**Notes on the Project**

**Objective**

The plan is to build a website that takes an image and seeks to determine if it is a dog, cat, or human by providing a percentage of what it likely is. The goal is to provide four options: dog, cat, human, or something else beyond the model’s capability.

**Steps**

1. Gather and assess the data.

2. Process the data to pick the best images for the model.

3. Split the data into training, validate, and test sets.

4. Normalize the data.

5. Train the model.

6. Test the model.

7. Build the website.

**Gather the Data**

**Data Sources**

1. <https://www.robots.ox.ac.uk/~vgg/data/pets/>

This website has over 7,000 images of cats and dogs of different breeds. These images have been well-sorted and prepared.

2. <https://www.kaggle.com/datasets/jessicali9530/lfw-dataset>

This website has over 13,000 human images already sorted with a focus on faces.

**Data Preprocessing**

This covered the work done in the following files:

1. data\_preprocessing\_1

2. data\_preprocessing\_2  
3. data\_preprocessing\_3

**Organizing the Data**

This is the distribution of the cat and dog breeds, along with the number of images in the initial data.

### ****Cat Breeds****:

1. **Abyssinian – 200 images**
2. **bengal – 200 images**
3. **birman – 200 images**
4. **bombay – 200 images**
5. **british** (British Shorthair) **– 200 images**
6. **egyptian** (Egyptian Mau) **– 200 images**
7. **maine** (Maine Coon) **– 200 images**
8. **persian – 200 images**
9. **ragdoll – 200 images**
10. **russian** (Russian Blue) **– 200 images**
11. **scottish** (Scottish Fold) **– 199 images**
12. **siamese – 200 images**
13. **sphynx – 200 images**

****Total Cat Images - 2599****

### ****Dog Breeds****:

1. **american** (American Bulldog or American Eskimo) **– 400 images**
2. **basset** (Basset Hound) **– 200 images**
3. **beagle – 200 images**
4. **boxer – 200 images**
5. **chihuahua – 200 images**
6. **english** (English Bulldog or English Springer Spaniel) **– 200 images**
7. **german** (likely German Shepherd) **– 200 images**
8. **great** (likely Great Dane) **– 200 images**
9. **havanese– 200 images**
10. **japanese** (Japanese Chin or Spitz) **– 200 images**
11. **keeshond – 200 images**
12. **leonberger – 200 images**
13. **miniature** (Miniature Schnauzer) **– 200 images**
14. **newfoundland – 200 images**
15. **pomeranian – 200 images**
16. **pug – 200 images**
17. **saint** (Saint Bernard) **– 200 images**
18. **samoyed – 200 images**
19. **shiba** (Shiba Inu) **– 200 images**
20. **staffordshire** (Staffordshire Bull Terrier) **– 191 images**
21. **wheaten** (Soft Coated Wheaten Terrier) **– 200 images**
22. **yorkshire** (Yorkshire Terrier) **– 200 images**

****Total Dog Images – 4791****

****Picking the Best Images to Use****

**Since the goal is to pick a cat,dog, human, I knew the best approach was to place all cat images in the same place and pick images randomly to train the model. To do this, I placed all of the cat, dog, and human images in their own folders. They would be placed in random order without acknowledgment of the different breeds. The breeds would be placed together without order.**

**Once I did that, I proceeded to apply the YOLOv5 on the cat and dog images. YOLOv5 is a deep learning-based object detection model that stands for "You Only Look Once" version 5. It is designed to detect and localize multiple objects within an image in a single forward pass, making it both fast and efficient for real-time applications. YOLOv5 is implemented in PyTorch and comes in different model sizes (e.g., YOLOv5s, YOLOv5m, YOLOv5l, YOLOv5x) to balance speed and accuracy depending on the use case. It outputs bounding boxes, class labels, and confidence scores for each detected object, and is widely used for tasks like surveillance, autonomous driving, and image dataset preprocessing.**

**In my project, I use YOLOv5 object detection model to identify and extract images of cats and dogs. The human images were already processed and well-curated as the source is very reputable. The model detects objects in an image and provides bounding boxes with confidence scores. I filtered out any detections with a confidence score below 50% to ensure quality. For each valid detection, I cropped the image to the bounding box, resized it to 224×224 pixels, and saved the cleaned image into organized output folders. This process helped create a high-quality, standardized dataset for further use.**

**After this step, there were 1139 cat and 3282 dog images left. However, I was certain that these images were of great quality and that could be of great impact to the model I end up creating.**

****Splitting The Data into Training, Validation, and Test Sets****

**The next step was to split the data into these three groups. I chose a ratio of of 80%, 20%, and 20% for training, validation, and test sets respectively. At this point, the breakdown of the data in the train, validation, and test sets is as follows:**

|  |  |  |  |
| --- | --- | --- | --- |
| Data | Folder Name | | |
|  | Cat | Dog | Human |
| Train | 911 | 2625 | 10586 |
| Validate | 113 | 328 | 1323 |
| Test | 115 | 329 | 1324 |

**This split shows that there is a great discrepancy in the training data. There are very few cat images and very many human images. The next step in this case is to try and address that. First, I wrote code that picked 2500 random human images. This would allow me to create training data that does not bias the data.**

****Augmenting the Training Data****

**In this case, I wanted to augment the cat images to get them to at least 3000 images for each to allow for better model training. This is where augmentation came in. Augmentation in image preparation refers to the process of artificially increasing the size and diversity of a dataset by applying various transformations to the original images. These transformations can include operations such as rotation, flipping, scaling, cropping, brightness adjustment, noise addition, and more. The goal of augmentation is to help a model generalize better by exposing it to a wider variety of image conditions, reducing the risk of overfitting and improving performance on unseen data. It is especially useful when working with limited datasets. With that, I have 3000 images of all three creatures for training.**

****Transforming The Data****

**After this, I placed the training, test, and validation folders under the same folder. I wanted to solely augment the training data while leaving the rest out of it. The next step is transformation. Transformation. in this context, refers to the preprocessing steps applied to each image before it is fed into the model. Specifically, the images are first converted from PIL format (or NumPy arrays) into PyTorch tensors using transforms.ToTensor(), which also scales pixel values to the [0, 1] range. Then, transforms.Normalize([0.5], [0.5]) standardizes the data by shifting the pixel values to the range [-1, 1] using a mean and standard deviation of 0.5. This normalization helps the model train more effectively by ensuring the input values are centered and scaled, which can lead to faster convergence and more stable gradients during backpropagation. Proper transformations also ensure that the input data format matches the model's expected input, which is crucial for correct and efficient training.**

**After transformation, I set up data loaders. Data loaders in PyTorch are utilities that handle the loading of datasets in manageable batches, optionally shuffling the data and loading it in parallel using multiple worker threads. In this case, DataLoader wraps around each dataset (train, validation, test) and creates an iterator that yields batches of 32 images at a time. This batching process is essential for training neural networks efficiently, especially when using GPUs, as it allows parallel computation across multiple samples. Additionally, shuffling the training data each epoch improves generalization by preventing the model from learning the order of the data. Using data loaders also helps manage memory efficiently, as images are only loaded as needed, rather than all at once into memory.**

****Summary Table****

|  |  |  |  |
| --- | --- | --- | --- |
| **Step** | **Train** | **Validation** | **Test** |
| Resize (to 224\*224) | ✅ | ✅ | ✅ |
| Augmentation | ✅ | X | X |
| ToTensor (Transformation) | ✅ | ✅ | ✅ |
| Normalize | ✅ | ✅ | ✅ |

**The Modeling Phase**

**Training The Model**

To build the model, I used ****PyTorch****, a popular deep learning framework. The goal of the model is to classify images into one of **four categories**. To do this, I used a powerful, pre-trained model called **ResNet-50**, which is part of the torchvision.models library. ResNet-50 is a deep convolutional neural network that was originally trained on a large dataset called ImageNet. Rather than training a model from scratch, I used **transfer learning** — meaning I took a model that already understands basic image features and fine-tuned it for my specific dataset.

The data for this project was organized into three folders: train, val (validation), and test, each containing subfolders for the four image classes. To help the model generalize better and not overfit the training data, I applied several **data augmentation** techniques like random flips, rotations, and brightness changes to the training images. I also resized all images to 224x224 pixels and normalized them using values that match what the ResNet model expects. The torchvision.transforms module was used for all of these preprocessing steps.

To fine-tune the ResNet-50 model, I froze most of its layers so they wouldn't be updated during training. I only allowed the last block of layers (layer4) and the final fully connected layer (fc) to be trained. Then, I replaced the original final layer, which was designed to output 1,000 classes, with a new layer that outputs just 4 classes — one for each category in my dataset. This approach helped me make use of the model's existing feature-detection abilities while still tailoring it to my specific classification task.

I trained the model over **10 epochs** using a batch size of 32. During each epoch, the model was trained on the training data and evaluated on the validation set. I used **cross-entropy loss**, which is standard for classification tasks, and optimized the model using the **Adam optimizer**. To improve training, I also included a **learning rate scheduler** that automatically reduces the learning rate if the validation loss stops improving.

After each epoch, I calculated and printed the training and validation accuracy and loss. I also kept track of the best-performing model by saving its weights whenever the validation loss decreased. The final result of this training process was a model that was well-tuned to recognize the four image classes in my dataset. Once training finished, I saved the best version of the model to a file called best\_model.pth.

**Testing The Model**

After training the image classification model using ResNet-50, I evaluated its performance on a separate test dataset. The goal of this step is to see how well the model generalizes to new, unseen data — which is important for understanding its real-world usefulness. To do this, I used the same image transformations as during training, including resizing, converting to tensors, and normalizing with the same mean and standard deviation values expected by the ResNet architecture.

The test images were stored in a directory called test, structured similarly to the training and validation sets. I loaded these images using PyTorch's ImageFolder and wrapped them in a DataLoader with no shuffling, so I could keep track of predictions in the original order. Each batch of images was moved to the correct device (either CPU or GPU), and I made predictions using the model.

Before making predictions, I had to rebuild the model architecture to match the one used during training. I recreated a ResNet-50 model and adjusted its final layer to output 4 classes. Then I loaded the saved model weights from the best-performing version trained earlier, stored in a file named best\_model.pth.

With the model in evaluation mode (model.eval()), I passed all the test images through it and collected both the predicted class labels and the true labels. I made sure not to compute gradients during this step (torch.no\_grad()), since we're not training here — just evaluating.

Finally, I used Scikit-learn to calculate evaluation metrics. Specifically, I printed a classification report, which shows precision, recall, and F1-score for each of the four classes. I also printed a confusion matrix, which helps visualize which classes the model is predicting correctly or confusing with others. These results give a clear picture of how well the model is performing across all categories, and can highlight any weaknesses or imbalances.

The results of the data were slightly suspicious as the results showed 100% accuracy. This could mean one of the following:

1. The model is severely overfitting.
2. The test set is not actually "unseen".
3. There’s a data leakage issue.
4. The dataset is too easy or too small.
5. There's an error in labeling or evaluation logic.

**Implementing The Model**

## **Summary of Client Upload Pipeline**

1. **Client uploads image**
2. **Run YOLOv5 to detect object**
3. **Crop image to bounding box**
4. **Resize to 224×224**
5. **Convert to tensor**
6. **Normalize**
7. **Feed into trained model**
8. **Get class probabilities (e.g., Cat: 85%, Dog: 10%, Human: 4%, Other: 1%)**
9. **Return label and confidence**

To do this, I used the Flask framework to help me in this endeavor. This code creates a simple web application that allows users to upload an image and get a prediction of what’s in that image. It uses a Python framework called Flask to build the website and manage how users interact with it. When someone visits the main page of the site, they will see a webpage where they can upload an image file. After they upload the image, the app processes it and tells them whether the image most likely shows a cat, a dog, a human, or something else.

Before the app starts working, it loads a machine learning model that has already been trained to recognize these four categories. This model is stored in a file, and the app loads it only once when it first starts up. That way, it doesn’t have to reload the model every time someone uploads a picture, which helps it work faster.

When a user uploads an image, the app saves it temporarily so it can pass it to the model. The model then looks at the image and makes a prediction. It returns both the most likely label — such as “dog” — and the confidence levels for each of the four possible options. The app then formats these results into a friendly message that shows the prediction and how confident the model is about each choice. After that, it deletes the temporary image to keep the server clean and organized.

In the end, the user receives a simple message that tells them what the model thinks the image is and how sure it is about its guess. The whole process happens quickly and smoothly, thanks to how the app is designed.

**Running the Project**

If you wish to run the project, open your terminal up until the app.py file which is in the ml\_web\_app. The HTML and CSS files are in the right positions according to flask and the functions that allow the operations is in the inference.py. If the app.py file is run and the corresponding IP address is run locally, the application will run.

**Conclusion**

This project shows how artificial intelligence can be used in a real-world application. It combines a trained machine learning model with a web interface so that people can easily interact with it. The result is a working system where anyone can upload a photo and get instant feedback, which is both useful and easy to understand.