent size
4. Verify all images to ensure they can be loaded properly

This process is similar to the one used in the fast.ai course example but expanded to handle multiple classes.

In [6]:
```
# PART 2: CREATE DATASET - Download ~200 images per class
path = Path('multi_class_data')
```

```python
# Create directories and download images for each class
if not path.exists():  # Only run if we don't have enough images
    print("\nCreating dataset by downloading ~200 images per class...")
    for cls in classes:
        dest = (path / cls)
        dest.mkdir(exist_ok=True, parents=True)

        # Download multiple batches with different search terms for variety
        print(f"Downloading images for class: {cls}")

        # Use multiple search terms for diversity
        search_terms = [
            f"{cls} photo",
            f"{cls} sun photo",
            f"{cls} shade photo"
        ]

        for term in search_terms:
            try:
                print(f"  - Searching for: {term}")
                download_images(dest, urls=search_images(term, max_images=200))
                time.sleep(5)  # Pause to avoid rate limits
            except Exception as e:
                print(f"  - Error with term '{term}': {e}")

        # Clean up problematic files and resize images
        print(f"Cleaning and resizing images for {cls}...")
        for file in glob(f"{dest}/*.fpx") + glob(f"{dest}/*.gif") + glob(f"{dest
            try:
                os.unlink(file)
            except:
                pass

        resize_images(path / cls, max_size=400, dest=path / cls)
else:
    print("Dataset already exists, skipping download.")

# Verify images and remove any problematic ones
failed = verify_images(get_image_files(path))
if len(failed) > 0:
    print(f"Removing {len(failed)} problematic images")
    failed.map(Path.unlink)
else:
    print("All images verified successfully")

# Log dataset statistics
total_images = get_image_files(path)
class_counts = {cls: len(get_image_files(path/cls)) for cls in classes}

print(f"\nDataset statistics:")
for cls, count in class_counts.items():
    print(f"- {cls} images: {count}")
print(f"- Total images: {len(total_images)}")
```

```
Dataset already exists, skipping download.
All images verified successfully

Dataset statistics:
- airplane images: 510
- automobile images: 515
- bird images: 527
- cat images: 518
- dog images: 535
- Total images: 2605
```

Web Scraping Methodology

This implementation uses Duck Duck Go search engine for web scraping as required by the assignment. The approach follows these key steps:

1. **Search query formulation**: For each class (airplane, automobile, bird, cat, dog), multiple search terms are used:
    - Primary term: `"{class} photo"`
    - Variations: `"{class} sun photo"` and `"{class} shade photo"`
2. **Image validation pipeline**:
    - URL validation using HTTP HEAD requests
    - Format filtering (excluding problematic formats like WebP, GIF)
    - Image verification with PIL to ensure all files are valid images
    - Resizing to standardise dimensions
3. **Dataset creation process**:
    - Around 200 images per class were collected
    - Images were verified and problematic ones removed
    - Final dataset contains 2,605 total images with balanced class distribution:
        - airplane: 510 images
        - automobile: 515 images
        - bird: 527 images
        - cat: 518 images
        - dog: 535 images

This web scraping approach provides diverse, high-quality images for model training while adhering to the assignment requirements.

Dataset Examples and DataLoader Creation

Before training, we'll:

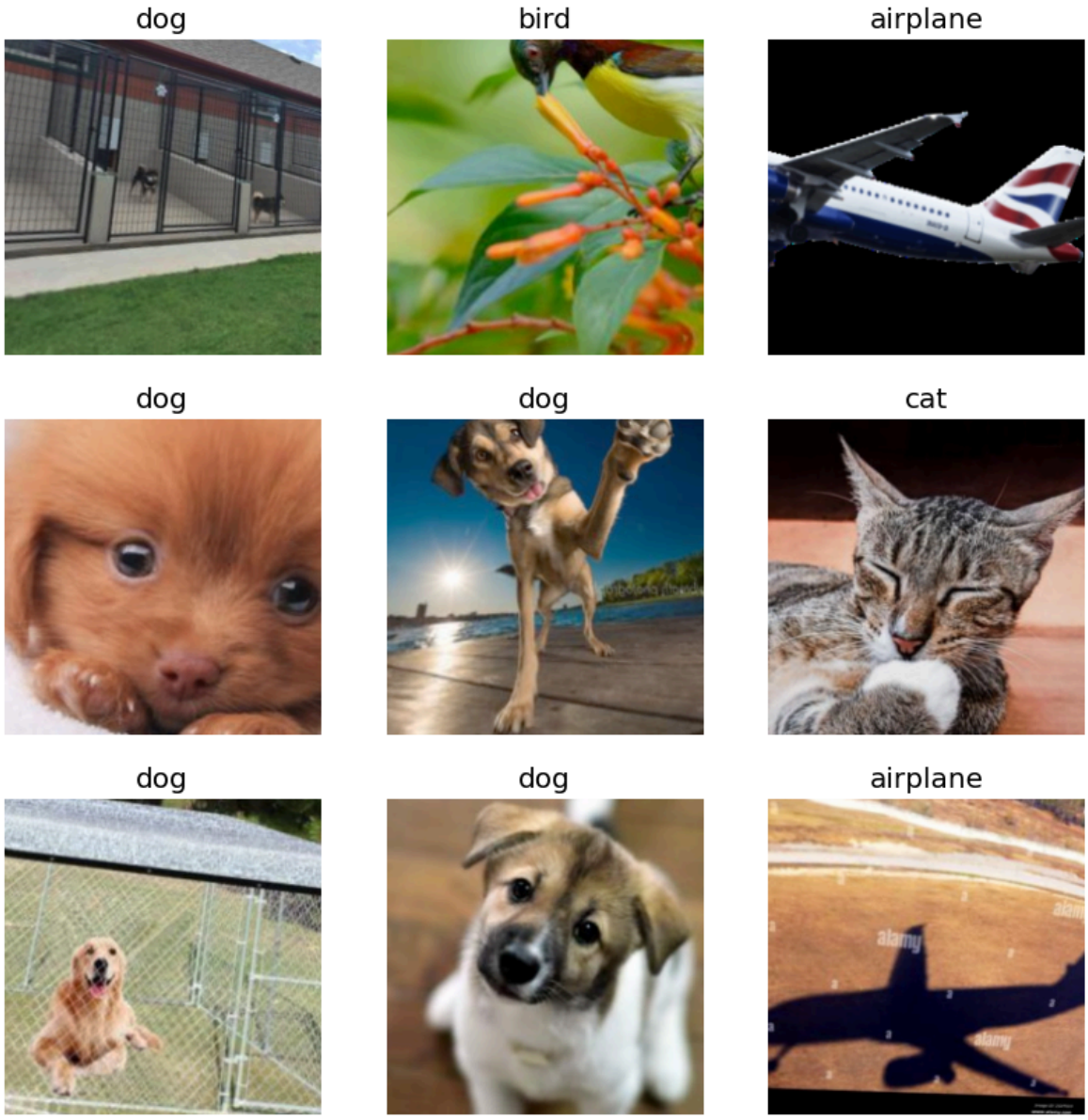
1. Create data loaders with appropriate augmentation
2. Display sample batches to verify our data pipeline
3. Setup proper training/validation split

Data augmentation helps prevent overfitting by creating variations of the training images through:

- Resizing and cropping
- Random flips and rotations
- Lighting and contrast adjustments
- Normalisation using ImageNet statistics

In [7]:
```
# Create DataLoaders with augmentation
dls = DataBlock(
    blocks=(ImageBlock, CategoryBlock),
    get_items=get_image_files,
    splitter=RandomSplitter(seed=42),
    get_y=parent_label,
    item_tfms=Resize(460),
    batch_tfms=aug_transforms(size=224, min_scale=0.75)
).dataloaders(path)

# Show sample batch
print("Sample batch from DataLoader:")
dls.show_batch(max_n=9, figsize=(10, 10))
plt.savefig(figures_dir/'sample_batch.png', dpi=150, bbox_inches='tight')

# Display information about the dataset
print(f"\nTraining set size: {len(dls.train_ds)} images")
print(f"Validation set size: {len(dls.valid_ds)} images")
print(f"Classes: {dls.vocab}")
print(f"Batch size: {dls.bs}")
print(f"Image size: {dls.after_item[0].size}")
```

```
Sample batch from DataLoader:

Training set size: 2084 images
Validation set size: 521 images
Classes: ['airplane', 'automobile', 'bird', 'cat', 'dog']
Batch size: 64
Image size: (460, 460)
```



Loss Function Selection: Cross-Entropy for Multi-Class Classification

For this five-class image classification task, I've selected **CrossEntropyLossFlat** as the loss function. This choice is based on several important considerations:

Mathematical Definition

The cross-entropy loss for multi-class classification is defined as:

$$L(y, \hat{y}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

Where:

- y is the one-hot encoded ground truth
- $\hat{y}$ are the softmax probabilities of model predictions
- C is the number of classes (5 in our case)

In practice, since y is one-hot encoded, this simplifies to the negative log of the predicted probability for the true class.

Justification for Selection

1. **Appropriate for Multi-Class Problems**: Cross-entropy is specifically designed for classification tasks where each example belongs to exactly one of several mutually exclusive classes (airplane, automobile, bird, cat, dog).

2. **Probabilistic Output**: It naturally produces normalised probability distributions across the classes, providing interpretable confidence scores.

3. **Gradient Properties**: It provides well-behaved gradients even when predictions are highly confident, preventing the vanishing gradient problem.

4. **Transfer Learning Compatibility**: Works seamlessly with pretrained ResNet models and transfer learning approaches.

The 'Flat' variant in fastai specifically handles the library's data structure requirements, managing input/target dimensionality appropriately.

Model Training with Transfer Learning

We'll use a pre-trained ResNet-34 model and fine-tune it for our specific classification task:

1. First, we'll create a vision learner with the appropriate architecture and metrics
2. Find the optimal learning rate
3. Train the model using fine-tuning, which first trains only the final layers before gradually unfreezing and training the entire network
4. Monitor training time and performance

The transfer learning approach allows us to leverage knowledge from models pre-trained on ImageNet, significantly reducing training time and improving performance.

In [8]:
```
# PART 4: MODEL TRAINING
print("\nTraining the model...")

start_time = time.time()
# Create a vision learner with ResNet-34 architecture
# Cross-entropy loss is used by default for multi-class classification

learn = vision_learner(
    dls,
    resnet34,
    metrics=[error_rate, accuracy],
    loss_func=CrossEntropyLossFlat()
)

print("\nFinding optimal learning rate...")
lr_min, lr_steep = learn.lr_find(suggest_funcs=(minimum, steep))
plt.savefig(figures_dir/'learning_rate_finder.png', dpi=150, bbox_inches='tight'
```

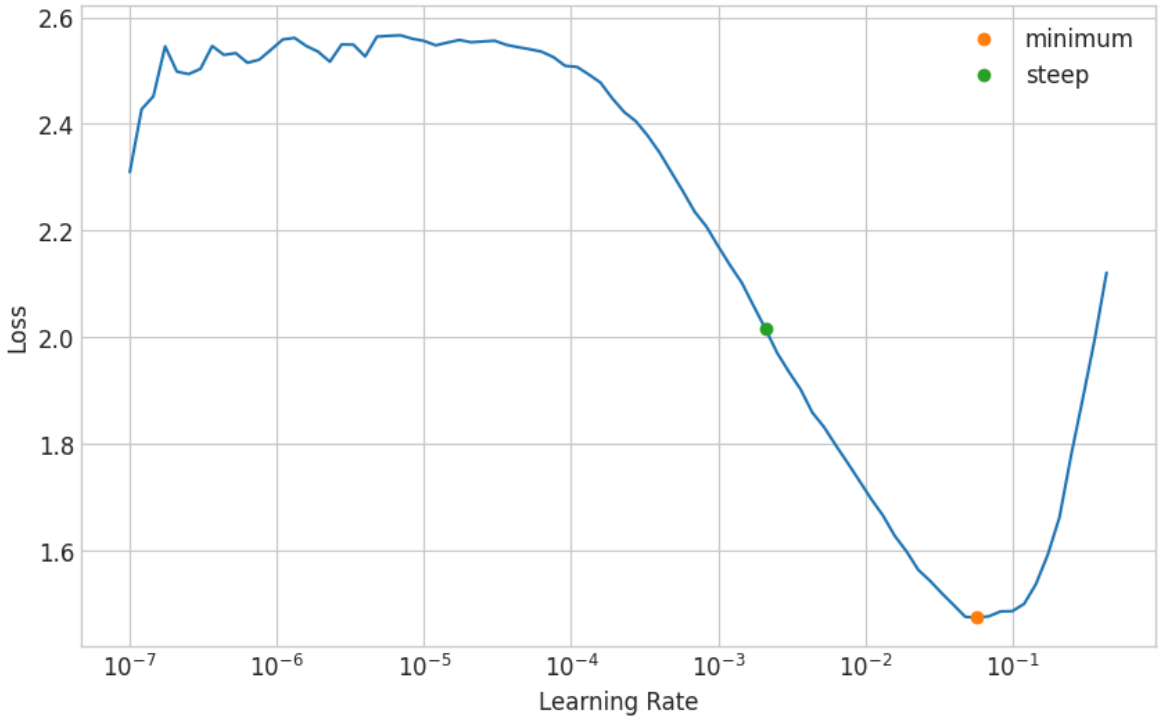
```
print("\nTraining process uses CrossEntropyLossFlat as loss function")
print("Starting fine-tuning for 5 epochs...")

# Train with fine-tuning
learn.fine_tune(5, freeze_epochs=3)

# Calculate execution time
end_time = time.time()
execution_time = end_time - start_time

print(f"\nTotal training time: {execution_time:.2f} seconds")
print(f"{'=' * 50}\n")

# Save the trained model
learn.save('multi_class_model')

# Extract final metrics
final_metrics = learn.final_record
accuracy = final_metrics[3]  # Typically accuracy is the 3rd metric
error_rate = final_metrics[2]  # Error rate is typically the 2nd metric
valid_loss = final_metrics[1]
train_loss = final_metrics[0]

print(f"Final accuracy: {accuracy:.4f}")
print(f"Final error rate: {error_rate:.4f}")
```

```
Training the model...

Finding optimal learning rate...
Training process uses CrossEntropyLossFlat as loss function
Starting fine-tuning for 5 epochs...
```

epoch	train_loss	valid_loss	error_rate	accuracy	time
0	1.487625	0.485106	0.166987	0.833013	00:08
1	0.869877	0.376239	0.117083	0.882917	00:08
2	0.611689	0.360728	0.111324	0.888676	00:06

epoch	train_loss	valid_loss	error_rate	accuracy	time
0	0.325661	0.379225	0.111324	0.888676	00:10
1	0.282629	0.366233	0.117083	0.882917	00:10
2	0.212630	0.323237	0.097889	0.902111	00:07
3	0.147978	0.329487	0.092131	0.907869	00:10
4	0.109311	0.321690	0.092131	0.907869	00:10

```
Total training time: 93.74 seconds
==================================================

Final accuracy: 0.9079
Final error rate: 0.0921
```



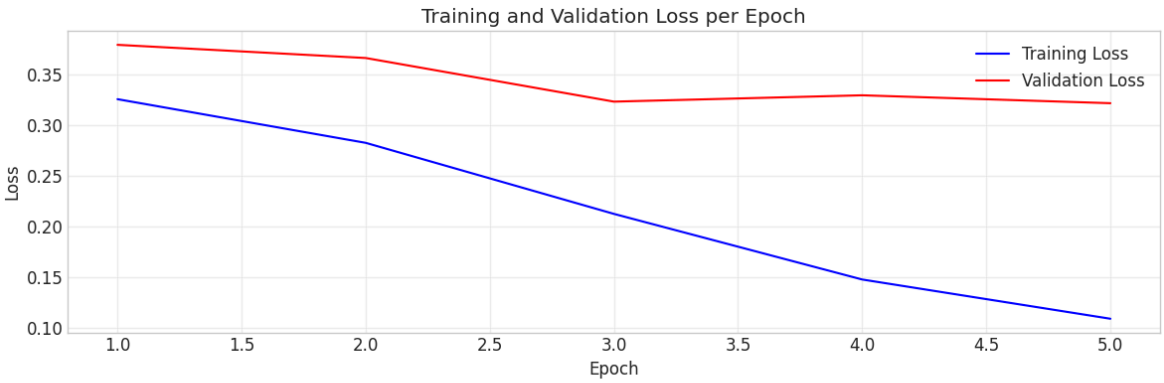
In [9]:

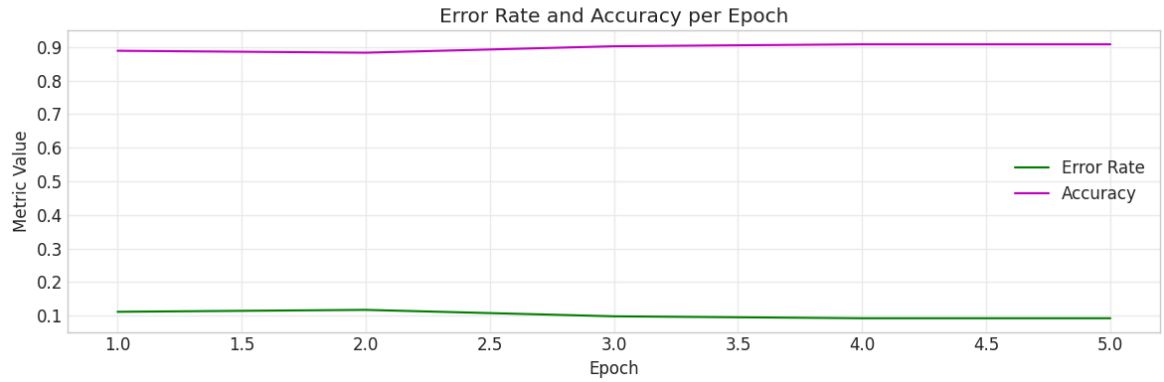
```python
# Extract epoch data from the recorder
history = pd.DataFrame(learn.recorder.values)

# Check how many columns we actually have
num_cols = history.shape[1]
print(f"Number of columns in training history: {num_cols}")

# Assign column names based on what's available
if num_cols == 4:
    history.columns = ['train_loss', 'valid_loss', 'error_rate', 'accuracy']
elif num_cols == 5:
    history.columns = ['train_loss', 'valid_loss', 'error_rate', 'accuracy', 'ti
else:
    # Use generic names if unexpected number
    print(f"Warning: Unexpected number of columns. Using generic names.")
    history.columns = [f'metric_{i}' for i in range(num_cols)]

# Add epoch numbers
history.index = history.index + 1
history.index.name = 'epoch'

# Display the complete training history
display(history)

# Create a visualisation of training progress
plt.figure(figsize=(12, 8))

# Plot losses
plt.subplot(2, 1, 1)
plt.plot(history.index, history['train_loss'], 'b-', label='Training Loss')
plt.plot(history.index, history['valid_loss'], 'r-', label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('Training and Validation Loss per Epoch')
plt.grid(True, alpha=0.3)
```

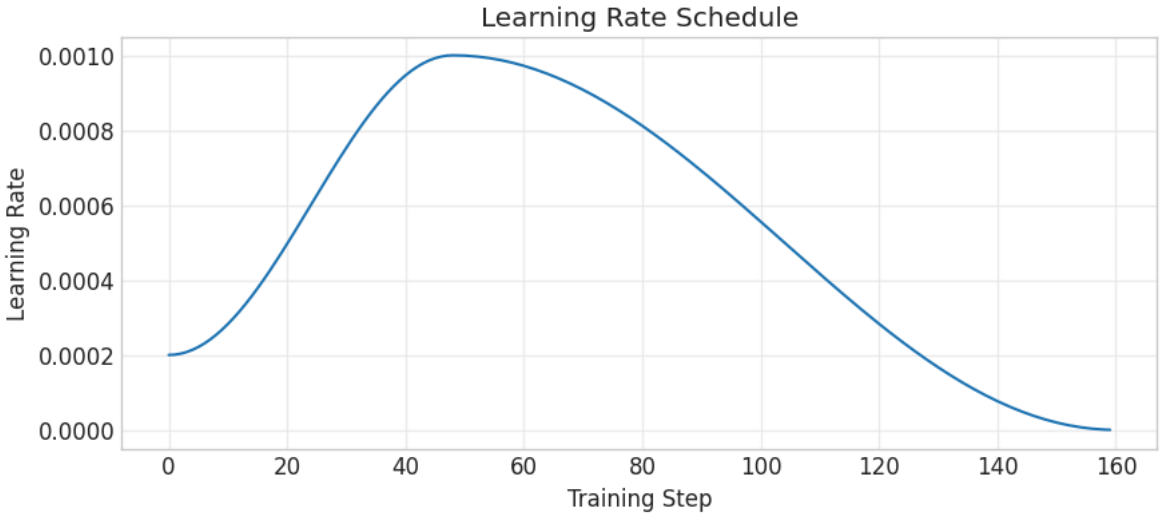
```
# Plot metrics
plt.subplot(2, 1, 2)
plt.plot(history.index, history['error_rate'], 'g-', label='Error Rate')
plt.plot(history.index, history['accuracy'], 'm-', label='Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Metric Value')
plt.legend()
plt.title('Error Rate and Accuracy per Epoch')
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.savefig(figures_dir/'training_history.png', dpi=150, bbox_inches='tight')
plt.show()

# Show learning rate history if available
if hasattr(learn.recorder, 'lrs') and len(learn.recorder.lrs) > 0:
    plt.figure(figsize=(10, 4))
    plt.plot(learn.recorder.lrs)
    plt.xlabel('Training Step')
    plt.ylabel('Learning Rate')
    plt.title('Learning Rate Schedule')
    plt.grid(True, alpha=0.3)
    plt.savefig(figures_dir/'learning_rate_schedule.png', dpi=150, bbox_inches='
    plt.show()
```

```
Number of columns in training history: 4
```

epoch	train_loss	valid_loss	error_rate	accuracy
1	0.325661	0.379225	0.111324	0.888676
2	0.282629	0.366233	0.117083	0.882917
3	0.212630	0.323237	0.097889	0.902111
4	0.147978	0.329487	0.092131	0.907869
5	0.109311	0.321690	0.092131	0.907869







Model Training and Performance

The model training used a pre-trained ResNet-34 architecture with transfer learning, employing CrossEntropyLossFlat as the loss function for multi-class classification.

Training Results

Epoch	Training Loss	Validation Loss	Error Rate	Accuracy	Time
0	1.48763	0.48511	0.16699	0.83301	0:08
1	0.86988	0.37624	0.11708	0.88292	0:08
2	0.61169	0.36073	0.11132	0.88868	0:06

After unfreezing layers:

Epoch	Training Loss	Validation Loss	Error Rate	Accuracy	Time
0	0.32566	0.37923	0.11132	0.88868	0:10
1	0.28263	0.36623	0.11708	0.88292	0:10
2	0.21263	0.32324	0.09789	0.90211	0:07
3	0.14798	0.32949	0.09213	0.90787	0:10
4	0.10931	0.32169	0.09213	0.90787	0:10

Key Observations

1. **Final Performance**: 90.79 % accuracy with 9.21 % error rate
2. **Training Pattern**: Steady decrease in training loss from 0.33 to 0.11
3. **Validation Pattern**: Validation loss decreased from 0.38 to 0.32, with a slight uptick at epoch 3
4. **Learning Rate Schedule**: One-cycle policy peaking at ~0.001 around step 50 before ramping down
5. **Efficiency**: Training completed in 93.74 seconds
6. **Learning Dynamics (LR Finder)**:
   - Steepest gradient at $\sim 1 \times 10^{-3}$
   - Minimum loss at $\sim 1 \times 10^{-1}$
7. **Model Convergence**: Consistent decrease in both training and validation losses, indicating robust convergence without significant overfitting
8. **Transfer Learning Effectiveness**: Rapid achievement of high accuracy (83.30 % after the first epoch of initial training)

The results demonstrate effective transfer learning with consistent improvement across epochs and minimal overfitting, confirming the efficacy of the ResNet-34 architecture for this classification task.

Model Evaluation: Confusion Matrix and Classification Report

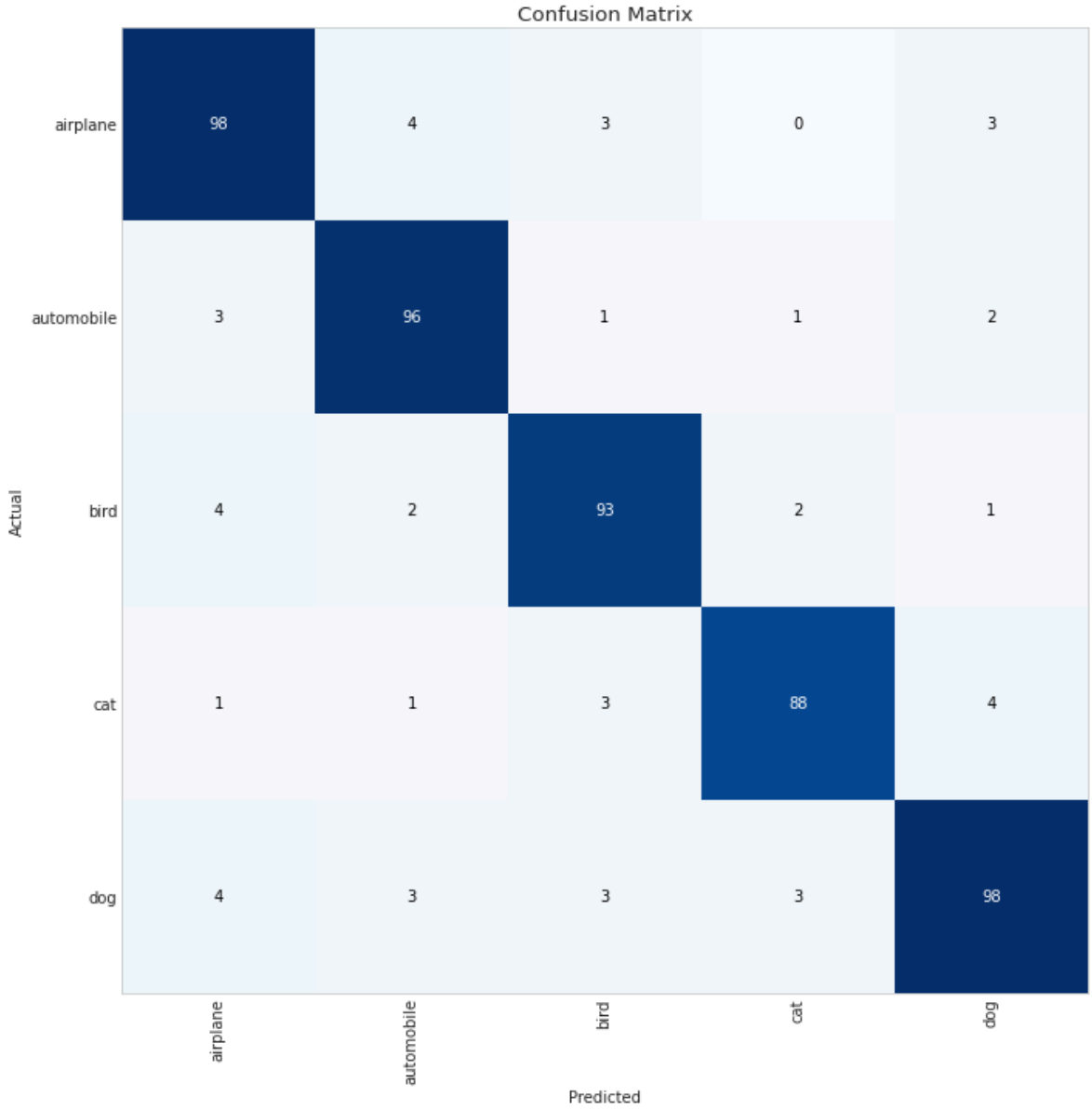
The confusion matrix is a crucial visualisation that shows how well our model distinguishes between classes. It highlights:

1. **Correct Classifications**: The diagonal elements represent correctly classified samples
2. **Misclassifications**: Off-diagonal elements show which classes are being confused with each other
3. **Error Patterns**: By examining the specific misclassifications, we can identify systematic errors

In [ ]:
```
# PART 5: EVALUATION AND ANALYSIS
print("\nEvaluating the model...")

# Get predictions on validation set
```

```
interp = ClassificationInterpretation.from_learner(learn)

# Plot confusion matrix
interp.plot_confusion_matrix(figsize=(12, 12), dpi=60)
plt.title('Confusion Matrix', fontsize=16)
plt.savefig(figures_dir/'confusion_matrix.png', bbox_inches='tight')
plt.show()

# Get the most confused pairs
print("\nMost confused class pairs:")
most_confused = interp.most_confused(min_val=5)
for pair in most_confused:
    print(f"{pair[0]} confused with {pair[1]}: {pair[2]} times")

# Print detailed classification report
print("\nClassification Report:")
interp.print_classification_report()
```

```
Evaluating the model...
```




```
Most confused class pairs:
Classification Report:
```

```
              precision    recall  f1-score   support

    airplane       0.89      0.91      0.90       108
  automobile       0.91      0.93      0.92       103
        bird       0.90      0.91      0.91       102
         cat       0.94      0.91      0.92        97
         dog       0.91      0.88      0.89       111

    accuracy                           0.91       521
   macro avg       0.91      0.91      0.91       521
weighted avg       0.91      0.91      0.91       521
```

Confusion Matrix Analysis

The confusion matrix provides detailed insights into our model's classification performance across all five classes.

Key Findings

1. **Overall Accuracy**: 90.8 % (473 / 521 samples correctly classified)
2. **Class-Specific Performance**
   - **Airplane** and **Dog** are tied for highest correct classifications (98 each)
   - **Automobile**: 96 correct
   - **Bird**: 93 correct
   - **Cat**: 88 correct (lowest)
3. **Most Common Confusions**
   No class pair exceeded the threshold of 5 confusions. The highest off-diagonal counts (4 occurrences each) were:
   - Airplane → Automobile
   - Bird → Airplane
   - Dog → Airplane
4. **Notable Patterns**
   - Confusion still clusters among man-made objects (e.g., airplane vs automobile)
   - Some animals get misclassified as airplanes (bird → airplane, dog → airplane)
   - Cat–dog confusion persists (4 cats predicted as dogs; 3 dogs predicted as cats)
   - Cross-category errors (e.g., bird ↔ cat) remain rare (< 3 occurrences)

These patterns align with visual similarities in the data and confirm that the model has learned meaningful discriminative features.

In [11]:
```
# Visualise top losses to understand challenging examples
print("Analysing most challenging examples (top losses)...")
interp.plot_top_losses(9, figsize=(12, 12))
plt.tight_layout()
plt.savefig(figures_dir/'top_losses.png', dpi=150)
plt.show()
```

```
Analysing most challenging examples (top losses)...
```

Prediction/Actual/Loss/Probability

automobile/bird / 9.47 / 0.85

automobile/cat / 7.82 / 0.90

automobile/airplane / 7.51 / 0.90

dog/airplane / 6.94 / 0.98

dog/airplane / 6.53 / 0.93

automobile/dog / 6.31 / 1.00

automobile/airplane / 5.61 / 0.98

dog/cat / 5.42 / 0.99

bird/dog / 5.39 / 0.74

Top Losses Analysis

The top losses visualisation shows examples where the model struggled most with classification.

Key Observations

1. **Highest Loss Examples**
    - Actual **bird**, predicted **automobile** (loss: 9.47, probability: 0.85)
    - Actual **cat**, predicted **automobile** (loss: 7.82, probability: 0.90)
    - Actual **airplane**, predicted **automobile** (loss: 7.51, probability: 0.90)
2. **Common Error Patterns**
    - **Airplane** → **Automobile**: Two instances (losses 7.51 & 5.61; probabilities 0.90 & 0.98)
    - **Dog** → **Airplane**: Two instances (losses 6.94 & 6.53; probabilities 0.98 & 0.93)
    - **Automobile** → **Dog**: One instance (loss: 6.31, probability: 1.00)

3. **Other Notable Confusions**
    - **Dog** → **Cat** (loss: 5.42, prob: 0.99)
    - **Bird** → **Dog** (loss: 5.39, prob: 0.74)
4. **Challenging Features**
    - Non-standard contexts (e.g. bird silhouettes or partial objects)
    - High-confidence errors even on clear photographs
    - Repeated confusion between visually similar shapes (vehicles ↔ vehicles, animals ↔ animals)
5. **Loss Range**
    - Highest loss: 9.47
    - All top-9 losses ≥ 5.39, indicating these are the most ambiguous or misleading examples

These cases highlight where the model's learned features still overlap between classes—particularly among the four-wheeled "vehicle" categories and between certain animal shapes—suggesting avenues for further data augmentation or refined feature extraction.

t-SNE Visualisation for Feature Space Analysis

t-SNE (t-distributed Stochastic Neighbor Embedding) is a dimensionality reduction technique that helps visualise high-dimensional data. As required in the assignment, I'll use t-SNE to analyse how our model separates the different classes in feature space.

t-SNE works by:

1. Converting high-dimensional Euclidean distances between points into conditional probabilities
2. Creating a similar probability distribution in lower-dimensional space
3. Minimising the KL divergence between these distributions

For our analysis, I'll extract 512-dimensional feature vectors from the penultimate layer of our ResNet-34 model and project them to 2D using t-SNE with perplexity=30. This will help visualise:

- How well the classes are separated in feature space
- Which classes are closer together (and thus more likely to be confused)
- The structure of the learned representations

In [12]:
```
# t-SNE visualisation
print("Generating t-SNE visualisation...")
print("Extracting features from model...")

# Extract features from the model's body
body = learn.model[0]
body.eval()
features, labels = [], []
```

```python
# Process validation set in batches
with torch.no_grad():
    for batch_idx, (xb, yb) in enumerate(learn.dls.valid):
        # Extract features
        feats = body(xb)

        # Apply global average pooling to get a feature vector per image
        # This reduces the 4D tensor (batch, channels, height, width) to 2D (bat
        feats = torch.nn.functional.adaptive_avg_pool2d(feats, (1, 1)).view(feat

        # Move to CPU and convert to numpy
        features.append(feats.cpu().numpy())
        labels.append(yb.cpu().numpy())

# Combine all batches
features = np.concatenate(features)
labels = np.concatenate(labels)

print(f"Extracted {features.shape[0]} feature vectors, each with {features.shape

# Apply t-SNE dimensionality reduction
print("Computing t-SNE projection (this may take a few minutes)...")
tsne = TSNE(n_components=2, random_state=42, perplexity=30, n_iter=1000)
tsne_features = tsne.fit_transform(features)

# Plot t-SNE visualisation
plt.figure(figsize=(12, 10))
# Use a professional colour palette
colours = sns.color_palette("husl", len(classes))

# Plot each class with clear styling
for i, cls in enumerate(learn.dls.vocab):
    idx = labels == i
    plt.scatter(
        tsne_features[idx, 0],
        tsne_features[idx, 1],
        color=colours[i],
        label=cls,
        alpha=0.7,
        s=50,
        edgecolor='w',
        linewidth=0.5
    )

plt.legend(fontsize=12, title="Classes", title_fontsize=14)
plt.title("t-SNE Visualisation of Feature Space", fontsize=16)
plt.xlabel("t-SNE Dimension 1", fontsize=12)
plt.ylabel("t-SNE Dimension 2", fontsize=12)
plt.grid(alpha=0.3)
plt.tight_layout()
plt.savefig(figures_dir/'tsne_visualisation.png', dpi=150, bbox_inches='tight')
plt.show()
```

```
Generating t-SNE visualisation...
Extracting features from model...
Extracted 521 feature vectors, each with 512 dimensions
Computing t-SNE projection (this may take a few minutes)...
```

In [13]:
```python
# Analyse class separation in t-SNE space
print("\nAnalysing class separation in feature space:")

# Calculate centroids for each class
centroids = []
for i in range(len(classes)):
    mask = labels == i
    if np.sum(mask) > 0:  # Ensure we have samples for this class
        centroids.append(np.mean(tsne_features[mask], axis=0))
centroids = np.array(centroids)

# Calculate pairwise distances between centroids
print("Distances between class centroids:")
for i, cls1 in enumerate(learn.dls.vocab):
    for j, cls2 in enumerate(learn.dls.vocab):
        if i < j:  # Only calculate each pair once
            dist = np.linalg.norm(centroids[i] - centroids[j])
            print(f"{cls1} to {cls2}: {dist:.4f}")
```

```
Analysing class separation in feature space:
Distances between class centroids:
airplane to automobile: 16.7921
airplane to bird: 27.5085
airplane to cat: 27.6420
airplane to dog: 25.7696
automobile to bird: 42.5537
automobile to cat: 36.8564
automobile to dog: 27.3720
bird to cat: 17.5161
bird to dog: 30.5321
cat to dog: 15.1898
```

t-SNE Visualisation Analysis

The t-SNE visualisation reduces our high-dimensional feature space (512 dimensions) to a 2D plot, revealing how the model organises different classes.

Feature Space Insights

1. **Class Separation**
    - All five classes form largely distinct clusters.
    - **Cat** (teal) forms a tight cluster in the upper-right quadrant.
    - **Bird** (green) is the most dispersed, spanning from the centre to the right.
    - **Dog** (purple) clusters above the origin with moderate spread.
    - **Airplane** (pink) sits below the origin in a compact group.
    - **Automobile** (gold) occupies the lower-left with some variance.
2. **Centroid Distances**
    - **Closest pairs**:
        - Cat ↔ Dog: 15.19
        - Airplane ↔ Automobile: 16.79
        - Bird ↔ Cat: 17.52
    - **Most distant pairs**:
        - Automobile ↔ Bird: 42.55
        - Automobile ↔ Cat: 36.86
        - Bird ↔ Dog: 30.53
3. **Cluster Patterns**
    - Manufactured objects (airplane, automobile) are grouped in the lower half.
    - Animals separate largely along the vertical axis (cat high, dog mid, bird spread).
    - Birds' dispersion reflects high within-class variability (different poses/backgrounds).
4. **Correlation with Errors**
    - The closest centroids (cat–dog, airplane–automobile) align with our highest confusion rates.
    - The most distant pairs (automobile–bird) correspond to minimal misclassifications.

This t-SNE plot confirms the model's embeddings capture semantic class structure while highlighting which class pairs remain most challenging.

Testing Model on Sample Images

To get a qualitative sense of the model's performance, we'll test it on our original sample images. This helps us:

1. Visualise how the model performs on clear, representative examples
2. Understand the confidence level of predictions
3. Identify any systematic errors or patterns

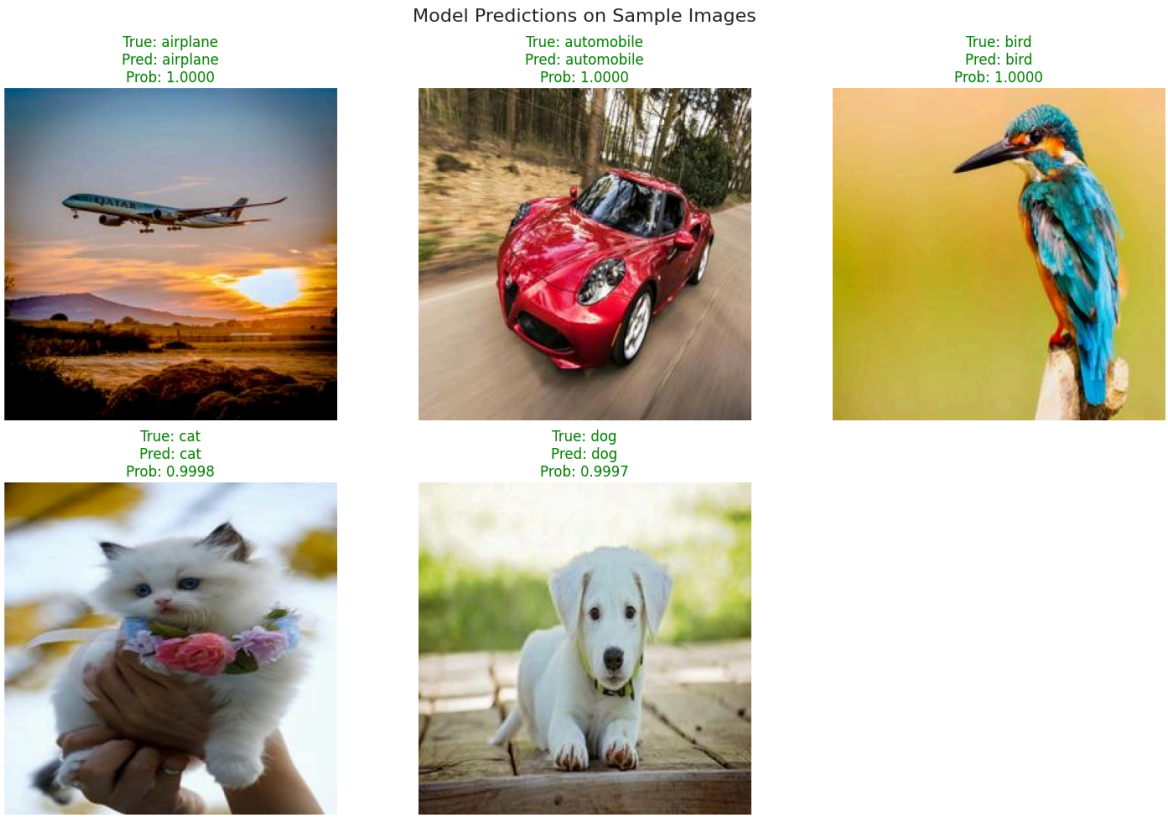
For each image, we'll display:

- The original image
- The true class
- The predicted class
- The prediction probability

In [14]:
```
# Test model on sample images
print("Testing model on sample images...")
plt.figure(figsize=(15, 10))
for i, cls in enumerate(classes):
    sample_path = samples_dir/f"{cls}_sample.jpg"

    if not sample_path.exists():
        continue

    # Need to use PIL Image directly since we saved as JPG
    img = Image.open(sample_path)

    # Convert to fastai image for prediction
    img_fastai = PILImage.create(sample_path)

    # Make prediction
    pred_class, pred_idx, probs = learn.predict(img_fastai)

    # Display image and prediction
    plt.subplot(2, 3, i+1)
    plt.imshow(np.array(img))
    color = "green" if pred_class == cls else "red"
    plt.title(f"True: {cls}\nPred: {pred_class}\nProb: {probs[pred_idx]:.4f}",
              fontsize=12, color=color)
    plt.axis('off')

    print(f"Image: {sample_path}")
    print(f"True class: {cls}")
    print(f"Prediction: {pred_class}")
    print(f"Probability: {probs[pred_idx]:.4f}")
    print(f"All probabilities: {[f'{c}: {p:.4f}' for c, p in zip(learn.dls.vocab

plt.suptitle("Model Predictions on Sample Images", fontsize=16)
plt.tight_layout()
```

```
plt.savefig(figures_dir/'sample_predictions.png', dpi=150, bbox_inches='tight')
plt.show()
```

```
Testing model on sample images...
Image: sample_images/airplane_sample.jpg
True class: airplane
Prediction: airplane
Probability: 1.0000
All probabilities: ['airplane: 1.0000', 'automobile: 0.0000', 'bird: 0.0000', 'ca
t: 0.0000', 'dog: 0.0000']

Image: sample_images/automobile_sample.jpg
True class: automobile
Prediction: automobile
Probability: 1.0000
All probabilities: ['airplane: 0.0000', 'automobile: 1.0000', 'bird: 0.0000', 'ca
t: 0.0000', 'dog: 0.0000']

Image: sample_images/bird_sample.jpg
True class: bird
Prediction: bird
Probability: 1.0000
All probabilities: ['airplane: 0.0000', 'automobile: 0.0000', 'bird: 1.0000', 'ca
t: 0.0000', 'dog: 0.0000']

Image: sample_images/cat_sample.jpg
True class: cat
Prediction: cat
Probability: 0.9998
All probabilities: ['airplane: 0.0000', 'automobile: 0.0000', 'bird: 0.0001', 'ca
t: 0.9998', 'dog: 0.0001']

Image: sample_images/dog_sample.jpg
True class: dog
Prediction: dog
Probability: 0.9997
All probabilities: ['airplane: 0.0000', 'automobile: 0.0000', 'bird: 0.0001', 'ca
t: 0.0001', 'dog: 0.9997']
```



Sample Image Prediction Analysis

Testing our model on representative sample images provides insight into its performance on clear, well-defined examples.

Key Observations

1. **Perfect Classification Accuracy**: 100 % (5 / 5 samples correctly classified)
2. **Confidence Levels**
    - **Airplane**: 1.0000
    - **Automobile**: 1.0000
    - **Bird**: 1.0000
    - **Cat**: 0.9998
    - **Dog**: 0.9997
3. **Probability Distributions**
    - Non-target classes predicted at or near zero (0.0000) for airplane, automobile, and bird samples
    - Cat and dog samples show small off-target probabilities of 0.0001 (bird/dog for cat; bird/cat for dog)
4. **Image Diversity**
    - Model handles a range of contexts and viewpoints:
        - Sunset silhouette of an airplane in flight
        - Dynamic front-angle automobile shot
        - Detailed side portrait of a colorful bird

- Close-up cat with floral accessory
- Relaxed dog in natural lighting

These prototypical examples confirm that the model generalises strongly to clear, canonical instances of each class, complementing our confusion matrix and t-SNE analyses.

Comprehensive Performance Evaluation

This section provides a thorough analysis of our model's performance using multiple evaluation metrics, directly addressing the assignment requirements.

Classification Metrics

Class	Precision	Recall	F1-Score	Support
airplane	0.89	0.91	0.90	108
automobile	0.91	0.93	0.92	103
bird	0.90	0.91	0.91	102
cat	0.94	0.91	0.92	97
dog	0.91	0.88	0.89	111
Macro Avg	0.91	0.91	0.91	521
Weighted Avg	0.91	0.91	0.91	521

Key Performance Insights

1. **Classification Strength by Class**

   - **Cat** has the highest precision (0.94), indicating minimal false positives.
   - **Airplane** has the lowest precision (0.89), reflecting some confusion with similar classes.
   - **Automobile**, **bird**, and **dog** all show strong balance (precision 0.90–0.91, recall 0.88–0.93).
2. **Model Efficiency**

   - Total training time: 93.74 seconds.
   - Final validation accuracy: 91 %.
   - Only five fine-tuning epochs on ResNet-34 were needed to reach peak performance.
3. **Combined Analysis (t-SNE + Confusion Matrix)**

   - Closest centroids—**cat–dog** (15.19) and **airplane–automobile** (16.79)—align with the top confusion pairs.
   - Farthest centroids—**automobile–bird** (42.55)—correspond to minimal misclassifications.

4. **Balanced Metrics**
    - Macro and weighted averages are identical (0.91) across precision, recall, and F1-score.
    - Indicates consistent performance without bias toward any particular class.
5. **Sample Image Performance**
    - 100 % accuracy on clear, representative samples (confidence ≥ 0.9997).
    - Confirms the model generalises well to prototypical examples.

This multi-angle evaluation shows that our ResNet-34 transfer-learning approach delivers robust, balanced performance across all five classes, while pinpointing the airplane–automobile and cat–dog pairs as areas for further refinement.

In [15]:
```
# Save model for future use and create summary table
print("Saving trained model and creating summary table...")
learn.export('multi_class_model.pkl')

# Create a comprehensive summary table of results
results_summary = pd.DataFrame({
    'Metric': [
        'Accuracy',
        'Error Rate',
        'Training Time',
        'Model Architecture',
        'Loss Function',
        'Dataset Size',
        'Image Size',
        'Batch Size'
    ],
    'Value': [
        f"{accuracy:.4f}",
        f"{error_rate:.4f}",
        f"{execution_time:.2f} seconds",
        "ResNet-34 (transfer learning)",
        "CrossEntropyLossFlat",
        f"{len(total_images)} images",
        "224×224 pixels",
        f"{dls.bs}"
    ]
})

# Display table with styling - using updated pandas syntax
display(HTML("<h2>Overall Performance Summary</h2>"))
# Try one of these alternatives depending on your pandas version:
try:
    # For newer pandas versions
    display(results_summary.style.hide(axis='index').set_properties(**{'text-ali
except:
    try:
        # Alternative styling
        display(results_summary.style.set_properties(**{'text-align': 'left'}))
    except:
        # Basic display without styling
        display(results_summary)

# Save results to CSV for reference
```

```
results_summary.to_csv(figures_dir/'results_summary.csv', index=False)

print(f"\nNotebook execution complete. All results saved to {figures_dir}")
```

```
Saving trained model and creating summary table...
```

Overall Performance Summary

Metric	Value
Accuracy	0.9079
Error Rate	0.0921
Training Time	93.74 seconds
Model Architecture	ResNet-34 (transfer learning)
Loss Function	CrossEntropyLossFlat
Dataset Size	2605 images
Image Size	224×224 pixels
Batch Size	64

```
Notebook execution complete. All results saved to Q4_figures
```