Per-Object Motion Blur for Live Action

Team Carnotaurus

Henry Huang, Ivan Guo, Nancy Makar, William Gong

Table of Contents:

Project Description:

Since this is a video based project, it is very hard to convey the idea and concepts via pictures and text. Hence, if any of the following is unclear, then please see our video for proper visual examples.

As videos are just rapid streams of pictures shown one after the other, “smoothness” has long been desirable. Traditional methods include recording at higher frame rates and motion blur. Higher frame rates require foresight, and cannot be appended. Thus, if someone wants a smoother video, they have to rerecord it, which is not always feasible.

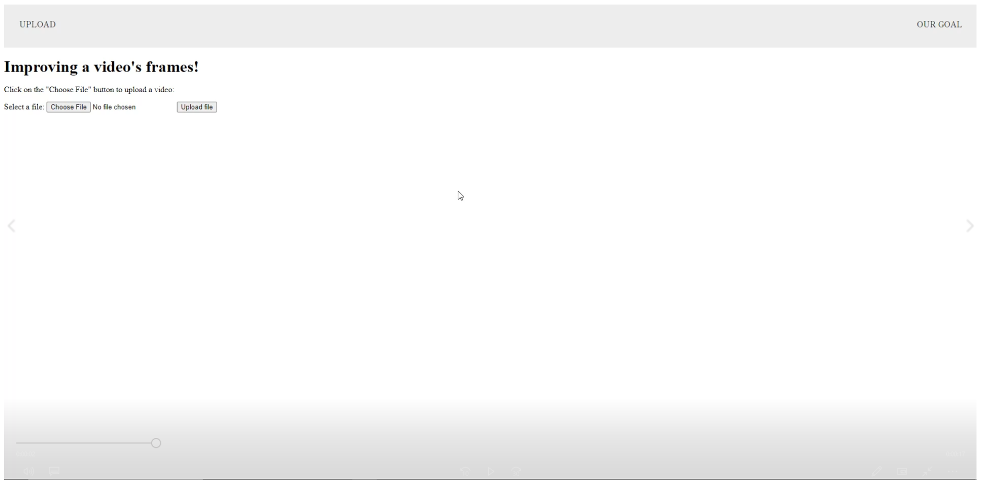
Regular motion blur is cheaper but affects entire frames in a video. In ideal conditions, like with a stable camera, this achieves the desired effect of smoothness. However, if the camera moves, then everything in the frame is moving relative to the camera. This causes background objects to be blurred, which can harm visual clarity.

This project seeks to implement an improved version of motion blur, known as per-object motion blur. This method only blurs objects that are moving irrespective of the camera’s movement. For example, if a camera turns while pointing at a person waving, regular motion blur would blur the entirety of the frames for when the camera is turning. Per-object motion blur would instead only blur the person’s arm waving. Consequently, none of the details of the unmoving person or background would be lost, resulting in more smooth motions while preserving visual clarity.

Per-object motion blur is usually applied to animation, where it is easy to identify beforehand which objects are moving. This project seeks to implement it in the context of live action recordings, where moving objects cannot be determined beforehand, and would have to be manually animated to achieve the effect. Manually going through every frame to apply motion blur is exceptionally tedious, so we aim to automate the process using an object tracking machine learning model.

Walkthrough Example:

1. Begin with choosing a video file that you would like to improve

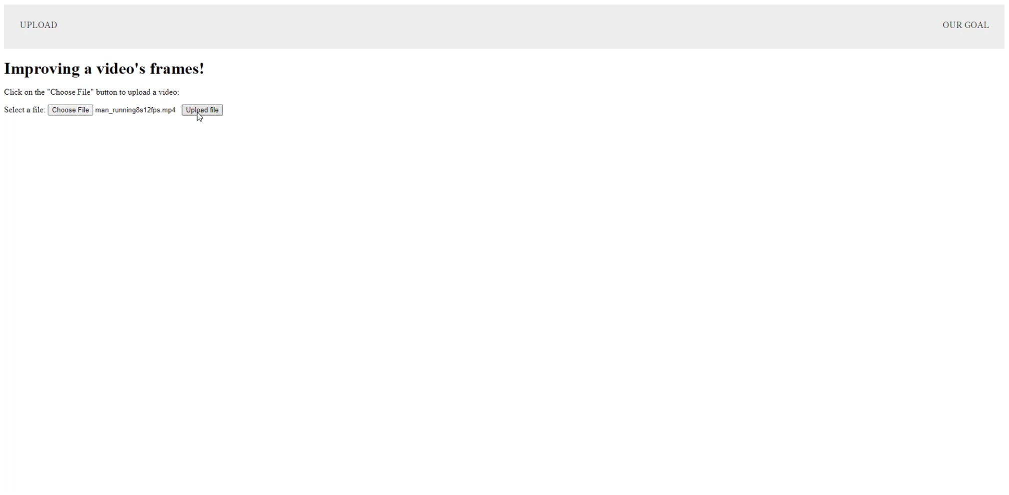


Improving a video's frames!

Click on the "Choose File" button to upload a video:

Select a file: Choose File No file chosen Upload file

2. After choosing the file, select upload file



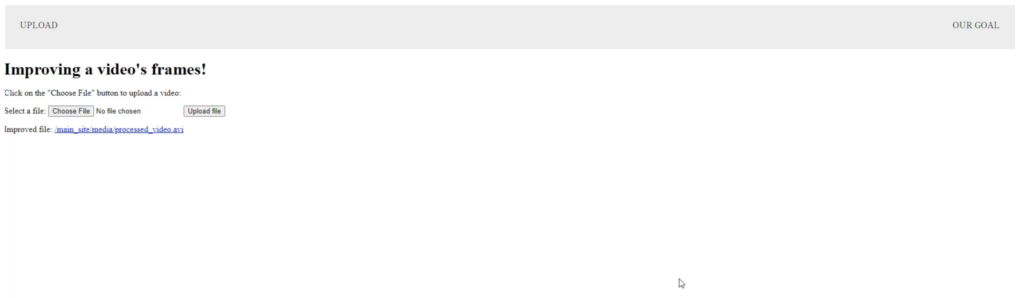
UPLOAD OUR GOAL

Improving a video's frames!

Click on the "Choose File" button to upload a video:

Select a file: Choose File man_running8s12fps.mp4 Upload file

3. After waiting a short period, click the blue link for your processed video



UPLOAD OUR GOAL

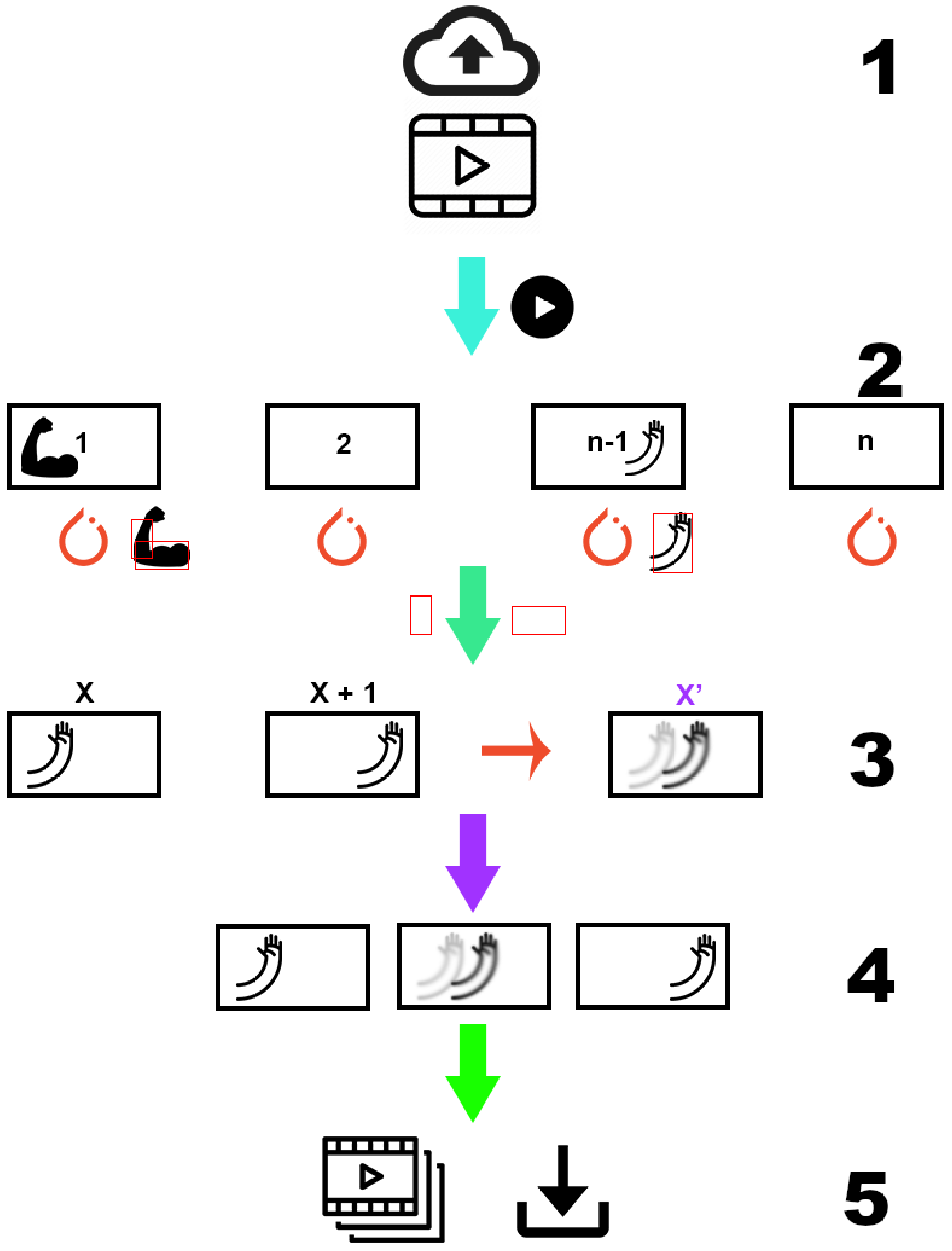
Improving a video's frames!

Click on the "Choose File" button to upload a video:

Select a file: Choose File No file chosen Upload file

Improved file: /main_site/media/processed_video.avi

Diagram (Legend on next page):



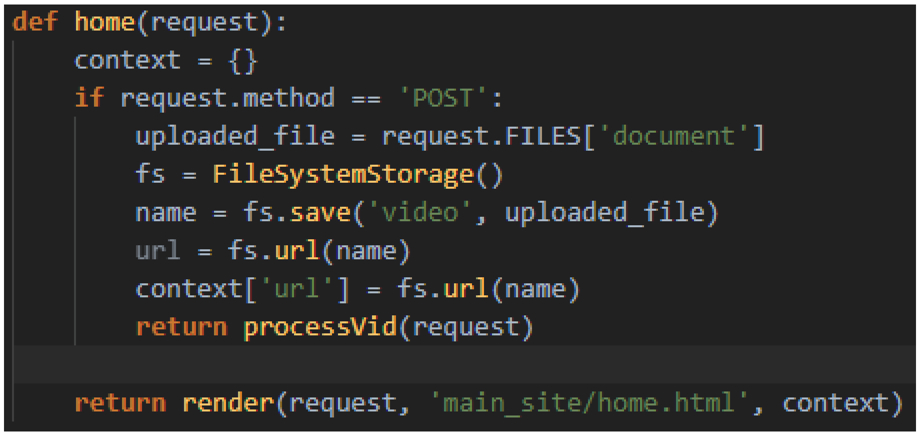
Component flow:

1. User uploads a video to the webpage.
2. An object tracking model is run on each individual frame of the video, giving bounding boxes of relevant, moving objects.
3. For every two frames, when the same object is detected between both frames, a new blurred frame is created.
4. For every new frame created in 3, insert it between the two frames it was created from.
5. Compile all the frames into a single video for playback and download.

Component Description:

Component 1: Frontend Video Upload

The first component of this application is the webpage. On it, a user can upload any video file to have the per-object motion blur effect applied. This video is passed as a POST request, stored in the FILES field, wrapped by a Django file class object, and saved to the media folder. Once the video is saved, the home function sends the same request to the processVid function where the path of the video is passed into the machine learning component.



```
def home(request):
    context = {}
    if request.method == 'POST':
        uploaded_file = request.FILES['document']
        fs = FileSystemStorage()
        name = fs.save('video', uploaded_file)
        url = fs.url(name)
        context['url'] = fs.url(name)
        return processVid(request)

    return render(request, 'main_site/home.html', context)
```

Component 2: Machine Learning

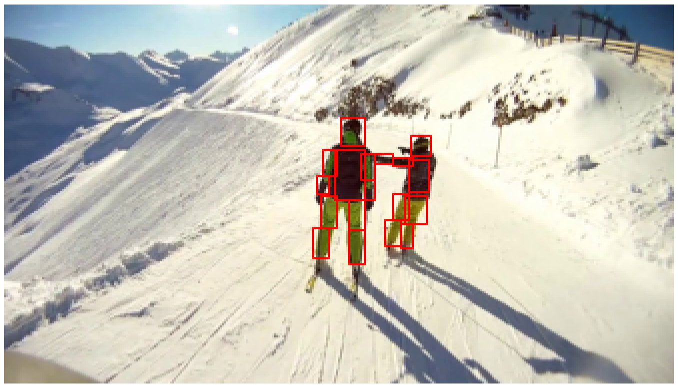
After receiving a video from the webpage's frontend, the next component detects and returns relevant objects in motion. The first step that this component does is run a Pytorch, Faster RCNN (Region-Based Convolutional Neural Network) model on each frame of the video, where each frame is obtained using OpenCV. To train the model (which is done in Model.ipynb), we used two different datasets.

The primary dataset is the MPII Human Pose Database. MPII features a wide array of images designed for training pose prediction AIs, but for our purposes, we used it as a limb detection dataset. The first task for using MPII, was to convert its labels from a Matlab file type to csv. First, the file's contents were extracted using Scipy and then relevant data was copied and formatted algorithmically. Code to do so is found in "Limb Detection"/"Dataset Filtering.ipynb". Since this project is focused on movement, categories like "sports" were used while categories like "home activities" were dropped. Joints labeled as "not visible" by MPII were also dropped as hidden labels could confuse the model and make it less accurate. The next task was to then convert the joint labels into bounding boxes necessary for object detection.

Figure: *An Example of some of the Annotations for an Image in MPII (Not all Annotations Shown)*

To use MPII for object detection, we used an algorithm to approximate bounding boxes. This algorithm can be found in “Limb Detection”/”Add Bounding Boxes.ipynb”. Essentially, we found that, given two joints, an approximate bounding box would have a width that is at least half the diagonal length of the box. For a forearm that is angled at 45 degrees, this does not do anything, but for a shin lying flat on the ground, this allows the data to be usable (with some additional shifting of the bounding box). Additionally, since this dataset is used for an application which features objects in motion being blurred, we decided that 100% accuracy for our bounding box labels was not necessary. Hence, we were able to create a large and comprehensive dataset for limb detection.

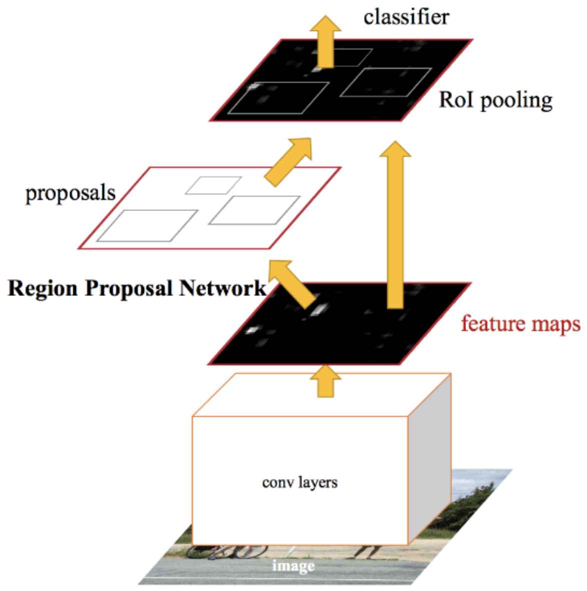
Figure: *The Bounding Box Algorithm Applied to a Sample*



Our second dataset, processed under “Dataset 2 - Tracking”/TrackingDataset.ipynb, features an assortment of different objects. The purpose of this dataset was to add more classes and make our model more versatile. For example, one of the classes is for cars, which are of course often in motion and thus would be viable for our application.

The Faster RCNN model performs object detection for 16 classes that are expected to be in motion. For example, forearms and cars. We chose Faster RCNN because it is specialized to do object detection quickly and effectively. Faster RCNN is based upon the RCNN and Fast RCNN algorithms, and thus operates similarly. The core concept behind RCNN is that it identifies regions of interest, ROI, by grouping similar sections of the image. Using these regions of interest, they can then run classifiers on these regions to achieve object detection. For Faster RCNN specifically, it first uses a region proposal network.

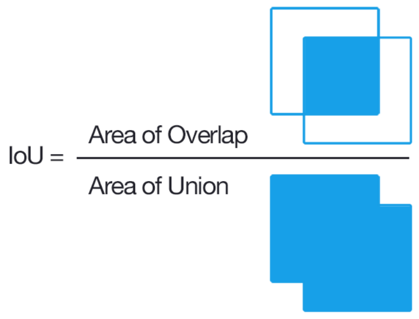
Figure: *Faster RCNN Component Diagram*



This region proposal network essentially preprocesses the image, filtering out regions of interest that are less likely to be relevant. This can significantly increase speed as it avoids processing every possible region of interest in an image.

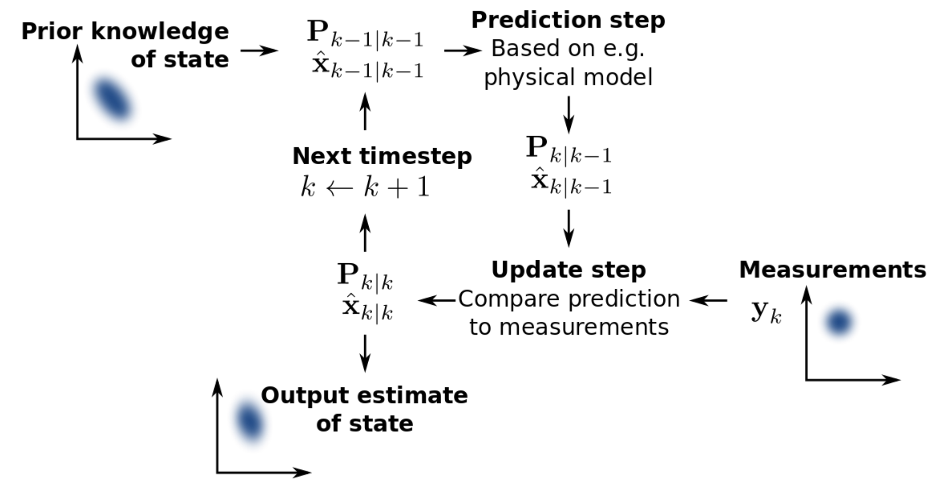
The detections returned by the model, namely bounding boxes and their scores, are passed to the SORT algorithm. The SORT algorithm turns the object detection results into object tracking results by assigning a unique ID to each tracked object throughout the video, while whatever SORT detects as the same object from a previous frame keeps its old ID. We chose SORT as it was designed to be fast and effective for real time, online tracking applications. It works by first taking two matching objects in a frame **X** and **X** + 1. Using the bounding boxes for the two objects, it can apply an intersection over union operation.

Figure: *How Intersection over Union is Defined*



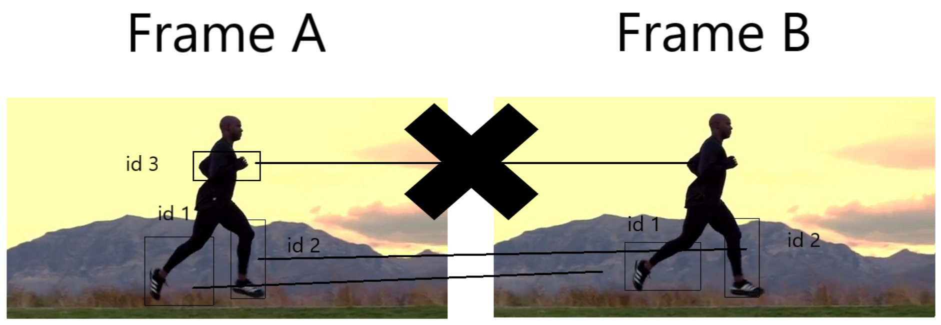
Then, this information is run through a Kalman filter. The Kalman filter keeps track of previous data given to it, and predicts where object **X** would likely move to in a following frame. Finally, it compares it's prediction with what the object detection model found and decides if they are the same object. At the end of this component, there is a list of tracked objects for each frame in the video that gets passed to the next component.

Figure*: Kalman Filter*

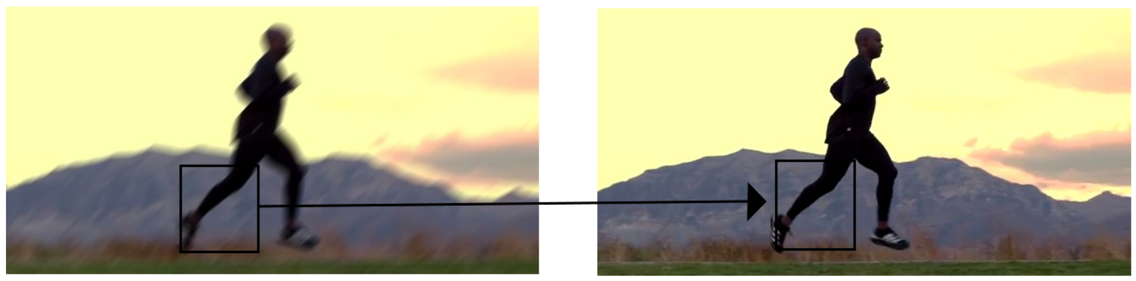

Prior knowledge of state
$\mathbf{P}_{k-1|k-1}$ $\hat{\mathbf{x}}_{k-1|k-1}$
Prediction step
Based on e.g. physical model
$\mathbf{P}_{k|k-1}$ $\hat{\mathbf{x}}_{k|k-1}$
Next timestep
$k \leftarrow k+1$
Update step
Compare prediction to measurements
Measurements
$\mathbf{y}_k$
$\mathbf{P}_{k|k}$ $\hat{\mathbf{x}}_{k|k}$
Output estimate of state

Component 3: Image Processing
After receiving a list of detections with their associated tracking IDs, the next component seeks to interpret and process the data to produce blurred images. First, it takes the path to the video and converts the entire video into its separate frames, stored on another folder, specified by the user. These frames are then stored on a 'frame array' which will store each frame's data as a numpy array, in order, so we can use their pixel data to create our blurred frame. We will then use a frame denoted as frame A, and its subsequent frame, denoted as frame B and add a blurred frame between them.

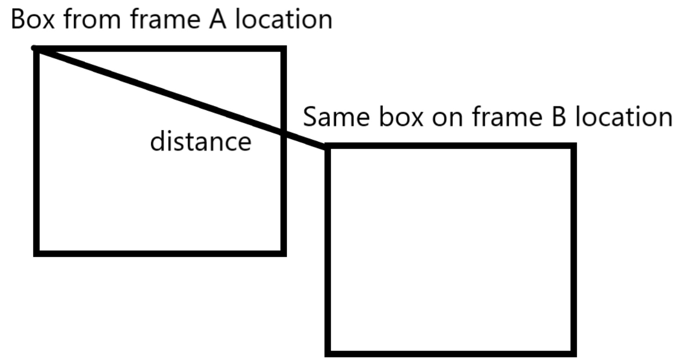
We check the boxes given for frame A and frame B. Each box has a corresponding id. If the id is in both frames, we apply a blur to it. If it is not, we do nothing.



Otherwise, it takes the two bounding boxes of the matching objects and attempts to create a new frame in between the two that smooths the motion. It does this by blurring frame A, then translating the detected object in the first frame to a point in between where it is in the first and second frame.



The intensity of the blur is determined by the velocity of the object that is tracked. The velocity is calculated by taking the displacement between the top-left corner of the two bounding boxes in pixels, adjusted for image size, and dividing it by the frame time, which is the time between frames. It then repeats this process, for each bounding box.

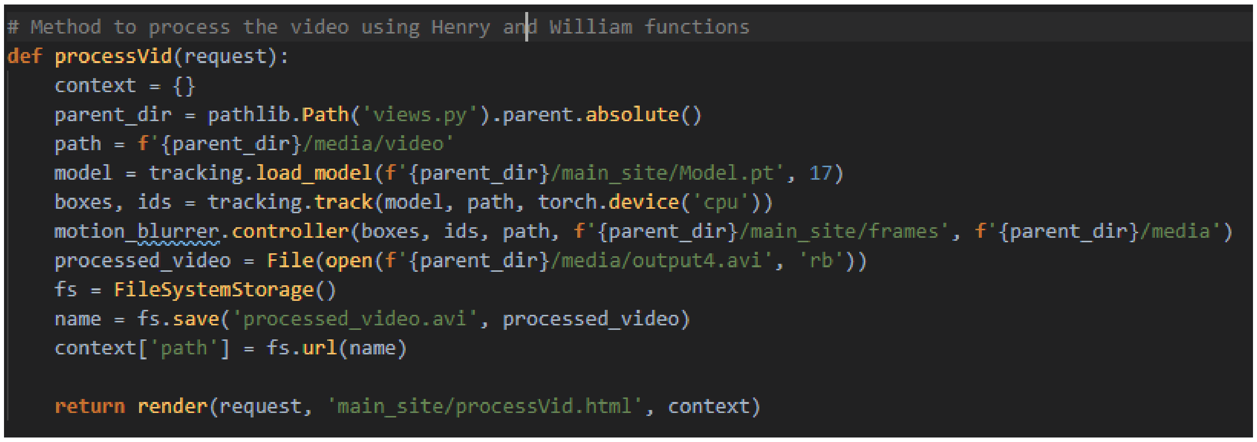


Then, this new blurred limb image is merged with the original frame.

This process is repeated for every limb to create the final blurred frame.

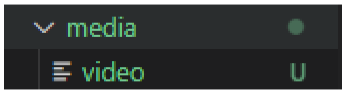
Component 4: Recreating the Video
These new blurred frames will then be inserted between frame A and frame B in the frame array. Then, with all the original and new frames, they are combined with OpenCV to create a new, continuous video with the original duration, but a higher fps. Using this blurring method as well as per object tracking, ensures that only moving limbs are blurred, and with faster movements, the transitions are a little smoother between limbs.

Component 5: Backend
After the video has been recreated and fully processed, it is saved in the media folder. The video is then wrapped with the Django file class object and its url is sent to the html file that allows the webpage to display the url as a downloadable link.



```python
# Method to process the video using Henry and William functions
def processVid(request):
    context = {}
    parent_dir = pathlib.Path('views.py').parent.absolute()
    path = f'{parent_dir}/media/video'
    model = tracking.load_model(f'{parent_dir}/main_site/Model.pt', 17)
    boxes, ids = tracking.track(model, path, torch.device('cpu'))
    motion_blurrer.controller(boxes, ids, path, f'{parent_dir}/main_site/frames', f'{parent_dir}/media')
    processed_video = File(open(f'{parent_dir}/media/output4.avi', 'rb'))
    fs = FileSystemStorage()
    name = fs.save('processed_video.avi', processed_video)
    context['path'] = fs.url(name)

    return render(request, 'main_site/processVid.html', context)
```



```
media
  video    U
```

Overall Contributions:

Henry:

- Modified the MPII human pose dataset by converting joints to approximate limb bounding boxes.
- Created the dataset class for both the limb detection data and second dataset.
- Trained the Faster RCNN object detection model.
- Compiled all detection and tracking code into the tracking.py file for easy use.
- Edited most of the video showcasing our project.

Nancy:

- Found and extracted the second object-detection dataset.
- Filtered and labeled the dataset then converted it into one csv file.
- Used SORT to transform the model from object-detection to motion-tracking

William:

- Created image manipulation code, i.e. blurring, rotating, translation. (component 3)
- Created video manipulation code that allows videos to be converted to frames, frames to be inserted in between certain frames of a video, and reconstructing frames back into a video. (component 3 & 4)
- Stitched previous code together to add blurred frames in between frames of a video.
- Provided abstraction so that inputs were boxes and ids from machine learning section, as well as paths for videos input, frames output, video output.

Ivan:

- Created the Django backend code
- Built the html and css templates for front-end aesthetics
- Routed different addresses for home page, about page, and page for processing and displaying improved video
- Housed the model code and motion blurring code to work in tandem in taking the video input and outputting the processed video in a reachable location
- Used Django's built in File class to store and wrap video inputs and outputs to allow efficient submissions and downloads

Conclusion:

Thank you for taking the time to go through our project and report. Again, if anything was unclear, please watch the video. It is very difficult to convey concepts and examples for a video based project via text. The video provides more, clearer, examples by virtue of being able to show the application at work.

Team Carnotaurus would also like to thank the CPEN 291 team for giving us the opportunity to learn how to use machine learning and the opportunity to use it in such an open-ended project.