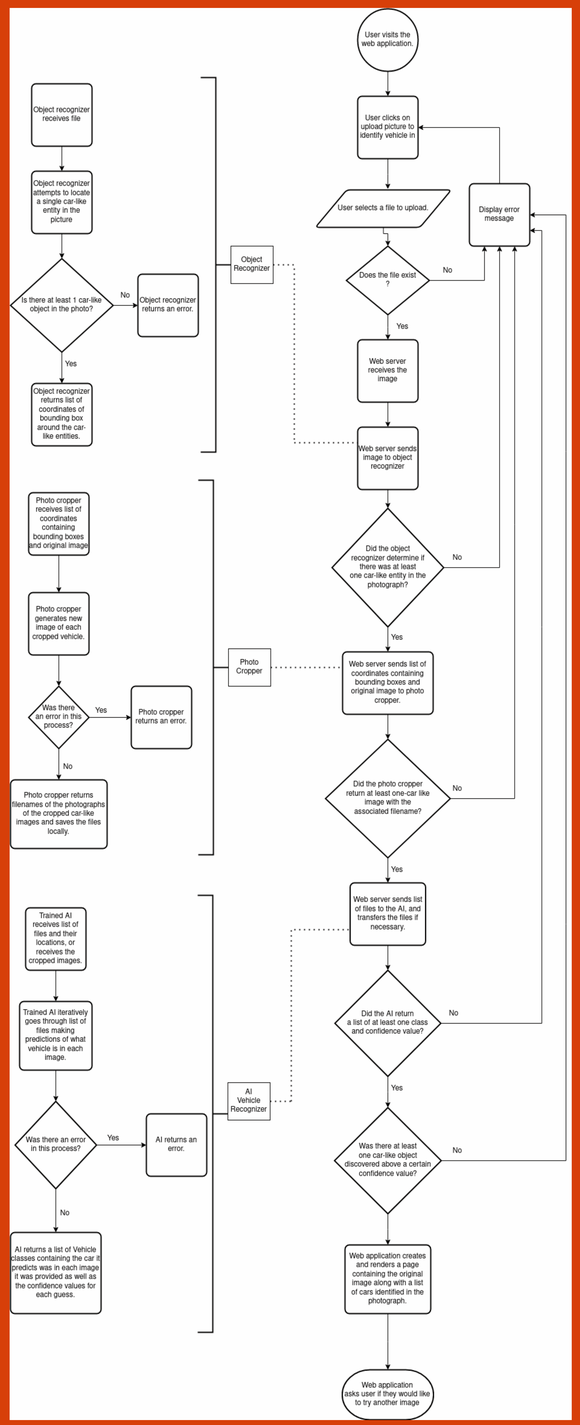
College of Engineering

Electrical Engineering and Computer Science

Application Flow Diagram

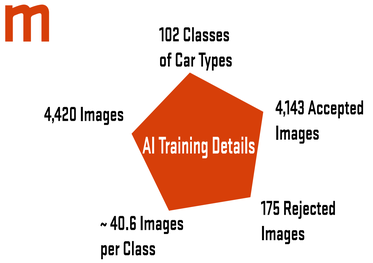


Carzam - Car Identification System

Kevin Joy, George Kochera, and Gabrielle Pang

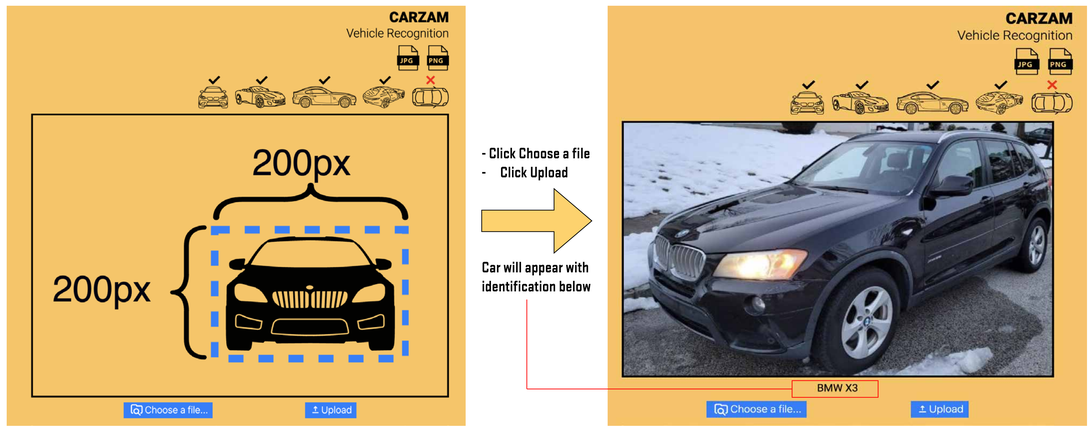
https://carzamm.herokuapp.com/

https://github.com/carzamm/carzam



Carzam - A website application deployed on heroku and served via a docker container. This website allows users to upload a photo in either png or jpg format of a vehicle for identification. In the current iteration, we have the ability to identify over 100 different types of cars!

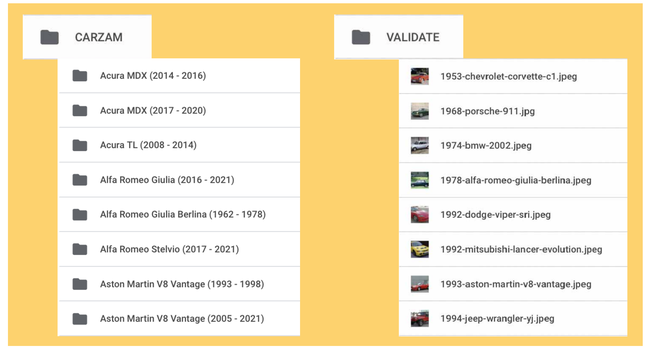
Website Functionality



Quirks and Features

- The first load requires time for Heroku App to spin up and load all the dependencies
- We recommend inputting images larger than 400x400 pixels for best results
- There is some wait time between uploading and identification

Training Model Layout



Underlying Code

```
# This specifies how the data will be transformed from the image to what PyTorch will recognize
# these values are specific and shouldn't be changed. They were specified by the PyTorch documentation.

train_tfms = transforms.Compose([transforms.Resize((400, 400)),
                                 transforms.RandomHorizontalFlip(),
                                 transforms.RandomRotation(15),
                                 transforms.ToTensor(),
                                 transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))])
test_tfms = transforms.Compose([transforms.Resize((400, 400)),
                                transforms.ToTensor(),
                                transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))])

# This defines the dataset location and then loads the data
dataset = torchvision.datasets.ImageFolder(root=dataset_dir+"/train", transform = train_tfms)
trainloader = torch.utils.data.DataLoader(dataset, batch_size = BATCH_SIZE, shuffle=True, num_workers = 2)

dataset2 = torchvision.datasets.ImageFolder(root=dataset_dir+"/test", transform = test_tfms)
testloader = torch.utils.data.DataLoader(dataset2, batch_size = BATCH_SIZE, shuffle=False, num_workers = 2)


# This specifies the model 'Resnet34' which is pretrained, we only want to change the last layer
# which is the output layer.
model_ft = models.resnet34(pretrained=True)
num_ftrs = model_ft.fc.in_features

# Freeze all the layers of the model since it is pretrained, we only want to update the last layer
# for param in model_ft.parameters():
#     param.requires_grad = False

# replace the last fc layer with an untrained one (requires grad by default)
model_ft.fc = nn.Linear(num_ftrs, QTY_CLASSES)
model_ft = model_ft.to(device)

# This specifies how the model is optimized between epochs such as learning rate and momentum
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model_ft.parameters(), lr=LEARNING_RATE, momentum=MOMENTUM)

# This line specifically handles learning rate of the AI and causes it to adjust when there is a 'learning plateau'.
lrscheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='max', patience=3, threshold = 0.9)

# This is the call to train the actual model: It returns the model with some data dervied from training
model_ft, training_losses, training_accs, test_accs = train_model(model_ft, criterion, optimizer, lrscheduler, n_epochs=TRAINING_EPOCHS)

# This plots the results
if args.p:
    plot_ai_results(training_losses, training_accs, test_accs)

# Save the CSV file if requested
if args.w:
    write_csv(training_losses, training_accs, test_accs)

# Save the model to the current directory
MODEL_FILENAME = "saved_model.pt"
torch.save(model_ft.state_dict(), MODEL_FILENAME)
```

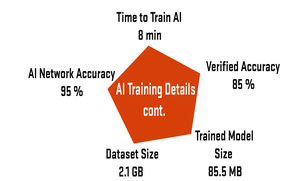
snippet from train.py - utilizes Pytorch to transform the images in the test and train directories, specifies that we're utilizing the resnet34 model, then we change the last output layer to be linear based on the number of classes (different types of cars e.g. BMW X3, Jeep Wrangler, etc.), specifies how the model should be optimized (utilizing the CrossEntropyLoss method), then we run the information through the train_model function which iteratively runs through all of the data we have collected (usually we set it to 20 Epochs) to learn the information and weights and provide the loss, training accuracy, and test accuracy and once that is completed we save the weights to a saved_model.pt to be used in identification.

snippet from recognizer.py - utilizes Yolov5 to obtain the location of the car within the image provided. We set Yolo to detect cars and trucks and it will output the boundary box (coordinates) of where the car/truck is in the image. The result is then sent to a cropping function which takes the coordinates and alters the image. This improves our accuracy because we are removing extraneous image information prior to comparing the image to our saved weights and also prior to utilizing the image for training.

```
class Recognizer:
    def __init__(self):
        self.model = torch.hub.load('ultralytics/yolov5', 'yolov5s', pretrained=True, verbose=False)

    def recognize_objects(self, path_to_image: str, verbose=False):
        """
        recognize_objects takes one argument, the path to the image it will be recognizing
        objects in. It will run the image through the YOLOv5 AI and will return a list of
        tuples in the format of:

        [(filename, index, x-left, y-top, x-right, y-bottom), ...]

        This datatype is referred to as crop_instructions
        """

        # Set YOLO to only detect cars (id = 2) and trucks (id = 7)
        self.model.classes = [2, 7]

        # This will be list of 6-tuples (filename, index, x-left, y-top, x-right, y-bottom)
        crop_instructions = []

        # Inference
        # results will be a list of Detection objects (cars, trucks, etc.)
        results = self.model(path_to_image).tolist()

        # For each detection object, go through all the objects detected for that class of detection
        for detection in results:

            # Print the relevant filename
            if verbose:
                print("\nFile ----> {}".format(path_to_image))

            # Bring the 'xyxy' tensors off the GPU back to the CPU and convert to a list
            vehicles = detection.xyxy.cpu().numpy().tolist()

            # For vehicle detected in the detection object, get the relevant information
            for index, vehicle in enumerate(vehicles):

                # Set some variables to track the pixel coordinates
                x_left = int(vehicle[0])
                y_top = int(vehicle[1])
                x_right = int(vehicle[2])
                y_bottom = int(vehicle[3])

                # Output the finding to the console
                if verbose:
                    print("\n #{}".format(index + 1))
                    print("\tTop-Left (x,y): {},{}".format(x_left, y_top))
                    print("\tBottom-Right (x,y): {}, {}".format(x_right, y_bottom))

                # Add this to the crop instructions so the image cropper can generate the new pictures
                crop_instructions.append((str(path_to_image), index, x_left, y_top, x_right, y_bottom))

        return crop_instructions
```

```
    if list_of_paths is None:
        files = [self.verification_dir + x for x in os.listdir(self.verification_dir)]
    else:
        files = list_of_paths

    # Used to store results of vehicle recognition
    results = []

    # Identify each file
    for file in files:
        results.append(self.test_single_car(file))

    return results

def test_single_car(self, path: str):
    filename = os.fsdecode(path)

    # transforms for the input image
    loader = transforms.Compose([transforms.Resize((400, 400)),
                                 transforms.ToTensor(),
                                 transforms.Normalize((0.485, 0.456, 0.406), (0.229,
    image = Image.open(path).convert('RGB')
    image = loader(image).float()
    image = torch.autograd.Variable(image, requires_grad=True)
    image = image.unsqueeze(0)

    if self.web:
        image = image.cpu()
    else:
        image = image.cuda()

    output = self.model_ft(image)
    conf, predicted = torch.max(output.data, 1)


    # This takes a plain text file and treats each separate line as a car class
    # The text file must have each class sorted alphabetically
    with open('./common/carzam102.dat', 'r') as class_file:
        lines = class_file.readlines()
        print(lines)
        classes = lines
```

snippet from identify.py - we load the saved weights file that was created in train.py. Then we go through all the images in verify and compare that image to the inputted image, we find which class it matches by comparing it to the information in the carzam102.dat file and then return the results to be displayed on the webpage.



snippet from app.py - this is what we use for the home page. We delete any old images from the server and deal with the image that is uploaded from the user. We save that image and perform the background code to determine the car class and render the results to the page (as shown in the Website Functionality section).

```
# on new upload event, remove old images from server
shutil.rmtree(IN_DIRECTORY)
shutil.rmtree(OUT_DIRECTORY)
Path(IN_DIRECTORY).mkdir(parents=True, exist_ok=True)
Path(OUT_DIRECTORY).mkdir(parents=True, exist_ok=True)

# check if the post request has the file part
if 'file' not in request.files:
    flash('No file')
    return redirect(request.url)
file = request.files['file']
# if user does not select file, browser also
# submit an empty part without filename
if file.filename == '':
    flash('No selected file')
    return redirect(request.url)

# This looks like the case when the file was uploaded and it was valid
if file and allowed_file(file.filename):
    path_to_file = save_file_to_upload_directory(file)
    # pass file to carzam.py
    results = parse_file(path_to_file)
    cropped_file_list = [(path, car, conf) for path, car, conf in results]

    return render_template('index.html', filename = cropped_file_list)
```

Oregon State University