



Creating a Web-based Tool for Time Without Clocks
AI Model

Rashid Al-Marri

268649

BSc Computer Science Department of Informatics

Dr. Warrick Roseboom

2024

Statement of Originality

I hereby declare that this dissertation project is my work and that I have acknowledged all sources used in it. The AI model used in the project was originally developed by Dr. Warrick Roseboom. I have modified the structure of the model to make it suitable for use in an online setting. I have not copied or plagiarized any part of this work from any other person or source besides the use of the AI model from Dr. Warrick. I understand that any breach of this declaration may result in academic penalties or disciplinary actions. This work may be used by and distributed to future students so long as the proper credits are given to the original creator.

Rashid Almarri

Acknowledgments

I would like to give thanks to Dr. Warrick Roseboom for guiding me on many aspects of this project and for allowing me to use the AI model he created within the development process. I would also like to thank the University of Sussex for allowing me to create such an exciting and fulfilling piece of work. This has ignited a spark within me to continue on the path of web development and I am incredibly grateful for that.

Professional Considerations

This project does not have very many significant ethical considerations. After reviewing the Chartered Institute for IT's code of conduct I can confirm that I am adhering to sections 1 (Public Interest), sections 2 (Professional Competence and Integrity), and sections 3 (Duty to Relevant Authority). Full code of conduct can be found at: <https://www.bcs.org/membership-and-registrations/become-a-member/bcs-code-of-conduct/>. This project does employ the use of an AI model and video hosting. Therefore, I have considered the ethical and professional aspects of hosting this type of website and followed the best practices and standards of web development and data management. I have ensured that the website is secure, reliable, and user-friendly. The website meets the accessibility and compatibility requirements on a wide variety of different browsers and devices. I have also ensured that the user's data, such as emails, passwords, and videos are handled with the utmost privacy and that they are not stored or shared without their explicit consent. My use of the AI model is within the scope of the original work and its use has been sanctioned by its creator.

Abstract

Understanding the methods of how the human mind experiences time passing is at the forefront of psychology and neuroscience research. Dr. Warrick Roseboom and his colleagues at Time Storm have a working AI model, Time Without Clocks, that can accurately predict subjective time from video stimuli. The goal of this dissertation is to make a web-based tool for researchers to utilize the Time Storm model in a user-friendly and efficient manner. To achieve this, I was tasked with creating a modern front-end web application that can interact with the model, a scalable back-end that stores user's videos, information, and hosts the AI model, and finally, updating and fine-tuning the AI model to be more modern and efficient. The result was a 39x increase in processing speeds for the Time Storm model using a modern AI framework, PyTorch. The web application itself gives users the ability to utilize the Time Storm model with a key interest in ease of use and allows users to visualize the results of the model practically and aesthetically. With AI, researchers can have a better understanding of how the human brain works and processes stimuli by recreating the mind's processes with a deep neural network to simulate visual cortex processing, the passing of dynamic thresholds to simulate the role attention plays in subjective time, and finally, a regression model that converts the accumulation of silent changes from temporal units of time to seconds.

Front-End Deployment: <https://time-flies-frontend.vercel.app/>

Front-End Repo: <https://github.com/rashidalmarri21/Time-Flies-Frontend>

Back-End Repo: <https://github.com/rashidalmarri21/Time-Flies-Backend-Final>

Updated AI Model Repo: <https://github.com/rashidalmarri21/Time-Without-Clocks-PyTorch>

Original AI Model Repo: <https://github.com/timestorm-project/time-without-clocks>

Original Study Article: <https://www.nature.com/articles/s41467-018-08194-7>

Table of Contents

Statement of Originality	2
Acknowledgments	3
Professional Considerations	4
Abstract.....	5
1. Introduction	8
1.1 Project Overview	8
1.2 Motivations	8
1.3 Results and Achievements	9
2. Time Storm AI Model.....	9
2.1 Overview	9
2.1.1 Background	9
2.1.2 Methodology	10
2.2 Implementation.....	12
2.2.1 Background	12
2.2.2 Conversion: Caffe to PyTorch	12
2.2.3 Conversion: Demo to Production.....	14
2.3 Testing and Validation.....	18
2.3.1 Background	18
2.3.2 Training the Regression	18
2.3.3 Results	20
2.3.4 Speed Comparisons	22
3. Back-End Development.....	24
3.1 Overview	24
3.2 Functional Requirements.....	24
3.2.1 Overview	24
3.2.2 Database Model: User.....	25
3.2.3 Database Model: Video	25
3.2.4 CURD Operations: User	26
3.2.5 CRUD Operations: Video	27
3.2.5 User Authentication.....	28
3.3 Back-End Frameworks	29
3.3.1 Overview	29

3.3.2 Python Framework: Django	30
3.3.3 Python Framework: Flask	31
3.3.4 Conclusion	32
3.4 Implementation.....	33
3.4.1 Flask Initialization	33
3.4.2 Database Initialization	33
3.4.3 Video and User Routes	34
3.4.4 User Authentication.....	35
3.5 AI Model Integration	36
4. Front-End Development.....	37
4.1 Overview	37
4.2 Functional Requirements.....	38
4.3 Technology Stack.....	39
4.3.1 Options	39
4.3.2 Decision	40
4.4 Implementation.....	42
4.4.1 Figma Design	42
4.4.2 Next.JS + Tailwind CSS.....	43
4.4.3 Back-End Integration.....	46
4.4.4 Dynamic Graph Visualization	47
5. Deployment	49
5.1 Front-End Deployment	49
5.2 Back-End Deployment	50
6. Discussion	51
6.1 Results and Limitations.....	51
6.1.1 AI Model	51
6.1.2 Back-End	52
6.1.3 Front-End	52
6.2 Future Improvements	52
Conclusion.....	53
References.....	53
Appendices.....	55
Appendix A: Speed Test Specifications.....	55

1. Introduction

1.1 Project Overview

In this project, I was given the simple task of creating a robust web-based tool that allows researchers to utilize the Time Storm – Time Without Clocks (TS) AI model which was developed by Dr. Warrick Roseboom. The TS model was created to estimate the subjective time of video stimuli. TS accomplishes this by using novel machine-learning tactics such as image classification, changes in salient features, dynamic thresholds, and regression which can accurately estimate the amount of subjective time that has passed (Roseboom et al., 2019). Researchers can compare their findings using traditional approaches of temporal subjective time estimation to that of the TS model.

1.2 Motivations

While the University of Sussex is known for its AI program, I have always been more interested in web development. Bringing an idea to life in a web format has always been a passion of mine and to be able to work on such a robust project has been very inspiring. Bridging the gap between AI and web development has been, in recent years, at the forefront of emerging web-based technologies. The rise of large language models (LLM), exemplified by OpenAI's GPT, has sparked an "AI boom", with numerous major companies heavily investing in artificial intelligence across various sectors (Griffith & Metz, 2023). From LLMs like Google's Gemini (formally known as Bard) and Microsoft's CoPilot to image-based models such as Stable Diffusion and Midjourney. A lot of money is being poured into this sector for what was once a niche part of computer science reserved for big data and image classification. These new models aim to be brought to the masses in a user-friendly, prompt-based online environment, with the goal of putting these AI models into the hands of as many people as possible (Griffith & Metz, 2023). The motivation for this tool is for it to be utilized by a wide range of researchers, from neurologists to psychologists to help understand the temporal mechanisms of time

perception similar to the above-mentioned models, with a key interest in making sure the tool is easy to use and user-friendly. Unrestricted by the technical knowledge needed to run the AI model on their internal systems.

1.3 Results and Achievements

The website I've created is called Time Flies (TF) which follows best practices in user authentication, video hosting, and video analysis with a clean and user-friendly interface. The result of the analysis is also represented in an easy-to-read and concise way, allowing researchers to utilize TS in their workflow in an efficient manner. TF employs a handful of modern frameworks these include React for the frontend (using Next.js), Flask for the backend (a popular backend framework written in Python), and D3.js which is the gold standard for dynamic data visualization in JavaScript. Additionally, the TS model has been fine-tuned and updated to focus on speed of delivery by using modern AI frameworks such as PyTorch instead of the outdated and depreciated Caffe framework, greatly improving its efficiency.

2. Time Storm AI Model

2.1 Overview

2.1.1 Background

Time Storm – Time Without Clocks was developed by Dr. Warrick Roseboom in 2015 to use AI to predict the subjective time someone would feel when watching video stimuli. TS utilizes a feed-forward image classification network (AlexNet) to replicate human visual processing elegantly. The system tracks changes in salient features and accumulates them to produce an estimation (Roseboom et al., 2019). To convert the accumulations of salient changes to seconds, the data is fed into a pre-trained regression model that predicts the duration of the video stimuli (see Fig. 2.1).

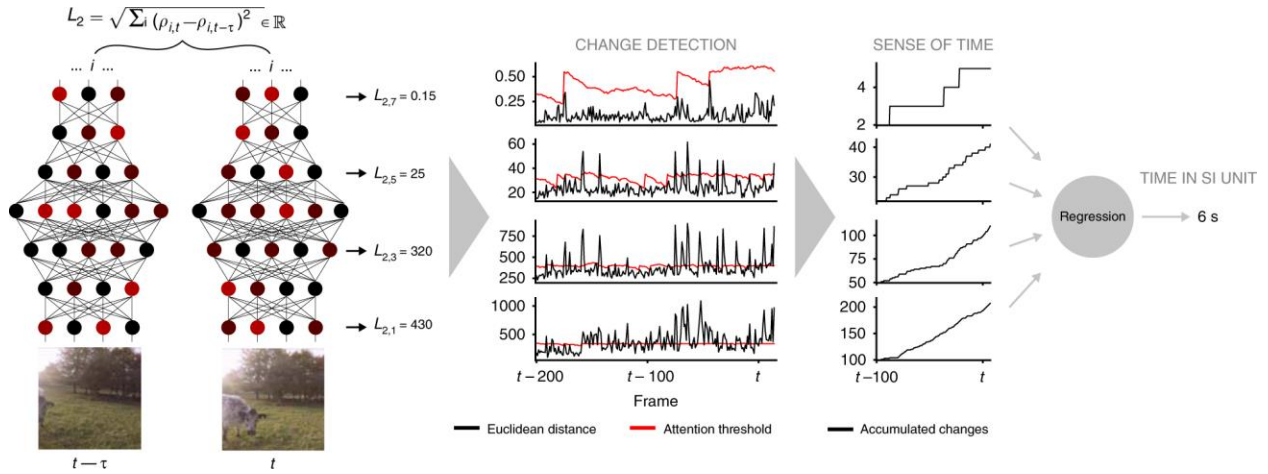


Figure 2.1: An overview of the TS model architecture. Reproduced from (Roseboom et al., 2019, Fig.2)

2.1.2 Methodology

The structure of the TS model is made up of four major components: an image classifier, a dynamic threshold system, accumulation calculations, and a regression model.

- Image Classifier (AlexNet):** The image classifier used to mimic human visual processing is AlexNet, a popular deep neural network originally used with the Python AI library Caffe, it has since been adapted to be used on more modern frameworks such as PyTorch. The classification process takes a frame of the video and attempts to classify based on a training set of 1000 features. TS extracts the activations from the conv2, pool5, and fc7 layers as well as the output probability then calculates the Euclidean distances between the current and previous states of each (Roseboom et al., 2019).
- Dynamic Thresholds:** Each layer has pre-defined thresholds (T_{\max} and T_{\min}) that slowly decay as each frame is fed into the system. When the Euclidean distance for a given layer exceeds the threshold, it resets to the T_{\max} value (see fig. 2.2). The variable C can be applied to the T_{\min} and T_{\max} values to emulate levels of “attention” where a value of “1” would be the baseline attention (100%) (Roseboom et al., 2019). For a more detailed explanation(see fig. 2.3.)

$$T_{t+1}^k = T_t^k - \left(\frac{T_{\max}^k - T_{\min}^k}{\tau^k} \right) e^{-\left(\frac{D}{\tau^k}\right)} + \mathcal{N} \left(0, \frac{T_{\max}^k - T_{\min}^k}{\alpha} \right),$$

Figure 2.2: “Where T_t^k is the threshold value of k th layer at timestep t and D indicates the number of timesteps since the last time the threshold value was reset. T_{\max}^k , T_{\min}^k and τ^k are the maximum threshold value, minimum threshold value and decay timeconstant for k th layer, respectively.” (Roseboom et al., 2019).

$$T_{\min}^k \leftarrow C \cdot T_{\min}^k \text{ and } T_{\max}^k \leftarrow C \cdot T_{\max}^k.$$

Figure 2.3: The formula for calculating the attention scaling C for T_{\max} and T_{\min} (Roseboom et al., 2019).

- **Accumulators:** Anytime the Euclidean distance of a given layer exceeds the dynamic threshold a unit of “subjective time” is added to a counter for that layer i.e. If layer 2, “pool5” exceeded the threshold, the current accumulator would look like this: {0:0, 1:1, 2:0, 3:0} and the threshold would be reset back to T_{\max} . This repeats for each frame in the video stimuli and what we are left with is the accumulation of “salient changes” at each layer (conv2, pool5, fc7, and output prob).
- **Regression:** The regression model is used to convert the accumulations of salient changes to seconds. To accomplish this SVR (Support Vector Regression) is trained on 33 videos split into clips ranging from 1s – 64s. Over 4290 accumulators from each of these trials are “fit” into the model with the X value being a 2D array of all the accumulators and the Y being a 1D array with their corresponding durations (see fig. 2.4). A K-Fold scheme was used for cross-validation to mitigate overfitting. The X and Y data were split across 10 “folds” where 10% of the data was reserved for testing and the rest used for training the model (Roseboom et al., 2019). Any new data can now be used with the trained model to predict what the duration is based on the data’s accumulators.

```
model = SVR(kernel='rbf', C=1e-3, gamma=1e-4)
model.fit(X_train, y_train)
```

Figure 2.4: The SVR regression model from the Python scikit-learn. In the original implementation of the model the kernel coefficient (*gamma*) of 10^{-4} and a “penalty of the error term” (*C*) of 10^{-3} (Roseboom et al., 2019).

2.2 Implementation

2.2.1 Background

While the original implementation of TS worked fine for the time with its use of Caffe-CUDA and the best GPUs for AI in 2015, in modern times there are a few issues with using the original setup. Caffe is now a depreciated framework, therefore GPU support is a non-starter, and it exclusively runs on Linux distros, particularly Ubuntu: 20.04. That isn’t such a big problem as virtual machines (VMs), and containerized environments are very robust and can accommodate even compute-heavy tasks such as image classification. I managed to get the model’s demo running on both a VM running Ubuntu and through Docker on my Windows machine. One commonality between both approaches was how incredibly slow the AlexNet classification step took to compute on the Caffe-CPU framework (< 1 frame per second). After quite the rabbit hole in trying to get the depreciated Caffe-CUDA working, I ended up deciding that re-building the model in a more modern, CUDA-compatible framework such as PyTorch would be the best approach.

2.2.2 Conversion: Caffe to PyTorch

The AlexNet class in the TS repository consists of the following: Model Initialization, Data Pre-Processing, Feature and Hook Extractions, and Post-Processing. For every step of the process, I will show a side-by-side comparison of how the conversion was implemented.

Model Initialization:

- **Caffe:** Initialization of the AlexNet requires specifying the structure of the model with a “deploy.prototxt” file and its pretrained weights “bvlc_alexnet.caffemodel” file before passing them into a Caffe’s classifier loader.
- **PyTorch:** PyTorch uses a more streamlined approach to initialization. By invoking the `torchvision.model.alexnet` method you can load all the necessary files in one line. The model then checks if CUDA is available and casts the model to the CPU if it isn’t.

Data Pre-Processing:

- **Caffe:** Caffe uses a function `set_transformer_for_opencv_webcam` to shape images to the desired shape by invoking the `caffe.io.Transformer` method. Then the method `.set_transpose` changes the order of the dimensions from (height, width, channels) to (channels, width, height) which is what the model expects.
- **PyTorch:** PyTorch requires a bit more processing to shape it to what the model expects. This includes resizing, center cropping, converting to tensor, and normalization of the image. The function `preprocess_image` is called on each frame and this is done on the GPU.

Feature Extraction and Hooks:

- **Caffe:** In Caffe, accessing the data at each layer is as simple as invoking the `.forward` method which allows the data to be grabbed directly. Making the saving and labeling of features novel.
- **PyTorch:** PyTorch handles things a little bit differently. Due to the use of its Dynamic Computation Graph, also known as define-by-run, it isn’t as simple as just accessing the data at each layer. In my implementation, I defined a `_register_hook` function that takes in a layer and name as its arguments. A lambda function is then invoked inside the PyTorch method `register_forward_hook`, which pulls the output data from each layer before saving the “handle” to a hooks list, ensuring the output is detached from the layer that way it doesn’t run indefinitely causing memory leaks.

Post-Processing:

- Both **PyTorch** and **Caffe** handle this part similarly. The logic pretty much remained the same where both load the class labels from the “alexnet/synset_words.txt” file and calculate the highest probability class as its prediction for that iteration. Finally, in both

cases, the data from the three layers as well as the output probability are saved to the features dictionary for later use in the updateNetworks class.

The updated AlexNet class leverages PyTorch's dynamic computation graph, modular design for data transformations, and efficient state management. The result is a more readable, maintainable, and efficient implementation, taking advantage of PyTorch's modern features such as forward hooks for feature extraction and automatic device management for execution on GPU or CPU.

2.2.3 Conversion: Demo to Production

The original TS repository has a demo script that shows off some of the capabilities of the model and in turn, lets the user see how some of the classes work simply and visually. While the implementation for the demonstration is far from being the same as the original study was conducted, notably the way it tracks time and accumulates the salient features, it offers a template of sorts to get started converting the system to my web-based needs. So, my next objective is to understand how these classes work and build a new script that will be the prototype for analyzing a one-off video. For a simple flow chart of how the original demo script works (see Fig. 2.5).

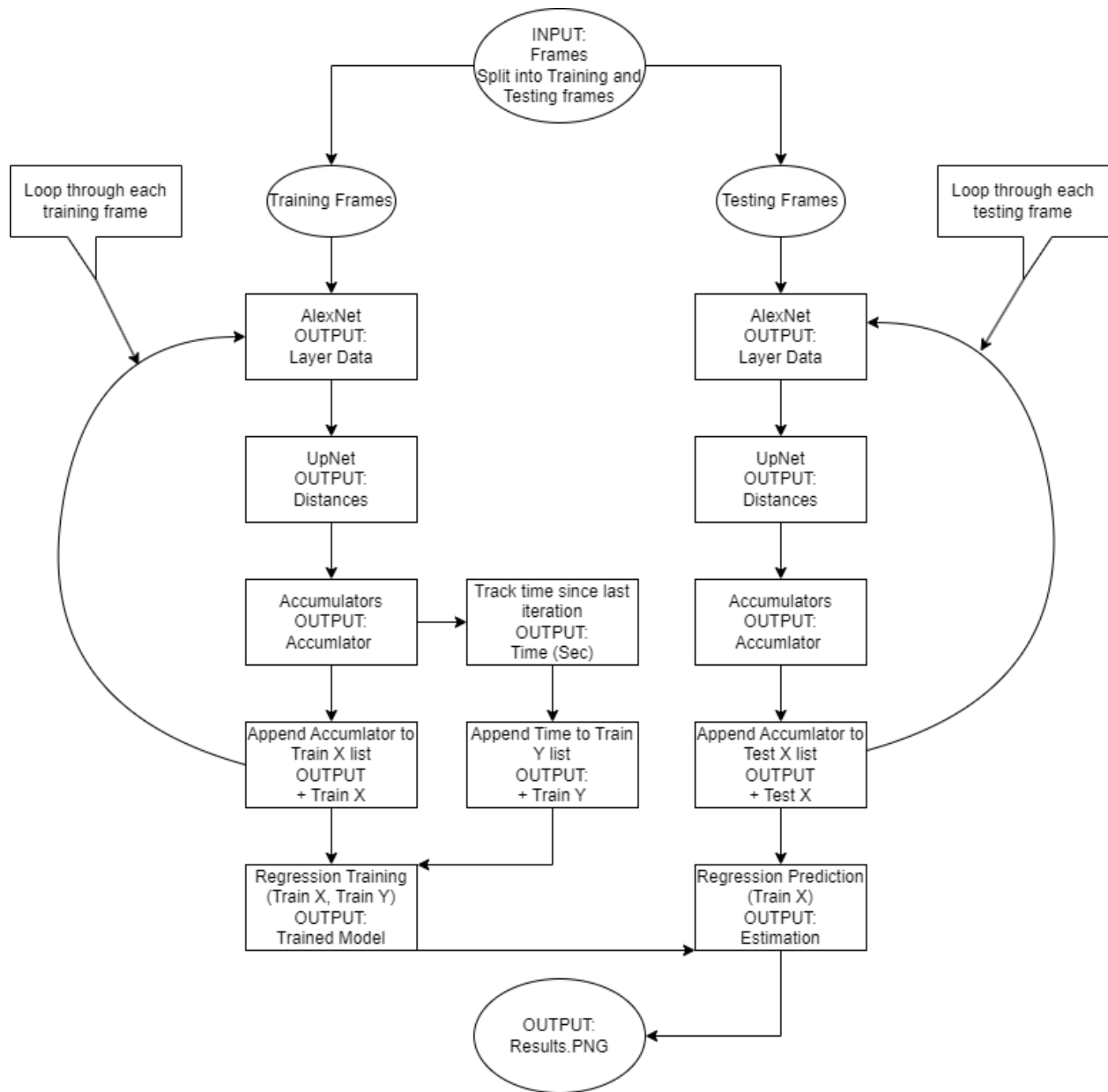


Figure 2.5: A flow chart of the original demo script from the TS repository.

The approach used by the demo script is to use the `updateNetworks` class to calculate and keep track of distances between AlexNet activations but does not use its internal accumulator and instead uses a separate custom class (`Accumulators`) for that function. The reason the demo did it this way is that, unlike the original study, the regression model isn't being pre-trained off a big data set of trials. So, to make the prediction as robust as possible, each frame is being treated as

a full trial. The accumulator class takes in a “time series” of distances for each layer and, like the `updateNetwork` class, calculates the accumulator. Only in this implementation, it doesn’t just update the accumulator once per iteration but for the entire “time series” of distances at each iteration. This is beneficial when working with a very small data set. However, this approach is strictly for demonstration purposes and is not ideal for a production environment as the results can be very misleading, especially if the first half of the video is drastically different from the second half.

The approach I chose to go with closely matches the original implementation that the study was written on. Instead of training a new regression model for each video, I pre-trained a model with the same data set from the original study, all 4290 trials, that way I can use all the frames in the video for the prediction phase. I will go into more detail about the training of the model in 2.3 Testing and Validation. By using a pre-trained model, I can essentially skip the “training” phase of the demo script and go directly to the “testing” phase. I also did not have to use the custom `Accumulator` class and just directly accessed the accumulator in the `updateNetwork` class thanks to the large dataset. A new class, `updateNetworkData`, was created to extract all the data from the `updateNetwork` class and package it into a JSON format to be sent along to the front end for dynamic rendering. To extend the scope slightly, I decided to implement the attention scaling factor, C , into the model at four thresholds: 50%, 100%, 150%, and 200%. The scaling factor directly affects the threshold calculations, which are done in the `updateNetwork` class. To reduce overhead, the changing of “attention” can be done after AlexNet processes the frame. By creating four instances of `updateNetwork` each with its internal scaling factor C updated, respectively, I can do all four calculations in parallel between each frame. This streamlines the process greatly and allows me to return four separate sets of results and graph JSON data files for each request. For a flowchart outlining how the production code works (see Fig. 2.6).

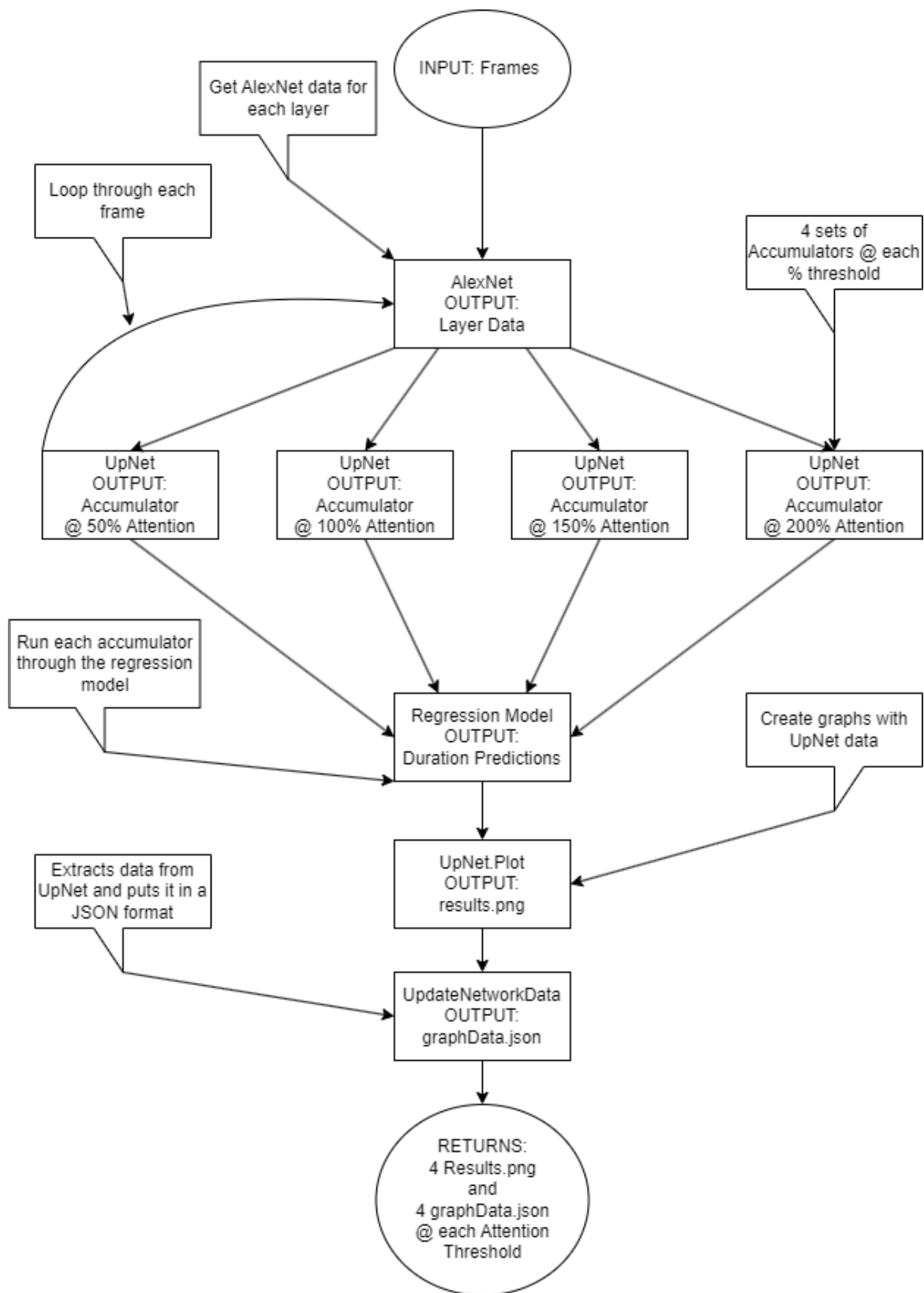


Figure 2.6: A flowchart of how the production code works.

2.3 Testing and Validation

2.3.1 Background

To ensure that the conversion of AlexNet from Caffe to PyTorch and the production code was implemented correctly I needed to cross-reference the results from my model with that of the original implementation. Luckily, Dr Roseboom has a repository containing all the estimates and distances between AlexNet activations made by the original model for each trial in a CSV file. This could allow me to completely skip the image classification step when training the model and what I am left with is the accumulators for each trial to feed into the regression for training. This would work if the way Caffe and PyTorch process the AlexNet model is the same. Unfortunately, there are very subtle differences between the two implementations which causes inconsistencies when using a regression model trained on Caffe data to predict estimates with the accumulators from a PyTorch implementation. So, to compare the estimates between implementations, I had to completely process the data from all 4290 trials and train the regression model on that instead.

2.3.2 Training the Regression

To train the regression model, I need to replicate the exact steps done in the original study to achieve results similar to the estimates in the repository. For starters, I needed to download all 33 videos used in the original study and split them into frames. Next, I needed to create “subsets” of frames for each trial based on the original 33 videos. Thankfully, Dr. Roseboom also had a CSV containing every trial’s start frame, end frame, and corresponding video ID in the file. So, it was as simple as creating a Python script to parse the CSV and appending to a dictionary the following: Trial ID as the keys and start frame, end frame, and video ID as the values. Once the CSV was processed, I could run each entry through a `calculate_accumulators` function (see fig. 2.7) that ran each trial through AlexNet and `updateNetworks` and returned the accumulator and duration of that trial then append that data to an `acc_combined` and `time_combined` list,

respectively. In the original Caffe implementation, this took ~10 days to complete whereas it only took ~40 hours to complete with PyTorch.

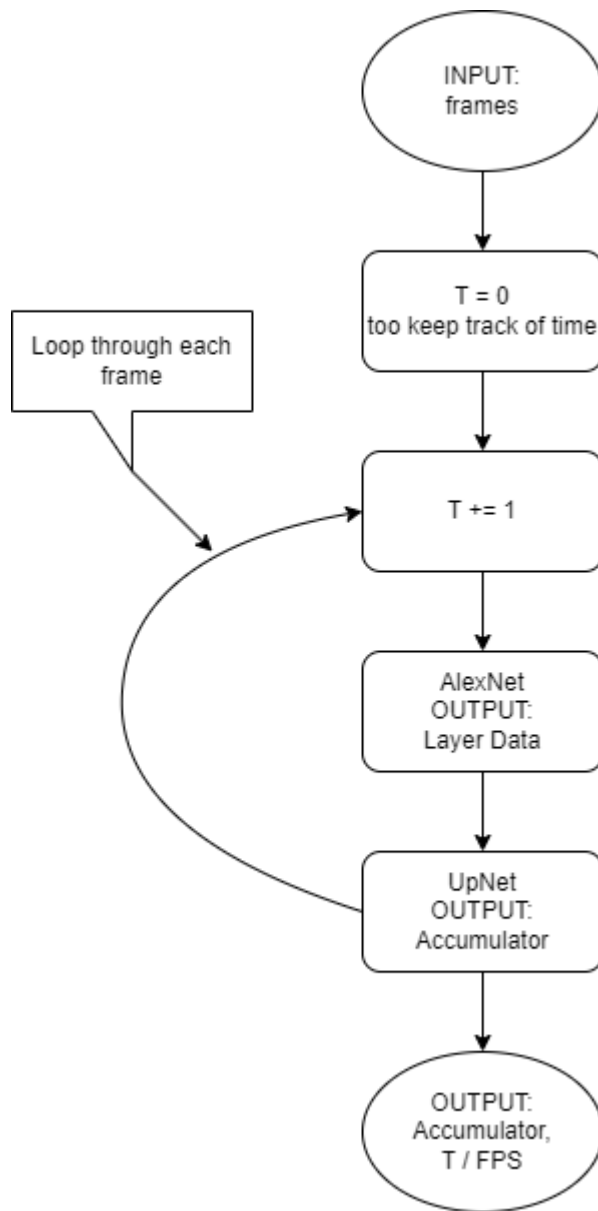


Figure 2.7: A flow chart of the calculate_accumulator function.

Once I had the acc_combined and time_combined lists, I converted the lists to NumPy arrays with the shape (4290, 4) and (4290,), respectively. These function as the X and Y data that the SVR regression model expects when training (Awad & Khanna , *Support Vector Regression* 2015).

To match the original study and prevent overfitting, instead of directly fitting all of the data into the SVG for training I implemented a 10-fold cross-validation scheme that splits the data into 10 “folds” reserving 10% of the data for testing. This allows me to access how well the model performs on new data by offering performance metrics averaged across all 10 folds which can confirm that the model generalizes well when tested with unseen data (Wong & Yeh, 2020). It also returns 10 separate models that I can use later to get a more accurate estimate by taking the average estimate across all 10 when predicting with new data. After fine-tuning the hyperparameters with a Grid Search, the top-performing model has these parameters: Kernel Coefficient (*gamma*) of $1e-4$ and a “term of error” (*C*) of $1e3$. What I was left with is a list containing all 10 regression models ready to be used in production.

2.3.3 Results

Now that I have a regression model pre-trained on all the same data from the original study, I can finally cross-reference my results with the results recorded by Dr. Roseboom. To do this, I used the same method to train the model, extended slightly to include the estimated duration and the human estimate from the CSV as well. Instead of tracking the time for each trial, I will use the pre-trained regression model to predict the time and record the results as the “actual estimate” for each trial. This allows me to create a JSON that contains four entries: Real Duration, Human Estimate, Expected Estimate, and Actual Estimate for each trial. This gives me enough data to properly evaluate if the PyTorch conversion and production code perform as well as the original implementation. I ran predictions on every trial that used video 9 and compared the human estimates to the actual estimates as well as the human estimates to the expected estimates (see Fig. 2.8). This gives an idea of how well my estimates compare to the original using a MAPE (Mean Absolute Percentage Error) score and MAE (Mean Absolute Error). Multiple runs of the same data can offer differing results to the MAE because it measures how far off the estimate is compared to the actual values. Due to the randomness of the model’s threshold system, multiple runs of the same data will produce different results which can be exaggerated by the MAE score. MAPE tends to be a more reliable metric across multiple data sets. This is due to how the MAPE score is produced. It does not consider magnitude when

comparing the estimates to the expected results. MAPE provides a percentage of how far off the entirety of the estimates and actual values are. It's impossible to get the exact results of the original study due to the random nature of the model's threshold system, but my implementation managed to stay within 10% of the original MAPE score across all runs. Both implementations tend to overestimate shorter videos and underestimate longer videos. This is consistent with the human estimates mentioned in the study (Roseboom et al., 2019).

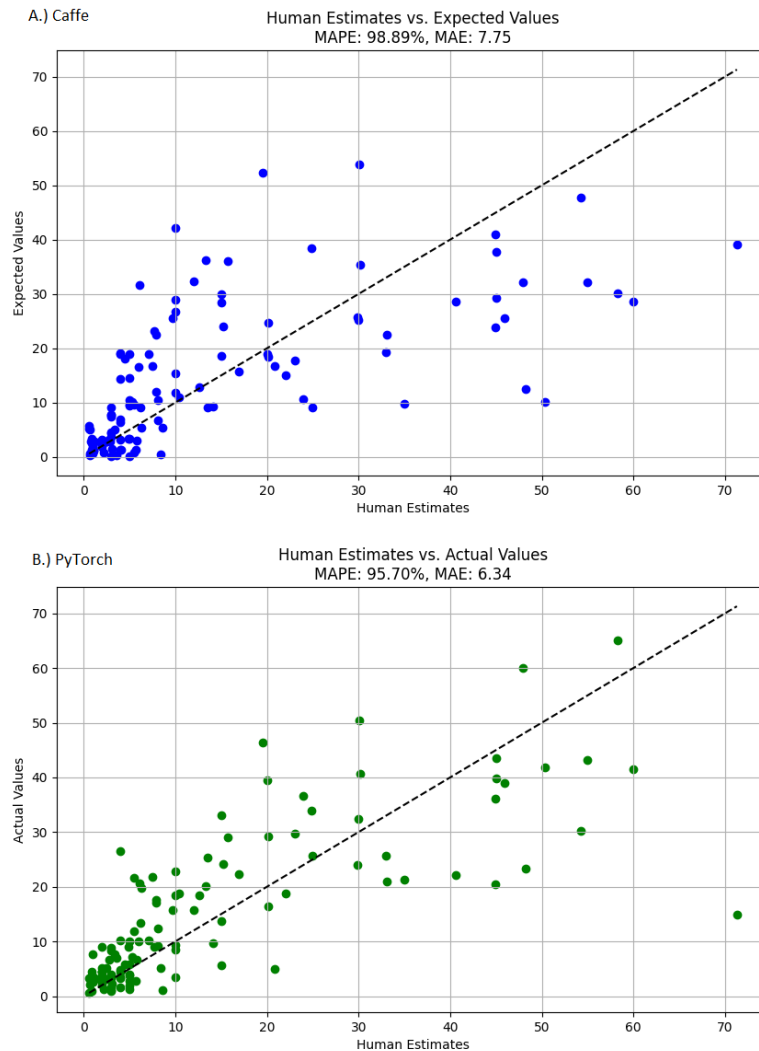


Figure 2.8: Data is from all 130 trials that used video 9. Graph **A** shows the estimates recorded by Dr. Roseboom in the original study vs the human estimates. Graph **B** shows the estimates I recorded from the updated model vs the same human estimates. My estimated MAPE (Mean Absolute Percentage Error) score is within 3% of the originals and is slightly closer to the human estimates in this particular test.

2.3.4 Speed Comparisons

With the updated PyTorch implementation of the AlexNet model, a video can now be processed up to 2439x faster than using Caffe-CPU (see fig. 2.9). The Caffe-CPU implementation (since Caffe-CUDA is deprecated) would process videos at ~0.5 frames per second whereas the PyTorch-CUDA implementation can process the videos at ~1,060 frames per second. This is a necessity when the goal is to allow people to use this tool online. A one-minute video in the old implementation that would have taken over 60 minutes to process, now only takes about 2 seconds. These tests and numbers exclude all the other components of the AI model. This is only for the frame processing phase using AlexNet. This proves that the bottleneck for the system certainly isn't the AlexNet implementation.

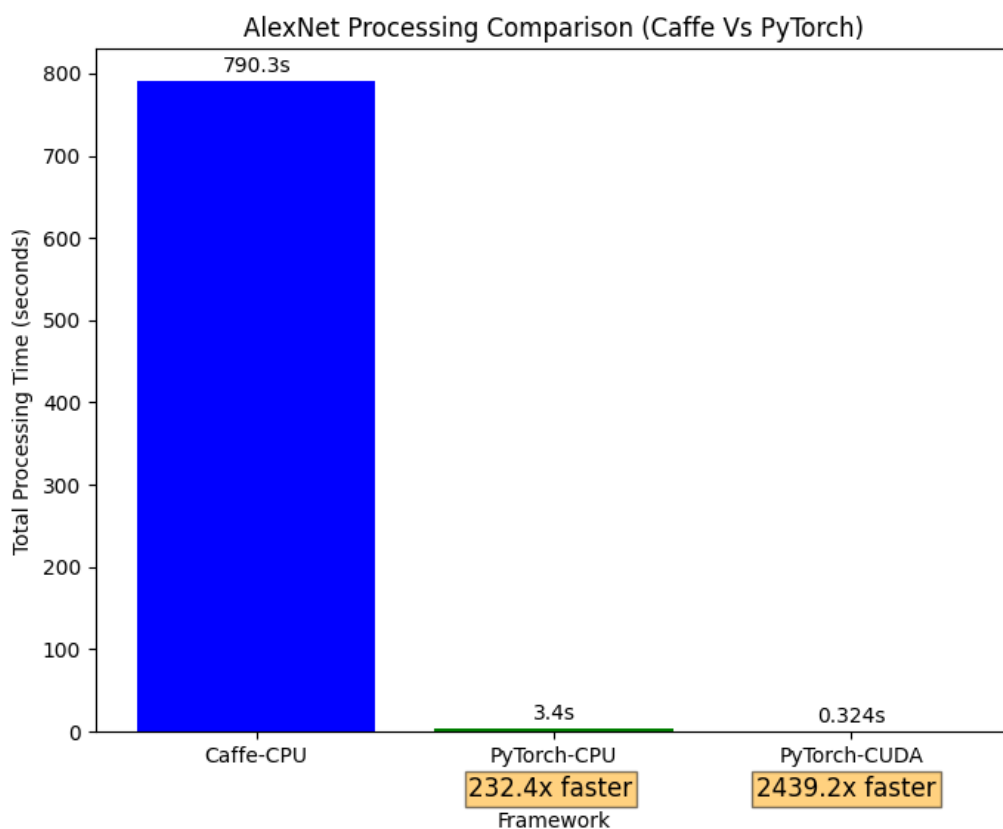


Figure 2.9: A comparison of Caffe-CPU vs PyTorch-CPU vs PyTorch-Cuda in processing frames with AlexNet. The test was run on a separate implementation of AlexNet with an 11-second video. This excludes any other components of TS as well as frame conversion.

For more realistic metrics on how the model performs when all the systems of TS are utilized, I created a testing script that takes the same 11-second video, runs it through the production code, and gauges the time of TS in its entirety. This gives greater insights into which framework performs the best in a real-world situation. The disparity between Caffe and PyTorch is greatly reduced in this example due to the processing overhead of the updateNetworks instances and frequent IO calls (which were all excluded in the previous test). That being said, PyTorch is still beating out Caffe by at worst, 29.6x and at best, 39.1x (see fig. 2.10). Although, one interesting insight gathered from this particular test is that PyTorch-CPU tends to slightly outperform PyTorch-CUDA in this real-world test. This is most likely due to the transfer of data from CPU to GPU and back between each frame and if I were to implement batch-processing on the GPU this disparity would be less pronounced. See Appendix A for more details on test specifications.

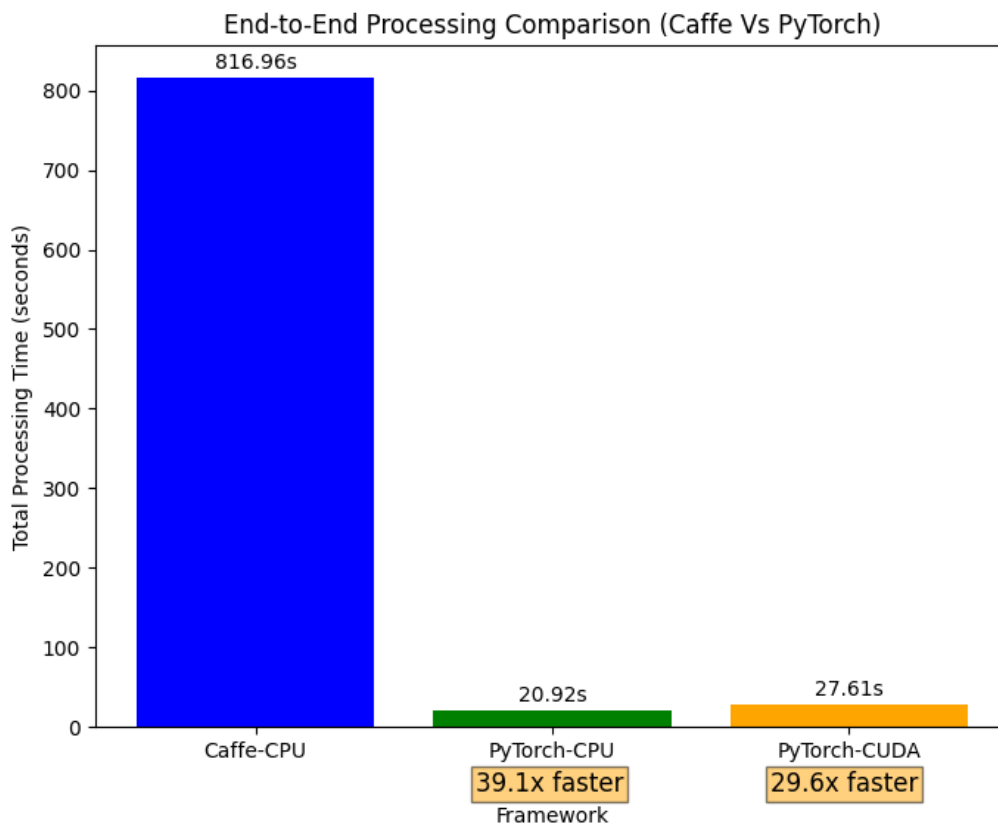


Figure 2.10: A comparison of Caffe-CPU vs PyTorch-CPU vs PyTorch-CUDA in total end-to-end processing time with the production code. This test was run on the production version of TS with the same 11-second video. The time reflects the model's performance excluding the frame conversions.

3. Back-End Development

3.1 Overview

The back end is an essential component of all web applications. It is responsible for data management, user authentication, input validation, receiving requests, and in our case hosting the AI model that will be used by the client (see fig. 3.1). There are 3 main components to a back-end: the server, the application, and the database (Muittari, 2020). The server is the platform that the application runs on. The application is responsible for handling the logic, AI models, API (Application Programming Interface) requests, and communicating with the database. The database handles the retrieval and storage of data that the application needs to function (see Fig. 3.1). Usually, the database has models for each object type that has data. For example, in the case of a video hosting website, the backend models may consist of a User and Video class each with attributes that help keep track of their respective information.

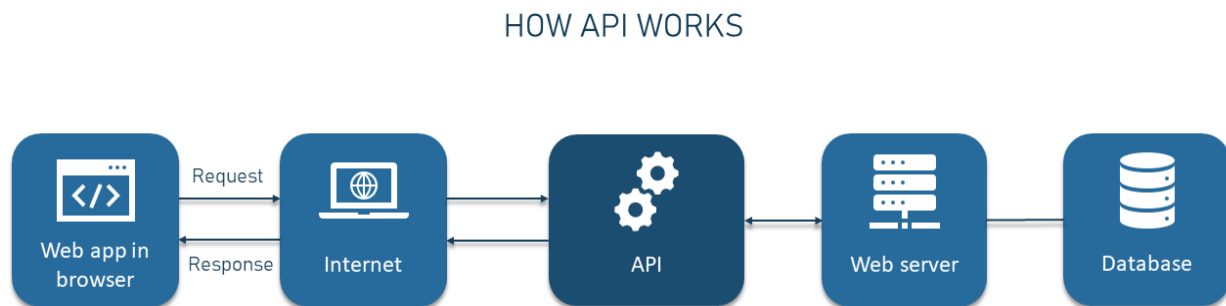


Figure 3.1: How API works (Hygraph Team, 2022).

3.2 Functional Requirements

3.2.1 Overview

The functional requirements for the back-end of a web application of this nature are as follows: Database models for User and Video, CRUD (Create, Read, Update, and Delete) operations for User and Video data, user authentication, and AI model integrations.

3.2.2 Database Model: User

To achieve the desired requirements for a User database model, the model must contain the following attributes:

- **ID:** The id attribute of this model will be the primary identifier for any user on the web application. It should be an integer and the primary key.
- **Name:** The name attribute of this model contains the full name of the user and should be limited to 50 characters.
- **Email:** The email attribute of this model contains the email address associated with the user. It should be limited to 100 characters and be a unique entry. In other words, once an email address is assigned to a user no other user can use the same address.
- **Password:** The password attribute of this model contains a hash-able string of the user's password. It should be at least 6 but limited to 100 characters. Additional requirements include at least one number and one special character i.e. (!, @, #, \$, %, ^, &, *, _ , -, =, +, ` ,)
- **Is Admin:** The 'is admin' attribute of this model is a Boolean flag that contains True or False input. By default, the flag should be set to False.
- **Profile Picture:** The profile picture attribute of this model contains a string that is the path to an upload folder containing the user's profile picture and can contain a 'null' value. By default, the string will be set to 'default.png' if it is 'null'.
- **Videos:** The video attribute of this model contains all the videos a user has uploaded. It should define a relationship to the Video database model. It should make a back reference 'Author' and add this attribute within the Video database.

3.2.3 Database Model: Video

To achieve the desired requirements for a Video database model, the model must contain the following attributes:

- **ID:** The id attribute of this model will be the primary identifier for any video uploaded to the web application. It should be an integer and the primary key.

- **User ID:** The user ID attribute of this model should contain an integer of the ID of the user who uploaded the video. This should be a foreign key within the User database model.
- **Title:** The title attribute of this model should be a string containing the name of the video file that has been uploaded. It should be limited to 100 characters. By default, the title will be the full file name of the video extensions included. This attribute should be editable.
- **Description:** The description attribute of this model contains the full description of the video as an uncapped Text type and can be 'null'.
- **Upload Date:** The upload date attribute of this model contains the timestamp of when the video was uploaded. It should automatically be added to the database on upload.
- **File Name:** The file name attribute of this model contains the full file name of the video, extensions included i.e. (example.mp4). It should be a string with a limit of 255 characters.
- **Banner:** The banner attribute of this model contains the path to an image that holds the very first frame of the video. This should automatically be created on upload and stored in the server's upload folder.
- **Results:** The results attribute of this model contains the image path that holds the raw .png output of a video analyzed by the AI model. It should be a string with a limit of 255 characters and it can contain a 'null' value.
- **Graph JSON Data:** The graph JSON data of this model contains a path to the .json file associated with the video. This data will be used to visualize the data of a video after it has been run through the AI model. It should be a string with a limit of 255 characters and can contain a 'null' value.

3.2.4 CURD Operations: User

To achieve the desired requirements for CURD operations on the User database, the following HTTP routes must be implemented:

- **Create User:** The HTTP route for creating a user should be a POST method. The data sent along with the request should contain the user's Name, Email, and Password. Validation checks of the user's password to make sure it meets the above-mentioned requirements and email to ensure it is not already in use. Before committing the new user to the database, a hashing function should be run on the password to securely store it.

- **Get User:** The HTTP route for retrieving a user's data should be a GET method. The back end should do an authentication check to see which user is currently logged in by checking the credentials header of the request. After verification, the route should send back a JSON file containing the user ID, Name, Email, and a True or False for their admin status.
- **Update User:** The HTTP route for updating a user's data should be a POST method. The data sent along with the request must contain a credentials header and the user's Name and/or Email to be updated. Before committing the changes to the database a user authentication check must be done.
- **Update Password:** The HTTP route for updating a user's password should be a POST method. The data sent along with the request should contain a current password and a new password field. A validation check must occur to ensure the current password matches the hashed password saved to the database. If the current password is valid, then the user's password should be updated to the new password before committing the changes to the database.
- **Update Profile Picture:** The HTTP route for updating a user's profile picture should be a POST method. The data sent along with the request should contain a file field that holds the new profile picture. A validation check is done to ensure the extension of the file is allowed i.e. (png, jpg, jpeg, gif). If the file is valid, then the file is saved to the server's upload folder before updating the user's profile picture to the path of the newly created entry.
- **Delete User:** The HTTP route for deleting a user should be a DELETE method. Before the user can be deleted from the database some cleanup should occur. All the statically stored files such as the user's videos, video banners, and profile pictures need to be deleted.

3.2.5 CRUD Operations: Video

To achieve the desired requirements for CURD operations on the Video database, the following HTTP routes must be implemented:

- **Upload Video:** The HTTP route for uploading a video should be a POST method. The data sent along with the request should be a file, title, a description. The only mandatory field is the file. If no title is given it should default to 'Untitled' and the description can be a null value as defined in the Video model.
- **Update Video:** The HTTP route for updating a video's title should be a POST method that also has the video's ID in the query address. The data sent along with the request should

be in a new title field. The video associated with the URL's id should then be updated before committing the changes to the database.

- **Get Video:** The HTTP route for retrieving a single video's data should be a GET method that also has the video's ID in the query address. An authentication check should be done to make sure the video the user is trying to retrieve belongs to the current user before returning the video's data.
- **Get Videos:** The HTTP route for retrieving all the videos of the current user should be the GET method. An authentication check should first be done to see which user is logged in before returning a list of every video by that author.
- **Serve Video:** The HTTP route for serving a video should be a GET method that also has the video ID in the query address. The data sent from the video player's request should have range headers to handle sending 'chunks' of the video, that way the user does not have to wait for the full video to load before watching. As well as cache control headers to clear any previous video data from the pipeline.
- **Delete Video:** The HTTP route for deleting a video should be a DELETE method that also has the video's ID in the query address. An authentication check should be done to ensure the video ID being deleted belongs to the current user. Before deletion, some cleanup needs to be done. The video file, banner, and (if any) result images or JSON data need to be deleted before committing the changes to the database.

3.2.5 User Authentication

The user authentication requirement is very important for the operational security of a web application. Allowing unauthorized users to do CRUD operations on other people's data would leave your application vulnerable to malicious users. The back end should have some way of verifying that the user sending a request is who they say they are. The following are some options that can be employed to accomplish this:

- **Basic Authentication:** This simple method takes the user's email and password and encodes it in Base64 which gets sent to and from the server via headers in HTTP requests. This isn't a very good option as it isn't very secure. If the headers were to be intercepted by malicious users, then the Base64 encoding can be easily decoded leaving the user compromised (Reschke, 2015).
- **Certificate-Based Authentication:** Considered one of the most secure ways to authenticate users. It requires an authentication server that must be maintained, usually

by a third party. It employs digitally signed certificates that have a public key that must be verified by the authentication server's private key before access (Hummen et al., 2013). While this would be a very secure and scalable way to implement user authentication, it doesn't seem to fit the needs of a basic, free-to-use web application as it would require too much time, maintenance, and money to deploy.

- **Session-Based Authentication:** Session-based authentication is a system that generates a session ID on a successful login that gets stored as a cookie in the user's browser and the server's Session ID state. Any follow-up request while the cookie is active in the browser can automatically verify the user, if the session ID is still valid in the server, making session-based authentication very convenient when dealing with REST-ful APIs like the one employed for this website (Balaj, 2017). In terms of security, the session cookies are encrypted and signed, and in most cases are sent over HTTPS making them a very secure option as well.

Session-based authentication seems to be the best fit for the web application I'm deploying due to its ease of use without compromising on security.

3.3 Back-End Frameworks

3.3.1 Overview

A programming language framework is a library of functions, APIs, and structures that usually are combined to solve a specific task (Ghimire, 2020). Choosing the right framework for a project is a fundamental step in the design process, as there are usually a variety of flavors to choose from. Making the right decision comes down to what goals you are trying to achieve in your application (Ghimire, 2020). In the case of back-end development, the first decision you should make is what programming language or languages you should use. Every modern programming language has some sort of back-end framework you can implement. Some popular language options include:

- **JavaScript:** Express.js, NestJS, and Koa.JS
- **Python:** Django, Flask, and FastAPI
- **Ruby:** Ruby on Rails and Sinatra
- **Java:** Spring Boot and Hibernate

For the back-end development of a video hosting website that employs the use of an AI model that is written in Python, the choice is easy. Why not use the same language the AI model uses for convenience? It does help that the Python options are very popular frameworks as is (see Fig. 3.2), so for my application, python seems the obvious decision.

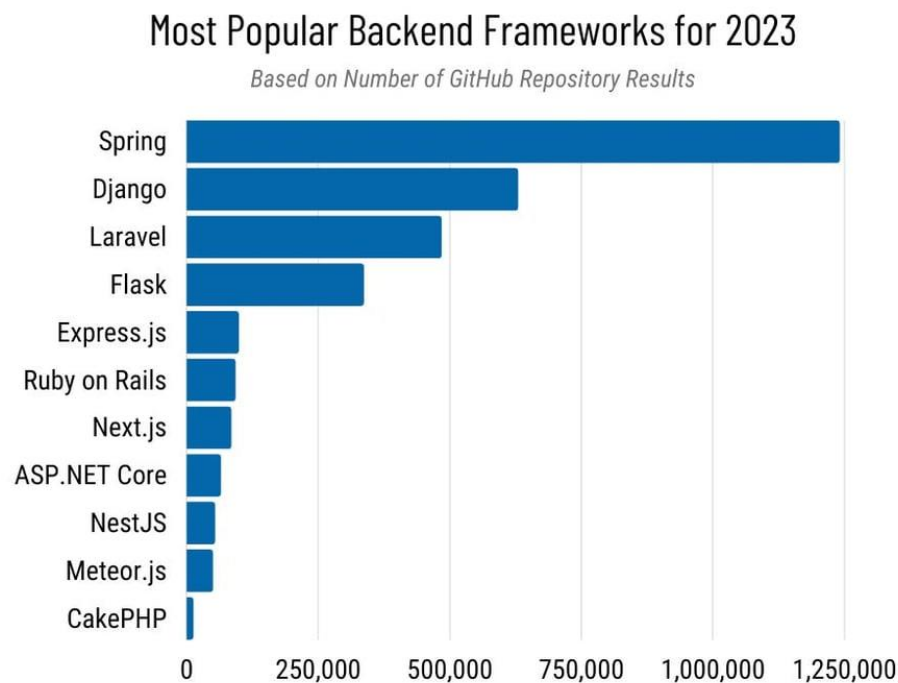


Figure 3.2: Most Popular Backend Frameworks for 2023. Reproduced from (Desmond, 2023).

3.3.2 Python Framework: Django

Django is a high-level Python web framework that focuses on speed and clarity of development. Developed in 2003 by Adrian Holovaty and Simon Willison, Django was created to assist in the creation of complex websites for the newspaper they worked at. Its “batteries included” approach offers a multitude of tools “pre-baked” into the framework, ideal for rapid development (Holovaty, 2009).

Advantages:

- **Batteries-Included:** Django follows a "batteries-included" approach, offering a wide array of built-in features, such as an ORM (Object-Relational Mapping), authentication, and a template engine. This can significantly speed up development time for complex applications.
- **Highly Scalable:** Django has built-in support for scaling applications, making it suitable for high-traffic websites.
- **Security:** Django offers robust security features that protect your web application from common attacks, such as SQL Injection, cross-site scripting, cross-site request forgery, and clickjacking.
- **Mature Ecosystem:** Django has a large community and a mature ecosystem, providing extensive documentation, third-party packages, and support.

Disadvantages:

- **Steep Learning Curve:** The comprehensive nature of Django can be overwhelming for beginners. Understanding its many components and their interactions takes time.
- **Less Flexible:** Since Django uses a "batteries included" approach to development, it makes it very hard to customize the built-in features. You must follow the structure Django provides, in most cases.

3.3.3 Python Framework: Flask

Flask is considered a lightweight alternative to the, at times, cumbersome Django framework. Flask was developed by Armin Ronacher in 2010 as an April Fool's joke but despite its curious inception, Flask rapidly gained popularity amongst Python enthusiasts for its simple and flexible approach to back-end development (Educative.io Team, 2020). Unlike Django's "batteries included" approach, Flask employs the "micro-framework" philosophy, meaning it provides the bare minimum necessary to get an application up and running. This approach offers fine control over the features and components a developer can use on an "as-needed" basis (Educative.io Team, 2020).

Advantages:

- **Simplicity and Flexibility:** Flask provides simplicity, making it more flexible for developers to use the right tools for their projects. It is a “micro-framework” that is designed to be lightweight and easy to extend.
- **Ease of Learning:** Flask has a smaller learning curve than Django, making it a good option for beginners or for developers who prefer to build their applications piece by piece.
- **Mature Ecosystem:** Like Django, Flask also has a very robust community of contributors building third-party libraries, such as ORM (Object Relationship Mapping), User authentication, and other extensions.

Disadvantages:

- **Additional Setup:** Due to the “micro-framework” approach, Flask requires a lot more setup for more advanced components. Although, thanks to Flask’s dedicated community a lot of these cumbersome boilerplates have been abstracted to extensions and third-party libraries.
- **Scalability:** While Flask applications can be scaled, it might require more effort to set up the necessary infrastructure compared to Django, which has built-in functionalities designed for scalability.

3.3.4 Conclusion

In conclusion, both Django and Flask are fine choices for the web application I am building. Both offer all the requirements necessary for an application of this nature. It comes down to personal preference as I am more familiar with the Flask paradigm and learning the Django framework would be too cumbersome. Therefore, I chose to use Flask as the back-end framework for the web application.

3.4 Implementation

3.4.1 Flask Initialization

Setting up a Flask environment was very simple. All that is needed is a main.py, models.py, and a config.py file to initialize the Flask server (see Fig 3.3). Since we are expecting requests from outside the flask server and not only internally, we must wrap our flask app with CORS (Cross-Origin Resource Sharing)(see Fig 3.3). This allows content and authentication requests to be sent to and from the front end freely. Once initialized you can start making HTTP routes directly in the main.py file. While this is not the only setup needed for all components of the back-end it goes to show just how simple Flask is to get a server up and running.

```
app = Flask(__name__)
CORS(app, supports_credentials=True)

db.init_app(app)

if __name__ == '__main__':
    with app.app_context():
        db.create_all()
    app.run(debug=True)
```

Figure 3.3: The initialization of the Flask back-end server.

3.4.2 Database Initialization

Setting up the database was also surprisingly simple. The engine I chose to go with was SQLite and to interface with this database easily Flask has a convenient extension, Flask SQLAlchemy, which needs minimal setup to get started. All that is needed is a models.py file that contains the initialization of the SQLAlchemy database variable (db) and classes for each database object (see Fig. 3.4), which fulfills the database requirements. Now main.py imports the db variable before initializing it within the app(see fig. 3.3).

```

db = SQLAlchemy()

# User model
class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)
    fullname = db.Column(db.String(50), nullable=False)
    email = db.Column(db.String(100), unique=True, nullable=False)
    password = db.Column(db.String(100), nullable=False)
    is_admin = db.Column(db.Boolean, default=False)
    profile_picture = db.Column(db.String(255), nullable=True, default='default.png')
    # Define a relationship to videos
    videos = db.relationship('Video', backref='author', lazy=True)

# Video model
class Video(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(100), nullable=False)
    description = db.Column(db.Text, nullable=True)
    upload_date = db.Column(db.DateTime, nullable=False, default=db.func.current_timestamp())
    filename = db.Column(db.String(255), nullable=False)
    banner = db.Column(db.String(255), nullable=True)
    results = db.Column(db.String(255), nullable=True)
    graphJSONData = db.Column(db.String(255), nullable=True)
    # Foreign key to link the video to its author (user)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)

```

Figure 3.4: The structure of the User and Video database models.

3.4.3 Video and User Routes

Instead of having our video and user HTTP routes directly in the main.py file and to practice the separation of concerns paradigm, two more Python files were created: video_routes.py and user_routes.py. To fulfill flask's requirements, we must create blueprints of these routes and initialize them within the main.py file, appending their respective URL prefixes (see Fig. 3.5 and 3.6). Within those route files, we can fulfill all the CRUD requirements necessary for the Video and User databases.

```

videos_bp = Blueprint('videos', __name__)
users_bp = Blueprint('users', __name__)

```

Figure 3.5: Creating the blueprints for the video and user blueprints within their respective .py files.

```

app.register_blueprint(users_bp, url_prefix='/users')
app.register_blueprint(videos_bp, url_prefix='/videos')

```

Figure 3.6: Registering the User and Video blueprints within the main.py file.

3.4.4 User Authentication

Implementing user authentication in Flask requires the third-party extension Flask Login to handle session management. Flask Login uses a session-based authentication system where on a successful login a session ID is created for the user (see fig. 3.6). The session ID is saved as a cookie within the browser the login request was sent from. A special decorator (`@login_required`) is added to routes that need user authentication to be accessed (see Fig 3.7). This works by checking the 'credentials' header of the request which stores the session ID cookie. The 'credentials' ID is then cross-referenced with all the user objects that currently are signed in on the server. On a successful logout, the server-side user object that stores the session ID is deleted revoking access to requests sent with that ID.

```
# Route for user login
@users_bp.route('/login', methods=['POST'])
def login():
    data = request.json
    if 'email' not in data or 'password' not in data or not data['email'] or not data['password']:
        return jsonify({'error': 'Missing email or password'}), 400

    user = User.query.filter_by(email=data['email']).first()
    if not user or not check_password_hash(user.password, data['password']):
        return jsonify({'error': 'Invalid email or password'}), 401

    login_user(user)
    session['user_id'] = user.id
    return jsonify({'message': 'Logged in successfully'}), 200
```

Figure 3.6: The highlighted code shows where the session ID is created in the login process.

```
# Route for user logout
@users_bp.route('/logout', methods=['POST'])
@login_required
def logout():
    logout_user()
    return jsonify({'message': 'Logged out successfully'})
```

Figure 3.7: Shows the `@login_required` decorator required for any route that needs user authentication.

3.5 AI Model Integration

Integrating the AI model with the backend requires an HTTP route that calls the `analyze_video` function which takes in a path to a folder containing all the frames of the video and returns the results and graph data to the appropriate folders. But before we can call the function, we must populate the frames folder with all the frames of the requested video. To accomplish this, a video ID is added to the query address when the request is sent. The video is then pulled from the uploads folder of the server and split into frames, creating, and populating a 'videoID'_frames folder(see fig. 3.8). Now we can run the `analyze_video` function and receive two dictionaries: `figures` and `json_data`. The figure is the raw results graph (see Fig 3.9) created by PyPlot and `json_data` is all the data used to populate the pyplot graph. The figure is saved to the uploads/ai-results folder on the server with its name that of the video IDs (i.e. 1.png) and the `json_data` is saved similarly in the uploads/jsonGraphData folder. The `json_data` is to be used in the front end to dynamically visualize the data.

```
@videos_bp.route('/analyze/<video_id>')
def analyze(video_id):
    video = Video.query.get(video_id)
    if video:
        # Construct the file path
        #video_title_with_underscores = video.title.replace(" ", "_")
        video_path = os.path.join(current_app.config['UPLOAD_FOLDER'], video.filename)
        print(video_path)

        save_dir = f"{video_id}_frames"
        .....convert_video_to_frames(video_path, save_dir)

        figure, json_data = analyze_video(f"{video_id}_frames")
```

Figure 3.8: Highlighted code converts the video at the path to frames then creates and populates the frames folder.

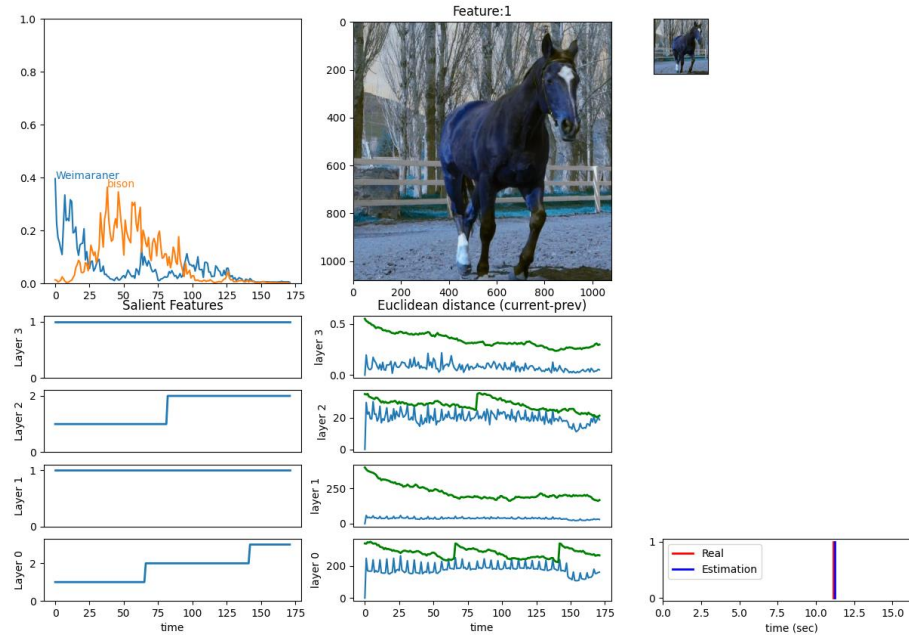


Figure 3.9: Raw graph data plotted by PyPlot.

4. Front-End Development

4.1 Overview

The front end serves as the window in which the end user can interact with a web application. Ensuring the user experience is as robust and non-invasive as possible is paramount to the success of any website. The identity of the front end in its simplest form is a platform in which the user can fetch, write, and interact with data and services from the back end. The web pages themselves are made up of HTML with some form of CSS styling and to incorporate interactivity, JavaScript is usually what modern developers reach for (Gallinelli, 2024). To communicate with the back end, each component of the front end can utilize a range of HTTP request calls to the server to bring some life to the static HTML pages.

4.2 Functional Requirements

To achieve the desired functional requirements for the front-end of this web application these components must be implemented: Navigation Bar, landing page, about section, FAQs, sign-in/register page, upload video, review previously uploaded videos, analyze/results page, and a user settings page. Other non-functional requirements include optimizations for speed, smooth user interfaces, and seamless back-end integration.

Components:

- **Navigation Bar:** The navigation bar has the website's logo, links to the home, about, and FAQs pages as well as prompts for users to either sign in or register and their profile picture and user information when signed in.
- **Landing Page:** The landing page consists of 3 sections: The hero, the product overview, and a call to action. First, the hero section should catch the eye of the user and make them interested in learning more about what the website offers. Next, the product overview tells the user all the features or the "selling point" of the website. Finally, the call to action tells the user how to get started and links them directly to the register and about pages.
- **About Page:** The About page delves deeper into how exactly the AI model works and goes into more detail about all the offered features. The about should also contain a hero with a call to action, that way the user can start using the product right away if they choose.
- **FAQs:** Before the footer of both the Home and About pages contains six frequently asked questions that have smooth accordion pop-outs for each one. This should answer some pressing questions a user may have or quell any reservations about using the service.
- **Sign-in/Registration Page:** The sign-in and registration page is simple and to the point. Offers users a quick registration form with minimal requirements and a remember me button on sign-in for quick re-access in the future.
- **Upload/Review Videos Page:** This page contains the portal in which a user can upload videos to the AI model. Toast messages guide the user through the process ensuring a smooth transition from upload to the analyze/results page. A tab for previously uploaded videos is also present if a user wishes to see older results.

- **User Settings Page:** The user settings page is accessed by clicking the user information element in the website's header. This page allows the user to update their profile picture, user information, password, email address, and delete their account if they choose.

4.3 Technology Stack

4.3.1 Options

Choosing the right technology stack for front-end development can be a very overwhelming endeavor. Even though there are only three main languages for front-end development: HTML, CSS, and JavaScript. There are dozens of frameworks for each language that can be mixed and matched to fit your custom needs or preferences. A few of the most popular options are listed below for each:

HTML (HyperText Markup Language)

- HTML5
- XHTML
- HTML Imports
- SVG

CSS (Cascading Style Sheets)

- CSS3
- Bootstrap
- Foundation
- Tailwind CSS
- PostCSS

JavaScript (TypeScript for type safety)

- jQuery
- React Native
- Angular

- Vue.js
- Next.js
- Vite.js

For a front-end framework flowchart (see fig. 4.1).

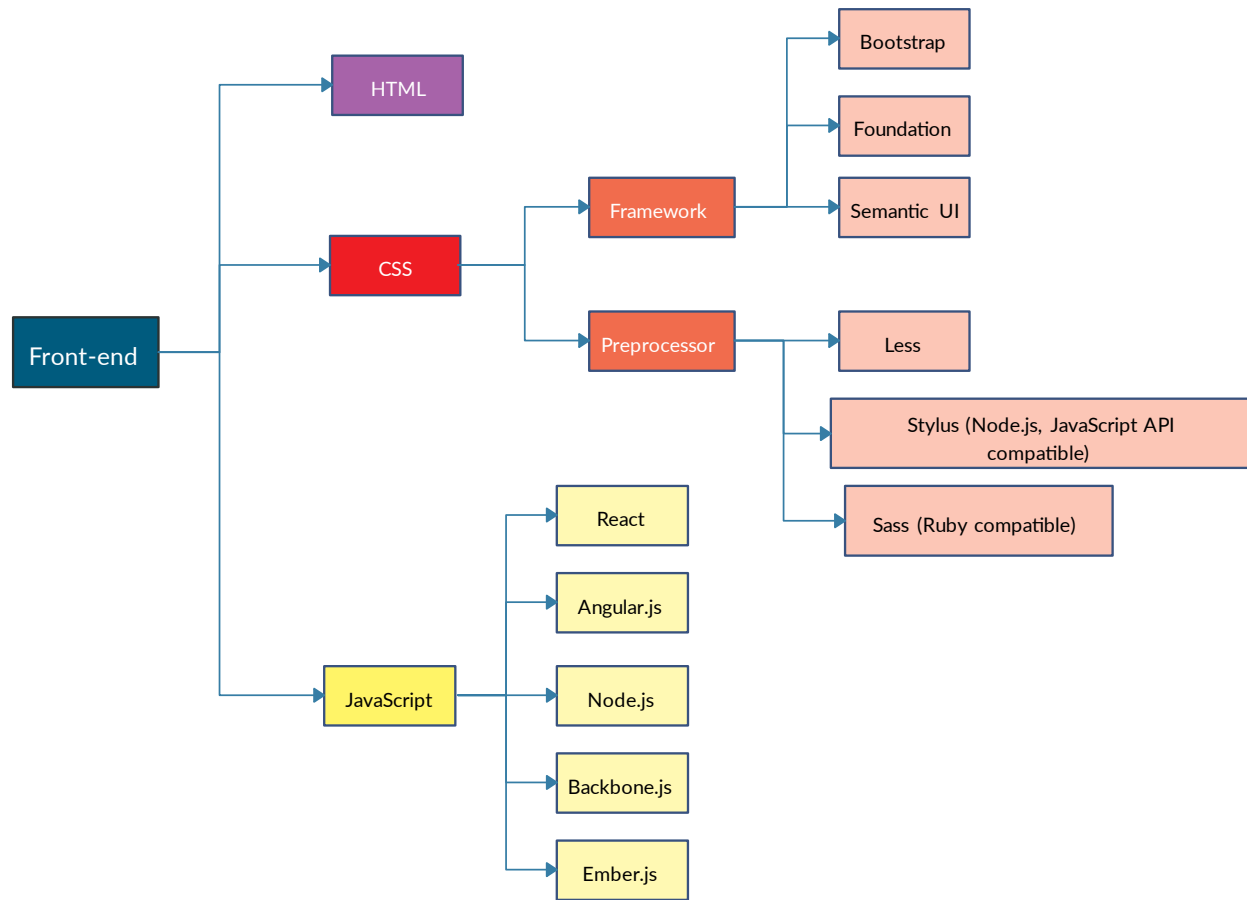


Figure 4.1: Front-end development [classic] (Khametov, 2022).

4.3.2 Decision

As you can see there are a lot of options to choose from and all are fine frameworks and mark-up languages, and the choice usually comes down to familiarity and preference. The tech stack I decided to use in Time Flies is React (Next.js) for the JavaScript interactivity which uses a JSX

component-based system to render HTML5. Next.js utilizes server-side rendering (SSR) to render complicated JavaScript-heavy webpages quickly by building the webpage server side and packaging all the HTML, CSS, and JavaScript into a single HTML static page to be sent to the client's browser. Once the page is rendered by the client Next.js uses a method called "hydration" which enables the react components to become interactive again. This allows the page's initial load times to be very fast and after the page is rendered the complex JavaScript code "comes to life". This makes for a very pleasant user experience.

For the styling of the HTML, I decided to use Tailwind CSS. Normally with native CSS, you must create a separate style.css file that contains all the different styles you wish to use in your HTML. With Tailwind, you can simply add the styling directly into the HTML elements. To accomplish this Tailwind has "classes" that are written in the opening brackets of the HTML. (ex. `<div className="flex items-center justify-between mb-8 md:mb-12 gap-8">`). While one look at this code can make even experienced developers a little bit overwhelmed, once you get used to using Tailwind class-based system, it will increase front-end productivity greatly (Rook, 2023). This makes creating and styling HTML very fast and offers all the styles experienced front-end developers expect in their tool kits.

The final framework I decided to use in Time Flies is D3.js. D3 (Data-Driven Documents) is a JavaScript library primarily used to dynamically visualize data within web browsers. I utilized D3 to offer robust visuals for the results of the AI model. The final product is a very smooth and visually pleasing set of graphs that the user can interact with. To conclude, I chose to use Next.js, Tailwind CSS, and D3.js as the front-end tech stack in this web application.

4.4 Implementation

4.4.1 Figma Design

Figma is a cloud-based design tool that is used to scaffold the look and feel of a web application. It allows designers and developers to create a prototype of the web application that can then be turned into HTML and CSS. The first step I took when creating the Figma design for Time Flies was to set up all the frames I would need for each component and label them accordingly. Next, for each of the frames I set up a grid and the margins/sections for the page. This acts as the skeleton that makes it easier to align the content you add later. Next, I used free stock photo sites to add static assets such as pictures to the project. For non-static items, I used Figma Community's extensive library of free UI elements to import into my design. This includes the buttons, graphs, gradients, accordions, etc. Next, I defined the styles I wanted to use with the project. This includes font, sizes, weights, color palettes, etc. This helps with maintaining consistency throughout the design. Finally, once I had all the design components in place, I was able to start the long process of building out each page until I populated each frame with the desired look (see Fig 4.2).

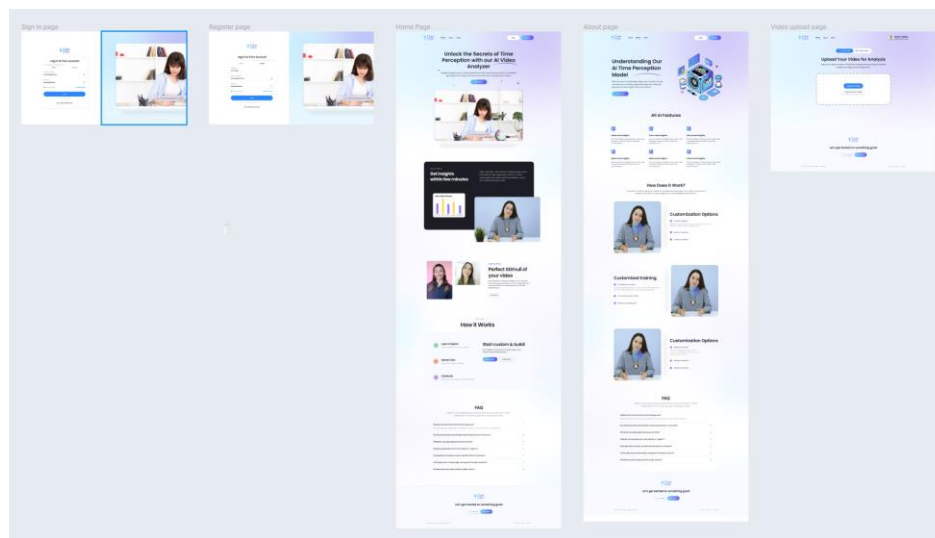


Figure 4.2: Figma design for Time Flies can be found at:

<https://www.figma.com/file/qyCSi9TX00GAF2pZdhVt2g/Time-Files?type=design&node-id=2602%3A1254&mode=design&t=8v49B91EVnagr1Cr-1>.

4.4.2 Next.JS + Tailwind CSS

Starting a Next.js project is very simple. To generate a baseline structure, you can use the “npx create-next-app project-name” command in the root directory of your project. This provides folders and files for me to get started developing my web application. It includes an app directory, node modules directory, public directory, and a multitude of config files to be customized for my specific needs (see Fig 4.3).

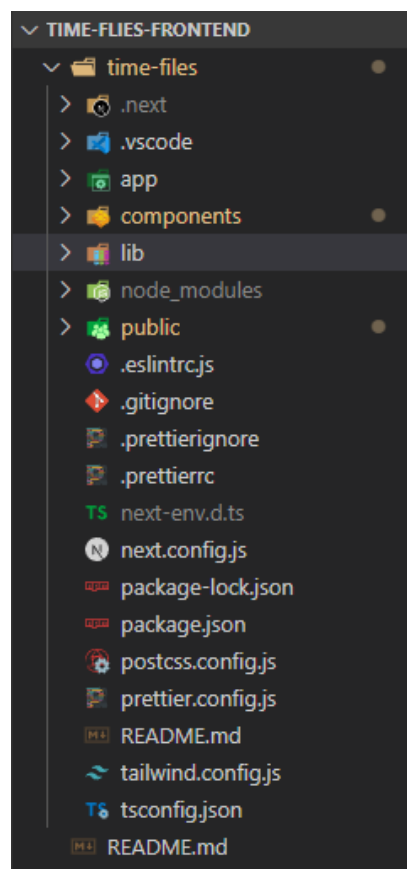


Figure 4.3: The basic structure of a Next.JS app.

Within the app directory, I set up the root pages of the website, the favicon, and the globals.css files (see fig. 4.4). The root page is the entry point to the website where I can define the pass in any components into props.children to maintain consistency across all pages. Instead of writing

the components directly in the root pages I instead created a components directory that contains all the content of the website. The favicon is the little image that gets displayed on the tab of the webpage in the browser. The globals.css file contains all the styles I use throughout the web application.

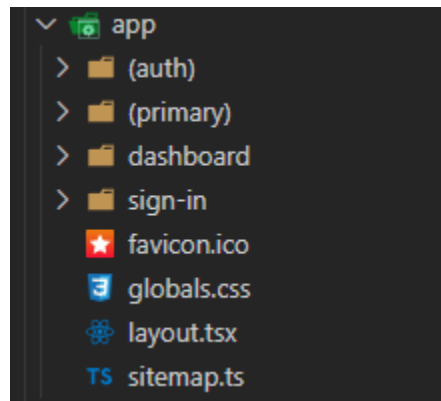


Figure 4.4: The app directory of the Time Flies front-end.

After installing Tailwind CSS, to generate the Tailwind and PostCSS config files in my project I used the “`npx tailwindcss init -p`” command which allows me to customize some of the global styles Tailwind will use in my application. To tell next.js to use tailwinds styles from the config I have to add a few decorators to the globals.css file in the app directory (see fig. 4.5). After that I can directly import the globals.css file directly into my root page layout with “`import './globals.css'`”. This will extend to every single component that gets passed into the root layout, effectively allowing tailwind classes to be used in my entire project.

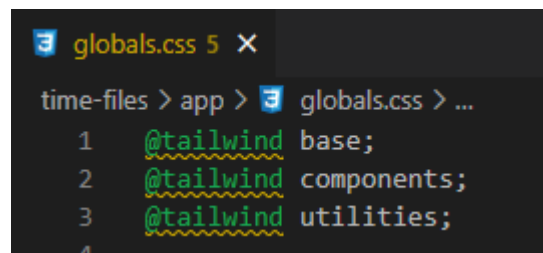


Figure 4.5: Tailwind CSS decorators in globals.css for importing the style parameters from the tailwind config file.

After all the boilerplate and styles with Tailwind were set up, I could finally start converting the Figma design into components that are used within my root page layouts. This required a lot of meticulous trial and error to get a 1:1 conversion. Thankfully, Figma has a lot of built-in tools to help with the process. All the style information, assets, and alignment parameters are easily assessable for each element of the design. I first started by importing all the assets into my component directory. Next, I took all the color palettes, fonts, sizes, and other style information and added them to my Tailwind.CSS config file. For each frame in the Figma design, I created a .JSX file in the component directory with the names accordingly. Within each .JSX file I put the basic boilerplate function a React project expects which returns some HTML. The work of the conversion takes place entirely in this returned HTML. Finally, I can add divs, sections, headings, p-tags, etc. to scaffold out that page's content. To implement the styling, I used Tailwind's className paradigm and with much trial and error was able to replicate the original design. To add interactivity to these static HTML returns, I implemented some React-specific JavaScript code in the body of the function such as navigation, switching tabs, opening and closing accordions, etc. (see fig 4.6).

```
// Component function
export default function FAQ() {
  // JavaScript logic goes here
  return (
    // Returns the HTML
    // Style the component using Tailwind CSS className parameters
    <section id="faq" className="mb-14 w-full pt-14 md:mb-36 md:pt-24">
      <div className="container">
        <div className="mx-auto w-full max-w-[1064px]">
          <div className="mx-auto w-full max-w-[790px] text-center">
            <h2 className="text-4xl font-bold leading-[56px] text-slate-800 md:text-5xl">FAQ</h2>
            <p className="mt-4 text-base font-normal leading-6 text-[#808D9E] md:text-xl md:leading" />
          </div>
          /* <!-----faq container-----> */
          <div className="mt-12 w-full">
            {faqs.map((item, index) => (
              <FAQItem key={index} {...item} />
            ))}
          </div>
        </div>
      </div>
    </section>
  );
}
```

Figure 4.6: The function for the FAQ component.

4.4.3 Back-End Integration

After I created all the static components and applied them to the app's root page layouts, I needed to implement the fetching, writing, and deleting of data to the back end. This includes user authentication, user information management, sending videos to be analyzed, and receiving the results and data of the analyzed video. This is achieved by sending requests to the back-end's API HTTP routes. React has many methods of sending, receiving, and manipulating requests and responses from an API. A common way React developers do this is with the use of "await fetch" declarations (Pavlutin, 2023) to send requests and receive responses (see Fig. 4.7). To use the response to "effect" the application a useState declaration with a value and function is initialized at the start of the component function (see fig. 4.6). This allows the data from the response to dynamically change the value of the useState declaration by calling its function and updating the value within the response section of a fetch request. For example, if I had a useState declaration that checks if a user is logged in (see Fig. 4.6), I could send a fetch GET request to the "users/is_logged_in" API route (see Fig. 4.7). This route returns a True or False if the current user is logged in or not. I then use the response data to dynamically update the isLoggedIn value by using the setIsLoggedIn function and if the user is not logged in redirect them to the sign-in page. It allows for fine control flow of your web pages and can be implemented for anything from setting the user's profile picture to the header to streaming video packets from the backend. Similar methods can be used to send or update data to the back-end database as well. useState and await fetch are very powerful tools React provides to its developers.

```
export default function VideoUploadRecent() {  
  const [tab, setTab] = useState("upload");  
  const [videos, setVideos] = useState<Video[]>([]);  
  const [isLoading, setIsLoading] = useState(false);  
  const [error, setError] = useState('');  
  const [isLoggedIn, setIsLoggedIn] = useState(false);  
}
```

Figure 4.6: Shows the initialization of useState [variables, functions] to dynamically change values at the start of a component function.

```

useEffect(() => {
  // Fetch login status when component mounts
  const fetchLoginStatus = async () => {
    try {
      const response = await fetch('http://127.0.0.1:5000/users/is_logged_in', {
        method: 'GET',
        credentials: 'include',
      });
      const data = await response.json();
      setIsLoggedIn(data.isLoggedIn); // Update login status based on response
      if (!isLoggedIn) {
        router.push('/sign-in?type=login'); // Redirect to login page if not logged in
      }
    } catch (error) {
      console.error('Error fetching login status:', error);
      router.push('/sign-in?type=login'); // Redirect to login page on error
    }
  };

  fetchLoginStatus();
}, [isLoggedIn, router]);

```

Figure 4.7: Shows the use of `await fetch` to send requests to the back-end API and the `useEffect`'s function to dynamically update values based on the response of the request.

4.4.4 Dynamic Graph Visualization

On the results page, I had the idea to visualize the data from the AI model smoothly and interactively. To accomplish this, I utilized the D3.js (Data Driven Documents) library to parse through the graph JSON data created by the AI model to dynamically create graphs for each data point. This allows the user to use their mouse to move through the graph, and the background is the analyzed video which creates a scrubbing effect (see Fig. 4.8). This allows users to see exactly where in the video substantial salient features were classified, when those changes warranted a passing of the dynamic thresholds, and the accumulation of said silent features at each layer.

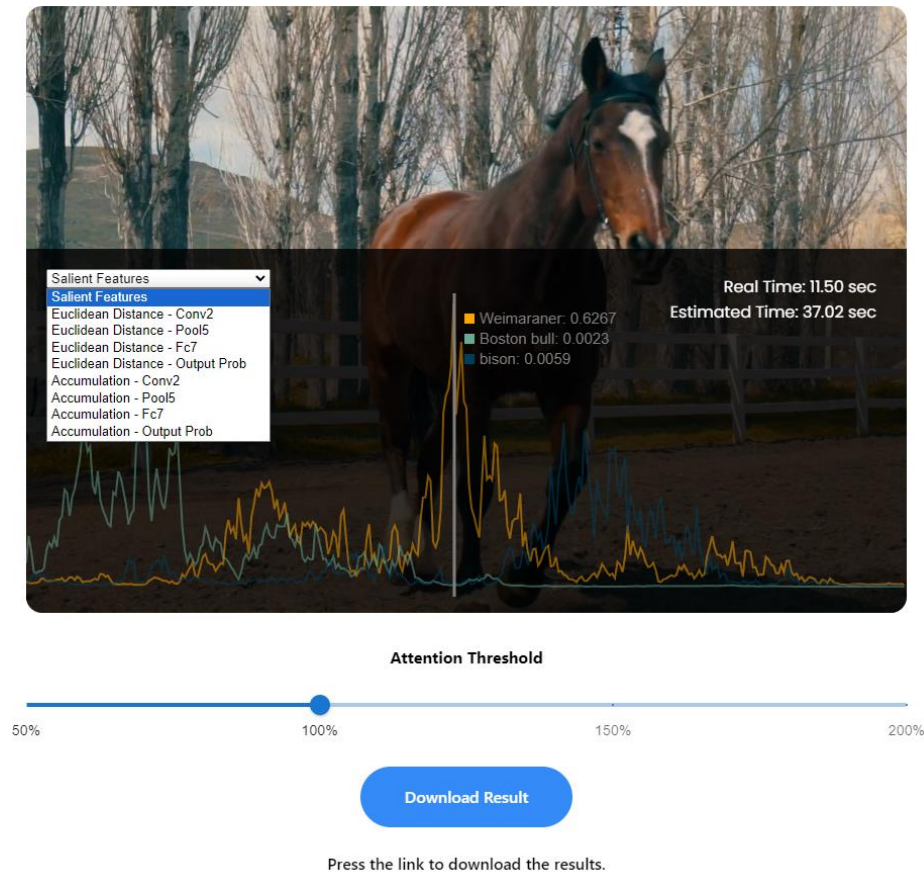


Figure 4.8: Shows the dynamic graph visualization on the results page.

Countless hours were spent attempting to make this experience as smooth and user-friendly as possible. Optimizations such as throttling updates to DOM and rendering the graphs in SVG to make them comply better with the way React updates the DOM. The inclusion of Dynamic legends and a drop-down window separating the different graphs shows a keen interest in ease of use. The attention slider directly affects the graphs and time estimation. When a video is analyzed by the back-end it stores four copies of the results and JSON data, one at each attention threshold. As the slider moves the parent component makes a fetch request to the corresponding results.png and graph JSON data. A download button is implemented to get the PNG of the results in its entirety.

5. Deployment

5.1 Front-End Deployment

Deploying a Next.js web application through Vercel is a straightforward process. Vercel owns and maintains the Next.JS framework and offers seamless integration for deploying its applications. First, you must make sure you have a `package.json` configuration file that contains all your project's dependencies and a `next.config.js` configuration file that contains the deployment instructions for your project. Next, you create a GitHub repository with the files for your web application. Then, you have to create a free Vercel account and once logged in you can import your GitHub repository and begin the build process. Within minutes you'll have a fully deployed website (see Fig. 5.1) with a default domain name that can be updated, and you can purchase more specific domains directly through Vercel. Anytime you push an update to your repository Vercel will automatically re-deploy the latest build. Vercel's free tier offers 100 GB hours of serverless function execution, 100 build minutes, and 100 GB of bandwidth per month that can be scaled and upgraded anytime if traffic increases. This is perfect for student and hobbyist projects like the application I've built. Time Flies is hosted at: <https://time-flies-frontend.vercel.app/>.

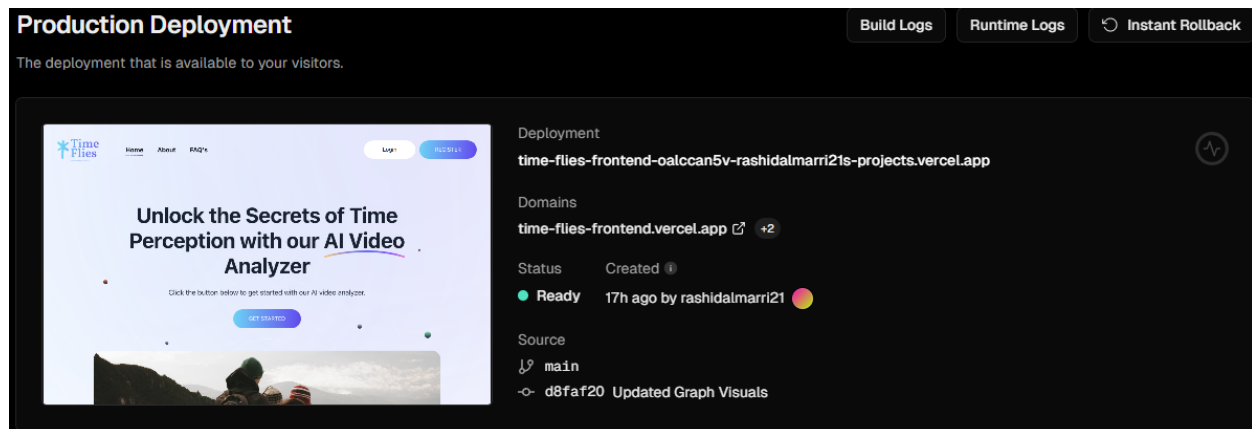
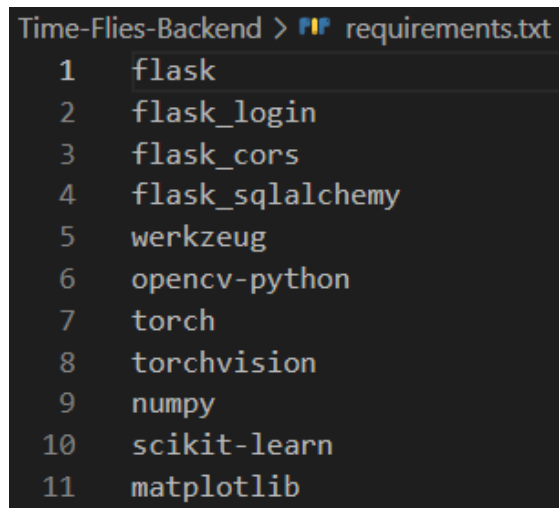


Figure 5.1: The production build for Time Files through Vercel.

5.2 Back-End Deployment

In contrast to the front-end, back-end deployments can be a little bit trickier. Since my application hosts a compute-intensive AI model it's not economical nor practical to host it through a cloud service as it would cost a lot of money to utilize. That being said, I will still include the full process of deploying my back-end through AWS (Amazon Web Services). First, you should create a requirements.txt file within the root directory of your application. This contains all the dependencies required for your application to run (see Fig. 5.2).



```
Time-Flies-Backend > requirements.txt
1 flask
2 flask_login
3 flask_cors
4 flask_sqlalchemy
5 werkzeug
6 opencv-python
7 torch
8 torchvision
9 numpy
10 scikit-learn
11 matplotlib
```

Figure 5.2: Requirements text file for Time Flies Backend.

Next, you should ensure you have an entry point Python script that will be run on initialization. In my case, it would be the main.py file in my root directory. It is good practice to include a Procfile that contains the exact script to be run. Then, you need to set up a WSGI server to handle requests as the development server Flask uses is not secure enough to be used in production. A popular choice among most Python developers is to implement Green Unicorn as your WSGI server. With that, your Python back-end is ready for a production environment through a cloud service. While AWS has many ways to deploy an application of this nature, I will be focusing on using its Elastic Beanstalk (EB) service. First, you need to install EB's CLI interface. Once installed, you initialize an EB instance using this command: "eb init -p python-python-

version my-flask-app --region my-aws-region” and then use “eb create” and “eb deploy” commands to finish deploying your application. Through the AWS website, you can also set up your database with their RDS (Relational Database Service) and continuous deployment through CI/CD pipelines such as AWS CodePipeline and AWS CodeBuild. While this is an easy process to implement, for the scope of my project I don’t find it practical to deploy my backend as you can still use it locally and make requests from the front-end as long as the back-end is running on the same computer. My back-end repository can be found at:

<https://github.com/rashidalmarri21/Time-Flies-Backend-Final>.

6. Discussion

6.1 Results and Limitations

6.1.1 AI Model

Overall, my implementation of the Time Storm – Time Without Clocks AI model is robust and efficient. Updating the deprecated Caffe implementation of AlexNet with PyTorch decreased end-to-end processing times by up to 39x (see Appendix A). The updated TS model is more suitable for a web-based environment which was the main goal of the conversion. Extending the scope to include the global attention variable, C , at four thresholds gives a better understanding of the role attention plays in temporal time estimation. That being said, there may be a few limitations to my implementation. The current model does give pretty accurate results even on new data, but it’s limited by the videos used in the regression training. If a user uploads a video that doesn’t fit the five or so video scene types it could lead to inaccurate results and a disappointing user experience. With the use of either PyTorch-CPU or PyTorch-CUDA, the model requires a lot of computing power to operate which could be bottlenecked if even tens of users all try processing videos at the same time. This would require an expensive scaling infrastructure through a cloud service that could cost thousands a month if the web application gets stressed by traffic. Additionally, the amount of “hot”, expensive storage needed to operate the AI model could also add more overhead costs in a production environment at scale.

6.1.2 Back-End

Overall, the Flask back-end archives its role in a lightweight, scalable, secure server to host our database, API, and AI model. WSGI servers like Green Unicorn to handle concurrent requests and the infrastructure to scale horizontally with cloud services such as Nginx and AWS ELB (Elastic Load Balancing) to manage high-traffic sites. While all of this is possible with the Flask implementation, it requires a lot more setup compared to competitors such as Django and can add a lot of overhead costs, especially on a free-to-use service like Time Flies.

6.1.3 Front-End

Overall, the Next.JS front-end implementation fulfills its role as a platform to interact with the AI Model. With a robust user experience and visually pleasing interface, it ticks all the marks of a modern front-end website. Ease of use was a high priority in its development, making it as easy as possible to get started analyzing videos with the AI model and visualizing the results. That being said, the choice to use Next.JS means I am heavily locked into using Vercel's infrastructure. Vendor lock-in can lead to headaches down the road if you need to switch to a different provider and Vercel is known to offer a generous free tier but as soon as your application needs to scale, the prices become unmanageable. To switch to a new cloud service would require extensive refactoring and reconfiguration of the codebase.

6.2 Future Improvements

If given more time to work on Time Flies, I would have loved to implement an AI Model Training Suite that would have let users upload their own videos to use in the training of the regression model. This could include automatically splitting videos into trials, similar to how the original study was conducted. This could lead to a more robust and accurate model that fits the users' needs better. To make something like this viable, extensive optimization to the entirety of the AI model needs to be done, not just the AlexNet implementation. While yes, I did succeed in decreasing the processing time of training the regression model from ~10 days down to ~40

hours, this would be unacceptable in a production environment. You can't have users waiting that long for results.

Another possible improvement would have been creating an API for the AI model so other developers and websites could utilize the AI model for their projects without having to manually set up everything and create a fork of the GitHub repository. This would protect the intellectual property of Dr. Warrick's and Time Storm's AI Model but also allow anyone to use and contribute to the work they started.

Conclusion

In conclusion, Time Flies accomplished all the goals I set at the beginning of the project. From updating the AI model to be more modern and efficient, to creating a robust and user-friendly front-end that researchers can use to interact with the AI model and understand temporal time perception better, to finally, creating a highly scalable back-end with a restful API to manage data and host the AI model. This project let me show off my full-stack development skills in a way that can be useful to the field of AI and neuroscience. I have also included a standalone repository with the updated Time Storm model equipped with a demo in the Abstract section of this paper for future students and AI researchers to use and extend.

References

- Awad, M. and Khanna , R. (2015) 'Support Vector Regression', in *Efficient Learning Machines*. Berkeley, CA: Apress, pp. 67–80. Available at: https://link.springer.com/chapter/10.1007/978-1-4302-5990-9_4#citeas (Accessed: 25 April 2024).
- Balaj, Y. (2017) (PDF) *token-based VS session-based authentication: A survey*, *researchgate.net*. Available at: https://www.researchgate.net/publication/320068250_Token-Based_vs_Session-Based_Authentication_A_survey (Accessed: 04 April 2024).

- Desmond, K. (2023) *Most Popular Backend Frameworks for 2023*, Coding Nomads. Available at: <https://codingnomads.com/blog/best-backend-frameworks> (Accessed: 04 April 2024).
- Educative.io Team (2020) *What is flask? - flask: Develop web applications in Python*, Educative. Available at: <https://www.educative.io/courses/flask-develop-web-applications-in-python/what-is-flask#Origins-of-Flask> (Accessed: 05 April 2024).
- Gallinelli, N. (2024) *Front end vs. back end development: What is the difference?*, Flatiron School. Available at: <https://flatironschool.com/blog/front-end-vs-back-end-development/> (Accessed: 15 April 2024).
- Ghimire, D. (2020) *Comparative study on python web frameworks: FLASK and ..., theseus.fi*. Available at: https://www.theseus.fi/bitstream/handle/10024/339796/Ghimire_Devndra.pdf?sequence=2&isAllowed=y (Accessed: 04 April 2024).
- Griffith, E. and Metz, C. (2023) *A new area of A.I. booms, even amid the tech gloom*, The New York Times. Available at: <https://www.nytimes.com/2023/01/07/technology/generative-ai-chatgpt-investments.html> (Accessed: 25 April 2024).
- Holovaty, A. (2009) *The django book, django-book-new.readthedocs.io*. Available at: <https://django-book-new.readthedocs.io/en/latest/frontmatter.html> (Accessed: 04 April 2024).
- Hummen, R. et al. (2013) 'Towards viable certificate-based authentication for the internet of things', *Proceedings of the 2nd ACM workshop on Hot topics on wireless network security and privacy* [Preprint]. doi:10.1145/2463183.2463193.
- Hygraph Team (2022) *How API Works*, Hygraph. Available at: <https://hygraph.com/blog/how-do-apis-work> (Accessed: 04 April 2024).
- Khametov, I. (2022) *Front-end development [classic]*, creately.com. Available at: <https://creately.com/diagram/example/inys15sq1/front-end-development-classic> (Accessed: 15 April 2024).
- Muittari, J. (2020) *Modern web back-end - theseus, theseus.fi*. Available at: https://www.theseus.fi/bitstream/handle/10024/336520/Joel_Muittari.pdf (Accessed: 04 April 2024).
- Pavlutin, D. (2023) *How to use fetch with Async/await*, Dmitri Pavlutin Blog. Available at: <https://dmitripavlutin.com/javascript-fetch-async-await/> (Accessed: 25 April 2024).
- Reschke, J. (2015) *The 'BASIC' HTTP authentication scheme* [Preprint]. doi:10.17487/rfc7617.
- Rook, R. (2023) *Tailwind for Productivity*, iO tech_hub. Available at: <https://techhub.iodigital.com/articles/tailwind-for-productivity> (Accessed: 25 April 2024).

Roseboom, W. *et al.* (2019) ‘Activity in perceptual classification networks as a basis for human subjective time perception’, *Nature Communications*, 10(1). doi:10.1038/s41467-018-08194-7.

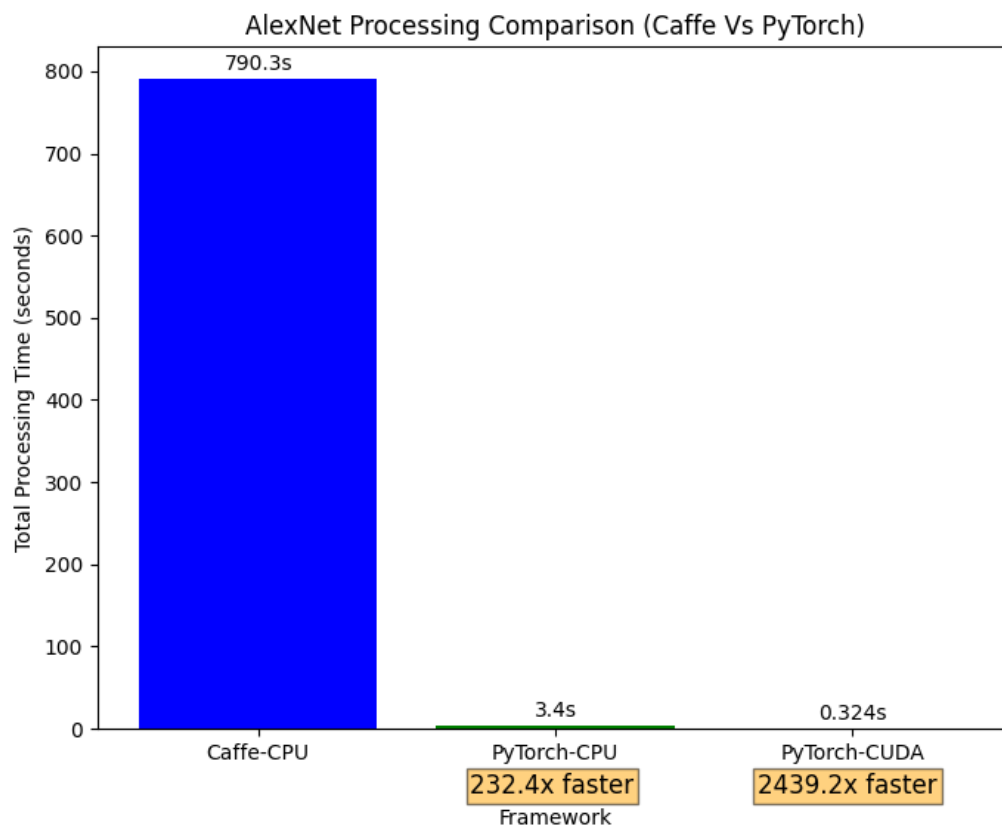
Wong, T.-T. and Yeh, P.-Y. (2020) ‘Reliable accuracy estimates from k-fold cross validation’, *IEEE Transactions on Knowledge and Data Engineering*, 32(8), pp. 1586–1594. doi:10.1109/tkde.2019.2912815.

Appendices

Appendix A: Speed Test Specifications

In **2.3.4 Speed Comparisons**, two tests were conducted to measure the performance of Caffe-CPU, PyTorch-CUP, and PyTorch-CUDA. This appendix will cover how those tests were conducted in greater detail.

Test one:



This test was conducted in a separate environment from that of the Time Storm model. Two separate classes, one for Caffe and one for PyTorch, were created to test their respective performances. The same 11-second video was used in both implementations and the results only include the time each frame took to process and not the initialization or conversion to frames. Each class does its respective transformations of the image before passing it through the AI model and only during the actual processing of the frame does Python's built-in Time library append that frame's processing time to the overall time. For an in-depth look at each class (see Fig. 1.1 and 1.2)


```

class AlexNet:
    def __init__(self, directory='alexnet', output_file='/app/output/total_processing_time.txt'):
        self.MODEL_DEF = os.path.join(directory, 'deploy.prototxt')
        self.MODEL_WEIGHTS = os.path.join(directory, 'bvlc_alexnet.caffemodel')

        # Check if model weights are present, if not download them
        if not os.path.isfile(self.MODEL_WEIGHTS):
            print('Downloading AlexNet weights...')
            urllib.request.urlretrieve("http://dl.caffe.berkeleyvision.org/bvlc_alexnet.caffemodel", self.MODEL_WEIGHTS)
            print('Download complete.')

        # Load the model in CPU mode
        caffe.set_mode_cpu()
        self.net = caffe.Classifier(self.MODEL_DEF, self.MODEL_WEIGHTS)

        # Setup transformer
        self.transformer = caffe.io.Transformer({'data': self.net.blobs['data'].data.shape})
        self.transformer.set_transpose('data', (2,0,1)) # Move image channels to outermost dimension
        self.transformer.set_raw_scale('data', 255) # caffe.io.load_image uses scale of [0, 1]
        self.transformer.set_channel_swap('data', (2,1,0)) # Swap channels from RGB to BGR
        print("AlexNet loaded successfully.")

        self.total_time = 0.0 # Initialize total processing time
        self.output_file = output_file

    def preprocess_and_classify(self, frame):
        # Preprocess the frame
        transformed_image = self.transformer.preprocess('data', frame)

        # Perform classification
        self.net.blobs['data'].data[...] = transformed_image
        start_time = time.time()
        output = self.net.forward()
        classification_time = time.time() - start_time

        return classification_time # Return the time taken for classification

    def run_test(self, video_path):
        if not os.path.exists(video_path):
            print(f"Video path {video_path} does not exist.")
            return

        cap = cv2.VideoCapture(video_path)
        if not cap.isOpened():
            print("Error opening video file.")
            return

        frame_counter = 0
        frame_count = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
        while True:
            ret, frame = cap.read()
            if not ret:
                break
            frame_counter += 1
            print(f"Processing frame... {frame_counter}/{frame_count}")
            frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB) # Convert BGR to RGB
            time_taken = self.preprocess_and_classify(frame_rgb)
            self.total_time += time_taken # Accumulate total processing time

        cap.release()
        # Save total processing time to a file
        self.save_total_time_to_file()

    def save_total_time_to_file(self):
        with open(self.output_file, 'w') as f:
            f.write(f"Total Processing Time: {self.total_time} seconds\n")
            print(f"Total processing time saved to {self.output_file}.")

```

Figure 1.1: Caffe testing class.

```

class AlexNetPyTorch:
    def __init__(self, video_path, output_file='output/PyTorch_times.txt'):
        # Load the pre-trained AlexNet model
        self.model = models.alexnet(pretrained=True)

        # Set the model to evaluation mode and use CUDA if available
        self.model.eval()
        self.device = torch.device("cpu")
        self.model.to(self.device)

        # Set up the image transforms
        self.transform = transforms.Compose([
            transforms.Resize(256),
            transforms.CenterCrop(224),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
        ])
        print("AlexNet loaded successfully on {}".format(self.device))

        # Video processing setup
        self.video_path = video_path
        self.total_time = 0
        self.output_file = output_file

    def preprocess_and_classify(self, frame):
        # Convert frame to PIL Image to be compatible with torchvision transforms
        frame = Image.fromarray(frame)

        # Apply the preprocessing transforms
        input_tensor = self.transform(frame)
        input_batch = input_tensor.unsqueeze(0) # Create a mini-batch as expected by the model

        # Move the input and model to GPU if available
        input_batch = input_batch.to(self.device)

        # Measure the inference time
        start_time = time.time()
        with torch.no_grad():
            output = self.model(input_batch)
        classification_time = time.time() - start_time

        return classification_time

    def run_test(self):
        # Open video
        cap = cv2.VideoCapture(self.video_path)
        if not cap.isOpened():
            print("Error opening video file.")
            return

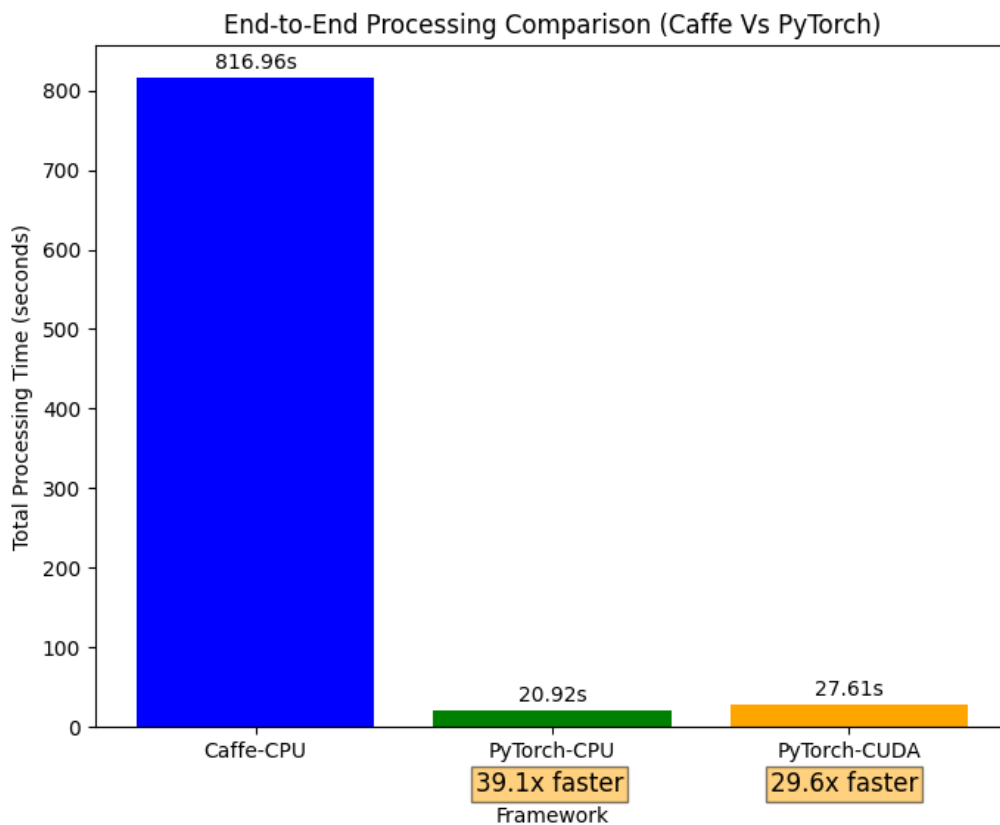
        while True:
            ret, frame = cap.read()
            if not ret:
                break
            # Process frame and store time
            frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB) # Convert BGR to RGB
            time_taken = self.preprocess_and_classify(frame)
            self.total_time += time_taken
        cap.release()
        # Save times to a file
        self.save_total_time_to_file()

    def save_total_time_to_file(self):
        with open(self.output_file, 'w') as f:
            f.write(f"Total Processing Time: {self.total_time} seconds\n")
        print(f"Total processing time saved to {self.output_file}.")

```

Figure 1.2: PyTorch testing class.

Test two:



This test was conducted in the same environment as the production code of Time Storm. The only difference between the Caffe and PyTorch tests is the use of the Original AlexNet class and the updated AlexNetNew class, respectively. A separate Python script (testing.py) was created to facilitate the test. The same 11-second video from the first test was used. This test simulated a real-world example of how the frameworks process the video through the entirety of the Time Storm model. For an in-depth look at the testing.py script (see fig. 1.3).

```

import time
from analyze_video import analyze_video
import cv2, os

def convert_video_to_frames(video_path, save_dir):
    # Make sure the save directory exists
    if not os.path.exists(save_dir):
        os.makedirs(save_dir)

    # Open the video
    cap = cv2.VideoCapture(video_path)

    # Determine video FPS
    original_fps = cap.get(cv2.CAP_PROP_FPS)
    frame_count = 0

    while True:
        # Read a frame
        success, frame = cap.read()

        # If the frame was not successfully read, break the loop
        if not success:
            break

        # Save every frame without trying to simulate 30 FPS
        save_path = os.path.join(save_dir, f'frame_{frame_count:04d}.png')
        cv2.imwrite(save_path, frame)
        frame_count += 1

    # Release the video capture object
    cap.release()

    print(f'All frames have been saved to {save_dir}. Total frames: {frame_count}.')
    return original_fps

def test_analyze_video(video_path):
    save_dir = 'test_frames'
    convert_video_to_frames(video_path, save_dir)

    start_time = time.time() # Start timing
    figure, json = analyze_video(save_dir)
    elapsed_time = time.time() - start_time # End timing

    print(f"analyze_video function took {elapsed_time:.2f} seconds to execute.")
    return

test_analyze_video('test_video/test.mp4')

```

Figure 1.3: Testing.py script for end-to-end processing test.