Sparse Visual Odometry:
Estimating Vehicle Speed from Timelapse Video

ASM591-AI Project

Timothy J Wiegman

2024-12-10

Abstract

Computer vision is a critical technology underlaying many modern vehicle systems. This paper explores a novel computer vision system for estimating the speed of a vehicle from onboard timelapse recordings of the vehicle's surroundings, using neural networks containing CNN and LSTM layers. The new system was trained on 24400 frames of video (corresponding to 6.7 hours of activity) and learned to estimate the vehicle's speed with a mean absolute percent error (MAPE) of 37.2%. While this level of performance means the system cannot be used to calculate precise quantitative estimates, it shows that the method is feasible in principle, and future developments could improve the accuracy.

1 Introduction & Related Work

Visual odometry, estimating the speed of objects from photographic images, is a classic computer vision challenge. A common formulation of this problem is in estimating the speed of vehicles on a road. However, while the task is well-characterized in several forms [1], most of these focus on detecting the speed of *other* vehicles on the road, either by stationary traffic cameras [2] or by onboard driver-assist-type systems [3]. Using an onboard camera to detect the vehicle's *own* ground speed is a less common goal, since most relevant systems would have access to the output of an odometer or speedometer that can directly measure the ground speed. In this study, the aim is to estimate the speed of the vehicle solely from its own onboard camera; it assumes that telematics (including recorded speedometer data) are unavailable. Additionally, this system works on low-frequency video, timelapses recorded with a full second of delay between subsequent frames.

This task and associated constraints were chosen in order to support an existing research project. The Evans Lab at Purdue University is studying roadside mowing practices in the state of Indiana. As part of this research, many hours of both telematics data and timelapse video were recorded from tractors towing mowers along Indiana roadways. However, there exist gaps in the data sets for both modalities; being able to accurately extract a vehicle speed from video records would allow portions without accompanying telematics information to partially replace the missing telematics data.

This problem is significant because it advances the science of fusing multimodal data sources in a physical environment, and, assuming the artificial intelligence's speed estimates prove to be reliable and accurate, because it allows the original research dataset to be retroactively expanded to cover more operations. While there exist some methods for estimating vehicle velocity from video input [4], they typically either rely on grounded external cameras [2] or estimate only the relative velocity of other vehicles [3], rather than the ground velocity of the vehicle carrying the camera itself. In addition, there is novelty in the fact that this project aims to use *timelapse* video, rather than conventional video; since timelapses capture images across longer periods, these kinds of videos are working with information that is more temporally sparse than in conventional videos by at least an order of magnitude. Timelapse imagery has been used in some existing studies [5], but not applied to active vehicles as in this study.

2 Methods

2.1 Data Sources

Data used in this study was collected from roadside mowing operations in the summer of 2024. The tractors used consisted of Maxxum 115 and 125 tractors (Case IH, Racine, WI). Towed mowers were all various models of flex-wing rotary cutters 15 feet in width (Brush Hog, Selma, AL). All mowing operations were performed by INDOT contractors with privately owned equipment, as shown in Figure 1.

Figure 1: An INDOT contractor's tractor and towed mower.

Timelapse videos were recorded using a Hero 8 Black action-sports camera (GoPro, San Mateo, CA). Cameras were mounted in rugged metal motorsports frames, as shown in Figure 2. These systems were flashed with an experimental firmware [6] that allows recordings to be triggered programmatically in response to different conditions. GPS metadata was extracted from video timelapses using an open-source python program [7]. Videos were recorded at a sample rate of 1 Hz and were triggered to record automatically whenever the camera detected the tractor to be powered on and moving.

Figure 2: The type of camera and mounting frame used in this study.

Telematics were logged using a Purdue ISOBlue system [8], shown in Figure 3. These systems record all messages on the tractor's internal controller area network (CAN), time and location as received from GPS, and have the capability to be remotely accessed over a cellular data connection. These devices were chosen because they are open-source [9] and technical support was readily available from local colleagues. Logged data was stored in three SQL tables; one for GPS data, one for CAN data, and one for cellular data. The GPS table contains columns for timestamps, latitude, and longitude. The CAN table contains columns for

timestamps, network interface, message ID, and message data. Vehicle ground speed was taken from CAN messages and cross-referenced with GPS data for validation.

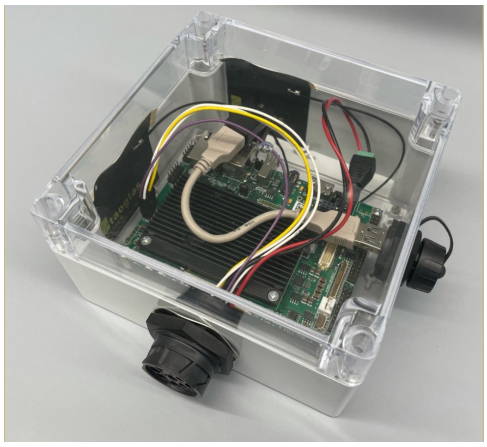

Figure 3: A Purdue ISOBlue system.

2.2 Machine Learning

The general architecture of the machine learning network trained in this study is shown in Figure 4. A notable piece of preprocessing applied to the data was the extraction of depth information with the pre-trained Depth Anything v2 model [10], shown in grey in the figure. This mimicked the use of a stereo depth camera, commonly used in robotics and autonomous vehicles, without requiring such a camera system to be deployed in the field. The depth information was added as an additional color channel to the images before being processed by the visual odometry model; the Depth Anything model was used solely in inference mode and not re-trained during the learning process.

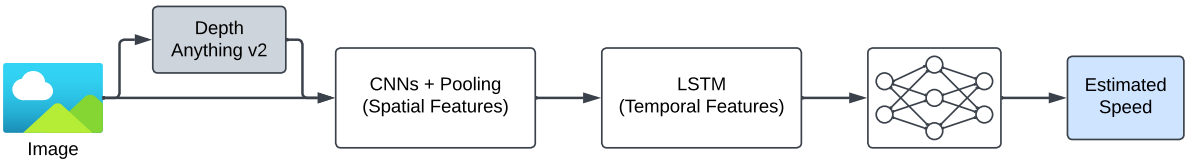

Figure 4: An overview of the architecture used for the machine learning model.

After being combined with depth information, images were resized to 256 pixels square, to enforce uniform input and to decrease computational load. Three convolutional neural networks (CNNs) were used in the initial stage of the model. The first was followed by an average pooling stage with a 2x2 kernel, combining spatial features from across the image and reducing the resolution to 128 pixels square. The second was followed by a maximum pooling stage with a 4x4 kernel, extracting all the most prominent features and reducing resolution to 32 pixels square. The third and final CNN layer was followed by an average pooling stage with a 2x2 kernel, condensing the information to 16 pixels square.

After the CNN layers, a Long Short-Term Memory recurrent neural network (LSTM) was used to capture temporal information from successive frames. The LSTM used in this model had eight nodes with four hidden layers, and was only used in unidirectional mode as vehicle speed requires no foreknowledge to measure.

Two layers of fully-connected nets were used to after the LSTM output to condense the information down to a single number for each input frame. Velocities were measured in the meters per second. Ground-truth data for supervised training was taken from telematics measurements, and recorded in rolling averages across 60 second intervals to reduce noise and provide more stable targets for the model to converge on.

The full program developed for this experiment is given in the appendix, section A.

3 Results

After training on 24400 frames of timelapse video (corresponding to about 6.8 hours of mowing operations), the model was evaluated against 4180 frames of new data (about 1.2 hours of mowing operations). For the evaluation dataset, the mean average percent error (MAPE) of the model's predicted speeds was 37.25%. Mean average percent error is given by the following equation:

$$\text{MAPE} = \left| \frac{y_{pred} - y_{true}}{y_{true}} \right|$$

Therefore, a MAPE of 37.25% shows that the technology has promise, but this model cannot provide high-confidence predictions of precise vehicle speeds. For example, a predicted speed of 10 m/s (~22 mph) could match a true speed of as low as 6.3 m/s (~14 mph) or as high as 13.7 m/s (~31 mph).

4 Discussion

The results show that the trained model has value, but is of limited precision. The trained model could be used to estimate the general activity of the monitored tractor; the model is sufficiently accurate to classify which periods contain low-speed or high-speed motion. If quantitative odometry is required, the MAPE can be used to provide a confidence interval for the estimated speed.

Several possible flaws in the system could be remedied in future development. One likely source of error in the model is the motion of other vehicles in the frame. A pre-trained object identification model could be a simple and effective solution to this issue, to give the model additional information about what objects are in the frame and whether they are likely to be stationary between frames. Such an extension could be added in similar fashion to the depth information that was explained in Section 2.2.

Another future improvement could be found in streamlining the training process. The data loading system calculated the depth frames on-demand, which seemed to introduce a significant bottleneck to the program. Rewriting the data loader to pre-process the videos in parallel may significantly speed up performance, allowing the model to be trained more quickly and/or with more data.

5 References

[1] D. Fernández Llorca, A. Hernández Martínez, and I. García Daza, "Vision-based vehicle speed estimation: A survey," *IET Intelligent Transport Systems*, vol. 15, no. 8, pp. 987–1005, 2021, doi: 10.1049/itr2.12079.

[2] D. Bell, W. Xiao, and P. James, "Accurate vehicle speed estimation from monocular camera footage," *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. 2, pp. 419–426, 2020.

[3] I. García-Aguilar, J. García-González, D. Medina, R. M. Luque-Baena, E. Domínguez, and E. López-Rubio, "Detection of dangerously approaching vehicles over onboard cameras by speed estimation from apparent size," *Neurocomputing*, vol. 567, p. 127057, Jan. 2024, doi: 10.1016/j.neucom.2023.127057.

[4] A. Marban, V. Srinivasan, W. Samek, J. Fernández, and A. Casals, "Estimating position and velocity in 3d space from monocular video sequences using a deep neural network," in *Proceedings of the IEEE international conference on computer vision workshops*, 2017, pp. 1460–1469.

[5] H. Hendrickx, X. Blanch, M. Elias, R. Delaloye, and A. Eltner, "AI-Based Tracking of Fast-Moving Alpine Landforms Using High Frequency Monoscopic Time-Lapse Imagery," *EGUsphere*, pp. 1–20, Aug. 2024, doi: 10.5194/egusphere-2024-2570.

[6] D. Newman, *GoPro Labs*. (2022). Accessed: Sep. 03, 2024. [Online]. Available: https://github.com/gopro/labs

[7] J. M. Casillas, *gopro2gpx*. (2023). Accessed: Sep. 03, 2024. [Online]. Available: https://github.com/juanmcasillas/gopro2gpx

[8] A. D. Balmos, F. A. Castiblanco, A. J. Neustedter, J. V. Krogmeier, and D. R. Buckmaster, "ISOBlue Avena: A Framework for Agricultural Edge Computing and Data Sovereignty," *IEEE Micro*, vol. 42, no. 1, pp. 78–86, Jan. 2022, doi: 10.1109/MM.2021.3134830.

[9] Oats-Center, "ISOBlue hardware, avena software, and deployment files." Online, 2023. Available: https://github.com/oats-center/isoblue

[10] L. Yang *et al.*, "Depth anything V2," 2024, Available: https://arxiv.org/abs/2406.09414

A Program Code

```
#!/usr/bin/python3
# Created 2024-12-07
# TJ Wiegman
# for ASM 591 AI final project

# Files to work with
training_videos = [
    '/mnt/nas/2024/Tractor02/2024-06-18_083656.MP4',
    '/mnt/nas/2024/Tractor02/2024-06-18_120818.MP4',
    # '/mnt/nas/2024/Tractor02/2024-06-19_083249.MP4',
    # '/mnt/nas/2024/Tractor02/2024-06-19_095128.MP4'
]
testing_videos = [
    # '/mnt/nas/2024/Tractor02/2024-06-18_083656.MP4',
    # '/mnt/nas/2024/Tractor02/2024-06-18_120818.MP4',
    '/mnt/nas/2024/Tractor02/2024-06-19_083249.MP4',
    # '/mnt/nas/2024/Tractor02/2024-06-19_095128.MP4'
]
model_checkpoint = "asm591ai.tar"

# Training/Testing controls
import os
MAX_EPOCHS = 1
checkpoint_already_exists = os.path.exists(model_checkpoint)

# Enable Pytorch GPU acceleration
import torch
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(device)

# NN Model
import torch.nn as nn
import torch.nn.functional as F
BATCH = 20

class VisOdoNet(nn.Module):
    def __init__(self):
        super().__init__()
        # Convolutional layers to learn spatial features
        self.conv1 = nn.Conv2d(4, 5, kernel_size=5, padding=2)
        self.conv2 = nn.Conv2d(5, 3, kernel_size=3, padding=3, dilation=2)
        self.conv3 = nn.Conv2d(3, 1, kernel_size=5, padding=2)

        # LSTM network to learn temporal features
        self.rnn1 = nn.LSTM(input_size=256,
                            hidden_size=8, num_layers=4,
                            bidirectional=False, # should be monotonic
                            batch_first=True)

        # And a few FC layers to tie it all together
        self.fc1 = nn.Linear(64, 16)
```

```
        self.fc2 = nn.Linear(16, BATCH)

    def forward(self, x):          # input [B x 4 x 256 x 256]
        x = F.relu(self.conv1(x)) # shape [B x 5 x 256 x 256]
        x = F.avg_pool2d(x, 2)    # shape [B x 5 x 128 x 128]
        x = F.relu(self.conv2(x)) # shape [B x 3 x 128 x 128]
        x = F.max_pool2d(x, 4)    # shape [B x 3 x  32 x  32]
        x = F.relu(self.conv3(x)) # shape [B x 1 x  32 x  32]
        x = F.avg_pool2d(x, 2)    # shape [B x 1 x  16 x  16]
        x = x.reshape(-1, 256)    # shape [B x 256]

        _, (x, c) = self.rnn1(x)  # pull both hidden and cell states
        x = x.reshape(-1, 32) # B x 32 hidden state
        c = c.reshape(-1, 32) # B x 32 cell state
        x = torch.concat((x,c))
        x = F.relu(x.reshape(-1, 64))   # [B x 64]
        x = F.relu(self.fc1(x))   # shape [B x 16]
        x = F.relu(self.fc2(x))   # shape [BATCH]
        return x.reshape(-1)

# Utility Functions
import datetime
from dateutil import parser

def stamp(dt):
    '''Converts a datetime into a string suitable for indexing'''
    return f"{dt.date()}_{dt.hour:02}:{dt.minute:02}"

def unstamp(st):
    '''Converts a `stamp` string back into a datetime'''
    date, time = st.split("_")
    yy,mm,dd = date.split("-")
    hh,mn = time.split(":")
    return datetime.datetime(
        year=int(yy), month=int(mm), day=int(dd),
        hour=int(hh), minute=int(mn)
    )

def roll_avg(frame, new):
    N = frame[1] + 1
    output = frame[0]*(1 - 1/N) + new/N
    return output, N

# Import depth feature extraction model
import sys
sys.path.append("./")
from Depth_Anything_V2.depth_anything_v2.dpt import DepthAnythingV2
# From https://github.com/DepthAnything/Depth-Anything-V2#use-our-models
model_configs = {
    'vits': {'encoder': 'vits', 'features': 64, 'out_channels': [48, 96, 192, 384]},
    'vitb': {'encoder': 'vitb', 'features': 128, 'out_channels': [96, 192, 384, 768]},
    'vitl': {'encoder': 'vitl', 'features': 256, 'out_channels': [256, 512, 1024, 1024]},
    'vitg': {'encoder': 'vitg', 'features': 384, 'out_channels': [1536, 1536, 1536, 1536]}
```

```
}
encoder = 'vitb' # or 'vits', 'vitl', 'vitg'
depth_model = DepthAnythingV2(**model_configs[encoder])
depth_model.load_state_dict(torch.load(f"Depth_Anything_V2/checkpoints/depth_anything_v2_{encoder}.pth"
depth_model = depth_model.to(device).eval()


# Data structure for multithreaded video file streaming
import cv2, time
from threading import Thread
from queue import Queue

class FileVideoStream:
    # queue up frames in the background for faster playback
    # adapted from https://pyimagesearch.com/2017/02/06/faster-video-file-fps-with-cv2-videocapture-and
    def __init__(self, path, queueSize=60):
        self.stream = cv2.VideoCapture(path)
        self.stopped = False
        self.Q = Queue(maxsize=queueSize)

    def more(self): return self.Q.qsize() > 0
    def read(self): return self.Q.get()
    def stop(self): self.stopped = True

    def update(self):
        while True:
            if self.stopped: return
            if not self.Q.full():
                (grabbed,frame) = self.stream.read()
                if not grabbed:
                    self.stop()
                    return
                self.Q.put(frame)

    def start(self):
        t = Thread(target=self.update, args=())
        t.daemon = True
        t.start()
        time.sleep(1.0) # give the queue a second to fill a bit
        return self

# Create frame-loading dataset
from torch.utils.data import Dataset, DataLoader
import numpy as np
import cv2, json

class VideoSet(Dataset):
    def __init__(self, video_paths):
        self.jsons = []
        self.sources = []
        self.n_frames = 0
        self.videos = {}
```

```
        for path in video_paths:
            # Get metadata from JSON
            name, ext = path.split(".")
            assert ext.upper() == "MP4"
            jfile = name + ".json"
            with open(jfile) as file:
                jdata = json.load(file)

            # Calculate frame numbers
            startFrame = self.n_frames
            self.n_frames += jdata[-1]["frame"] + 1 # because zero indexed
            endFrame = self.n_frames

            # Save data to self
            self.sources.append((startFrame, endFrame, path))
            self.jsons.append(jdata)

    def source_lookup(self, idx):
        i = 0
        for start, end, _ in self.sources:
            if idx >= start and idx < end:
                return i
            i += 1
        raise IndexError(f"Index {idx} not found in {self.sources}")

    def load_video(self, path):
        self.videos[path] = FileVideoStream(path).start()

    def __len__(self):
        return self.n_frames

    def __getitem__(self, idx):
        # Lookup correct source for idx
        i = self.source_lookup(idx)
        start, _, path = self.sources[i]
        fidx = idx - start
        if path not in self.videos:
            self.load_video(path) # lazy loading

        # Get visual data
        frame = self.videos[path].read() # shape H x W x 3
        depth = np.expand_dims(depth_model.infer_image(frame), -1) # shape H x W x 1
        frame = np.concat([x for x in [frame, depth]], axis = 2) # shape H x W x 4
        frame = cv2.resize(src=frame, dsize=(256,256), interpolation=cv2.INTER_AREA)
        frame = torch.tensor(frame.transpose(2,0,1), # because cv2 and pytorch don't agree on axis orde
                             dtype=torch.float) # shape 4 x H x W

        # Get timestamp
        time = stamp(parser.isoparse(self.jsons[i][fidx]["gps_time"]))
        return frame, time

# Import ground-truth ground-speed data
with open("gt_data.json", "r") as file:
```

```
    gt_speed = json.load(file)

# Create data loaders
train_set = VideoSet(training_videos)
test_set = VideoSet(testing_videos)

train_loader = DataLoader(
    dataset = train_set,
    batch_size = BATCH,
    shuffle = False # loads videos sequentially == faster reads
)
test_loader = DataLoader(
    dataset = test_set,
    batch_size = BATCH,
    shuffle = False
)

# Create training function
def train(epoch, model, device, optimizer, data_loader, loss_function, gt_data):
    try:
        # Prepare model
        model = model.to(device)
        model = model.train()
        for batch_idx, (frame, time) in enumerate(data_loader):
            optimizer.zero_grad()
            frame = frame.to(device)

            # Get ground truth for comparison
            preds = []
            for st in time:
                if st in gt_data:
                    preds.append(gt_data[st][0])
                else:
                    preds.append(np.nan)
            y = np.array(preds).reshape(-1)
            y = torch.tensor(y, dtype=torch.float).to(device)

            # Calculate and record output & loss
            output = model(frame)

            # Ensure dimensions match:
            if output.shape == y.shape:
                loss = loss_function(output, y)
                loss.backward()
                optimizer.step()
            else:
                print(f"Dimension mismatch between prediction {output} and ground truth {y}")
                print(f" --> Skipping loss calculation for {time}")

            # Periodically report on training progress
            print(f"\rEpoch {epoch}: Training {batch_idx*BATCH}/{len(data_loader.dataset)} " +
                  f"(Loss: {loss.item():02.4})", end=" "*10)
```

```
        print(f"\rEpoch {epoch}: Trained {len(data_loader.dataset)}/{len(data_loader.dataset)} " +
                  f"(Loss: {loss.item():02.4})" + " "*10)
        return (loss, None)
    except KeyboardInterrupt:
        return (loss, KeyboardInterrupt)

# Create testing function
def test(epoch, model, device, data_loader, loss_function, gt_data):
    # Prepare model and data
    model = model.to(device)
    model = model.eval()
    test_loss = []
    map = []

    with torch.no_grad():
        for batch_idx, (frame, time) in enumerate(data_loader):
            # Load data into `device`
            frame = frame.to(device)

            # Get ground truth for comparison
            preds = []
            for st in time:
                if st in gt_data:
                    preds.append(gt_data[st][0])
                else:
                    preds.append(np.nan)
            y = np.array(preds).reshape(-1)
            y = torch.tensor(y, dtype=torch.float).to(device)

            # Calculate loss and accuracy
            output = model(frame)

            # Ensure dimensions match
            if output.shape == y.shape:
                test_loss.append(loss_function(output, y).item())
                map.append(torch.mean(torch.abs((output - y) / y)) * 100)
            else:
                print(f"Dimension mismatch between prediction {output} and ground truth {y}")
                print(f" --> Skipping loss calculation for {time}")

            # Periodically report on testing progress
            print(f"\rEpoch {epoch}: Testing {batch_idx*BATCH}/{len(data_loader.dataset)}, estimated MA
        print(f"\rEpoch {epoch}: Testing {len(data_loader.dataset)}/{len(data_loader.dataset)}" + " "*1

    # Report results
    test_loss = torch.mean(torch.tensor(test_loss))
    accuracy = torch.tensor(map)
    print(f"Test Result, epoch {epoch}: Avg loss {test_loss:04.4}, MAPE {torch.mean(accuracy):02.4}%")

    return accuracy


# Training/Testing Loop!
```

```
model = VisOdoNet()
optimizer = torch.optim.Adam(model.parameters())

if checkpoint_already_exists:
    checkpoint = torch.load(model_checkpoint, weights_only=True)
    model.load_state_dict(checkpoint['model_state_dict'])
    optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
    min_epoch = checkpoint['epoch']
    loss = checkpoint['loss']
    print(f"Loaded model from {model_checkpoint}, trained to epoch {min_epoch} with loss {loss.item():0
else:
    min_epoch = 1

for epoch in range(min_epoch, MAX_EPOCHS+1):
    loss = train(
        epoch=epoch,
        model=model,
        device=device,
        optimizer=optimizer,
        data_loader=train_loader,
        loss_function=F.mse_loss,
        gt_data=gt_speed["Tractor01"]
    )

    torch.save(
        {
            'epoch': epoch,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'loss': loss[0],
        },
        model_checkpoint)
    print(f"Saved model to {model_checkpoint}")

    if loss[1] == KeyboardInterrupt:
        print(f"Interrupted! Encountered {loss[1]}")
        break

    accuracy = test(
        epoch=epoch,
        model=model,
        device=device,
        data_loader=test_loader,
        loss_function=F.mse_loss,
        gt_data=gt_speed["Tractor01"]
    )
```