Team D4: KUB Final Report

Kristina Banh, Umang Bhatt, Brian Davis
Department of Electrical and Computer Engineering
Carnegie Mellon University, Pittsburgh, Pennsylvania 15213
Email: {kbanh, umang, briandavis}@cmu.edu

I. Introduction

KUB is a personalized and portable dance trainer that's able to view your movements and offer corrections on a set of moves. Learning to dance is not only difficult, but also expensive and inconvenient as you typically have to commute to a dance studio in order to take class. This makes it harder for an aspiring or beginner dancer to start dancing, and harder for experienced dancers to continue dancing. KUB solves these problems by being accessible and being much less expensive than conventional teaching options.

Currently, there is an application that utilizes a similar technology and method as a trainer, but it's for working out as opposed to dance. There are also dance games that ”teach” dance moves, but they offer no correction aspect and dont actually teach the movement very well since its purely for entertainment. This gives KUB an advantage edge as it is an application in a new field. KUB approaches dance correction using joint variance ranking, which automates the correction with a trainer feedback goal of 30 seconds after a movement is complete. This feedback should be relevant, correct, and useful to the user.

II. Design Requirements

This project can be split into two main components: The pose correction pipeline that runs in the backend of the application, and the user interface that provides the user access to the corrections. Each of these sections have a few key requirements that they have to meet.

A. Pose Correction Pipeline

Since we strive to replace the functions of a real dance teacher, we try to mimic the methodology that they would use in real life. When a dance teacher instructs, there are a few steps that they go through. First they look at their student's movements and remember what they see. They then analyze each aspect of the movement, and compare it with what they think the ideal version should be. Lastly, they convey this result to the student.

In order to be faithful to this process, our pose correction pipeline requires a few key components. First is the ability to capture the user pose or movement and store it. Second is the ability to convert this stored media into a format that can be analyzed easily. Third is the ability to assess the user pose and compare it to a stored dataset of ideal poses. Last is the ability to convey the corrections back to the user.

1) Movement Capture: We must be able to capture the entirety of a person's movement while it is being performed. Therefore, we need some sort of camera that is attached to the system that is in a good position while the user is performing the movement. This camera should ideally not need to have a high resolution, as most users do not have immediate access to an HD web camera or the sort. Due to this constraint, we do require that the input image be at least of a 480p resolution in order to keep the hardware requirements low while maintaining enough information for us to analyze the movement. Since most laptops come with 720p web cameras, this should not be an issue.

2) Pose Estimation: Once the movement has been captured as either a video or an image, we must now have a way to translate the media into a form that can be analyzed. This process must be portable, ruling out methods like the Microsoft Kinect, as well as consistent/robust to a variety of backgrounds. Though we can handle some frames being mislabeled, we require that most frames be converted correctly. For our purposes we require a 95 percent success rate for the pose estimation. When the estimation is incorrect, the algorithm returns a pose in which some joints are clearly incorrectly placed outside of normal ranges of motion. Therefore we can measure this requirement with a script that checks the positions of the joints in relation to one another, and asses if they are within normal human motion ranges.

3) Pose Correction: After the pose has been extracted from the image in a meaningful way, we must then have a method by which we can compare it to a stored example (the ideal version). In order to do this, we need some metric to compare two poses and assess their similarity. We must also have a way to rank the importance of each aspect of the movement so that we can prioritize the order in which we give corrections. This is so that we do not overwhelm the user with more corrections than they can handle at a time, and can correct more important features of the move before less important ones. Since dance is inherently subjective, we evaluate these requirements by polling trained dancers to evaluate our corrections. We deem it a success if the dancers evaluate our corrections above a 80 percent average in the metrics of feedback priority and feedback correctness.

B. User Interface

Since there are no interfaces that dance instructors typically use to teach, we need to devise a way for the user to interact with our correction pipeline that feels natural and mimics an interaction with a real instructor.

From the student's point of view there are a few things that happen during a dance lesson. Depending on the nature of the lesson, the student either chooses what move they will be performing, or is told to perform a movement. At this point, the user either already has an idea of how to perform the movement, or needs to see the instructor perform the movement to get a clear idea of what they need to do. The user then performs the movement in front of the instructor, and waits to be corrected, at which point the cycle repeats.

In order to mimic this process, we require a few key components of our UI. First is a page for the user to decide what movement they would like to perform. This page should be easy to navigate, and provide a clear categorization based on the type of move. Second, we need a page for the user to view the movement that they will be performing in its entirety. The user should be able to view the move until they think they are ready to perform. Third, we need a page for the user to perform the movement. Ideally, the user should be able to see themselves in order to mimic the fact that dance studios usually have mirrors. Lastly we need a way for the user to receive their feedback. The feedback should be clear and easy to digest, and displayed in a formatted way so that the user knows what they are receiving corrections for. In order to test these requirements, we test the users on the categories of ease of use and clarity of feedback, and deem it a success for averages above 80 percent.

III. Principle of Operation

In this portion of the document, we will review the core design of the pose estimation pipeline. First, we will describe how we get the angular ranking Φ from a frame V. Then, we will describe how to pick the angle ϕ_j to correct.

First let us introduce some notation. Let $\mathbf{V}$ be a video feed from a user. Let $V_1, \ldots, V_t \in \mathbf{V}$ be the t frames of the video feed. Without loss of generality, let us fix a frame V, which would be a still image; this allows us to avoid the temporal nature of pose correction to start.

A. Angle Estimation

For our purposes, we identified that mapping the pose into the angular domain will give us a much better representation of the user's pose characteristics than simply using the 2D points. It also allows us to bypass the issues of scale and translation that the Euclidean space suffers from. By identifying the angles between joints, we no longer have to require the user to be centered in the same way as our example data, and we are able to remove the issue of user height entirely. It also simplifies our problem very nicely without the loss of important information.

Given V, we identify the user's pose via AlphaPose (a real-time accurate pose estimation and tracking system), $f(V) = P \in R^{12\times 2}$. AlphaPose has improved accuracy over OpenPose. AlphaPose gives us one tuple $p_i = (x_i, y_i)$ for each joint i. See Figure 1 for details on all twelve joints, where the top left of the image is considered $(0,0)$ and the bottom right is considered (w,h). Note that we deal with pose estimation measures in a 2D plane and that we do **not** retraining AlphaPose; rather, we use pretrained weights for inference.

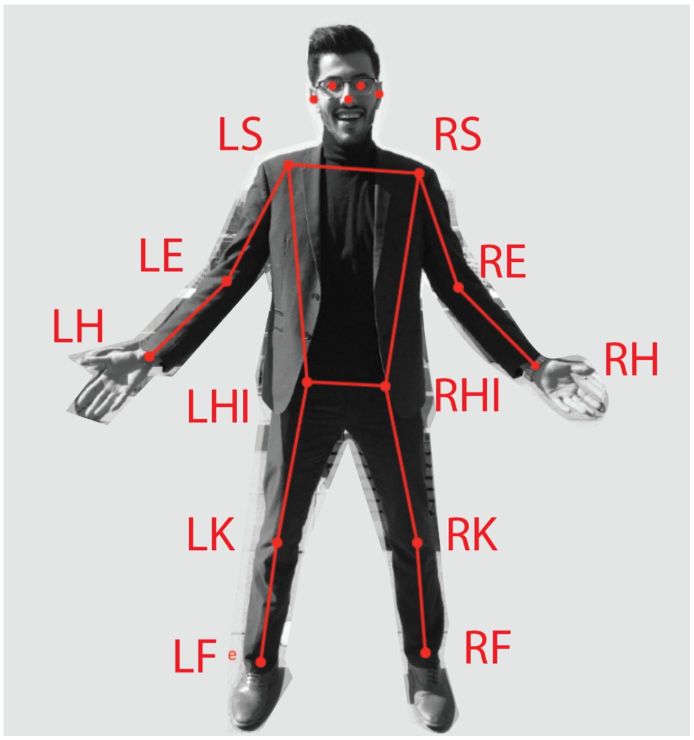


Fig. 1. Labeled Joints

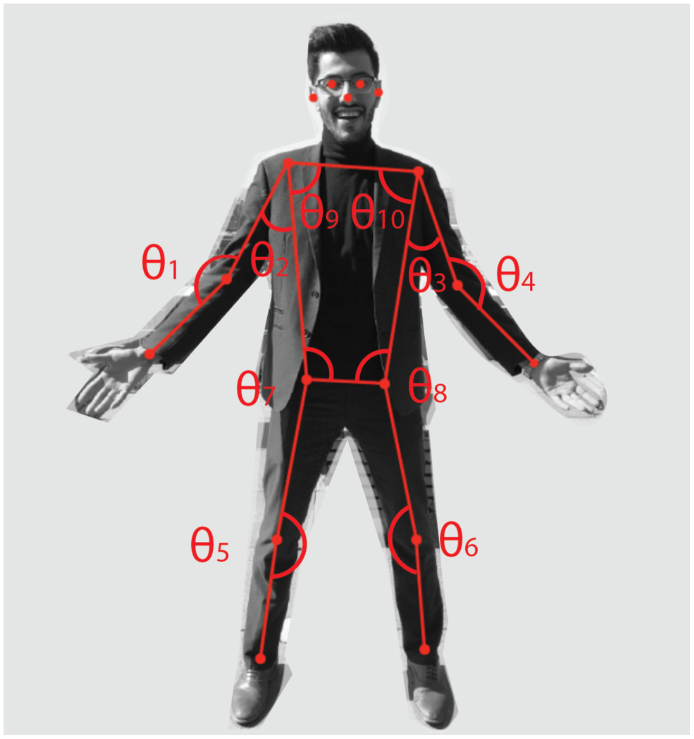


Fig. 2. Labeled Angles

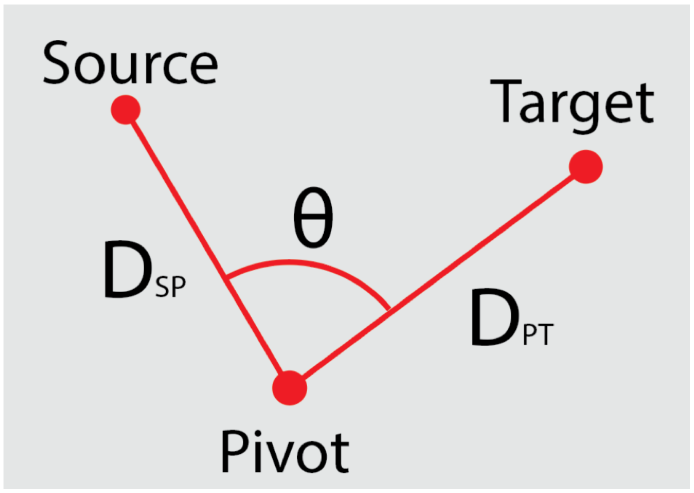


Fig. 3. Diagram of angle components

We identify ten angles for our angular ranking $\Theta \in R^{10}$, as shown in Figure 2. In Table I, we show the three joint tuples $p_i, p_j, p_k \in P$ used to find all ten angles, $\theta_i \in \Theta$. The joints are written in the following format: Source (S), Pivot (P), and Target (T). If we assume a triangle between SPT, we want to identify the angle at P to be θ, as shown in Figure 6. Since we have three points in Euclidean space, we want to find the distance between S and P: D_{SP}, between P and T: D_{PT}, and between S and T D_{ST}. We can do this with any real valued distance metric, d. For example, we can let d be the ℓ_2 distance (which would be the norm of the displacement vector between two points in 2D), resulting in the following. We use this distance metric for our application, as we deemed it to be most faithful to the way in which polled dancers evaluated the similarity between given poses.

$$\begin{aligned} D_{\text{SP}} &= d(\text{S}, \text{P}) \\ &= \sqrt{(x_s - x_p)^2 + (y_s - y_p)^2} \\ &= \|S - P\|_2 \end{aligned}$$

In practice, we only use the first six angles, as the others were not informative and were noisy over the training data. We also tried out the following distance metrics to see if the angles between certain joints are more sensitive when we use different distance metrics to get the side lengths of triangle SPT.

We use the same distance metric d to get all side lengths. Once we have D_{SP}, D_{PT}, and D_{ST}, we solve the following to get θ, which follows from the Law of Cosines. We first present a general vectorized formulation, followed by a formulation specific to the Euclidean space.

$$\begin{aligned} \theta &= \arccos\left(\frac{\vec{D_{\text{SP}}} \cdot \vec{D_{\text{PT}}}}{\|\vec{D_{\text{SP}}}\|\|\vec{D_{\text{PT}}}\|}\right) \\ &= \arccos\left(\frac{D_{\text{SP}}^2 + D_{\text{PT}}^2 - D_{\text{ST}}^2}{2D_{\text{SP}}D_{\text{PT}}}\right) \end{aligned} \quad (1)$$

ANGLE (θ)	SOURCE (S)	PIVOT (P)	TARGET (T)
θ_1	LH	LE	LS
θ_2	LE	LS	LHI
θ_3	RE	RS	RHI
θ_4	RH	RE	RS
θ_5	LHI	LK	LF
θ_6	RHI	RK	RF
θ_7	LS	LHI	RHI
θ_8	LHI	RHI	RS
θ_9	RS	LS	LHI
θ_{10}	RHI	RS	LS

TABLE I
COMPREHENSIVE SOURCE, PIVOT, AND TARGET SPECIFICATIONS FOR EACH TRACKED ANGLE

After consulting with dancers, we learned that it would be difficult to ask a dancer to correct their left hip (θ_7, the angle between one's torso and hip line) separately from their right hip, or their right shoulder (θ_{10}, the interior angle between one's torso and shoulder) separately from their left. As such, we need to edit our angle vector to be the combination of both, Θ_{OLD}. We define the following two angles.

$$\theta_{7'} = \theta_7 - \theta_8$$

$$\theta_{8'} = \theta_9 - \theta_{10}$$

As such, we define an angle vector $\Phi \in R^8$, where we the first six elements of Φ and Θ are the same and then we set ϕ_7 to $\theta_{7'}$ and set ϕ_8 to $\theta_{8'}$.

B. Angle Correction

Once we identify Φ for the current frame V, we need to rank all eight angles by how much they vary from the expert ground truth, $\mathbf{E}$. Let $U_{a,1}, \ldots, U_{a,t} \in \mathbf{U_a}$ be video frames from expert a. Let $U_{a,t}$ be the same frame as frame V_i from the user. Using Equation 1, we can find the expert angle vector $\Phi_a \in R^8$.

Since any angle vector Φ is invariant to scale or translation along the 2D plane of the frame, we take an expectation of angle over all experts performing a given pose, irrespective of whether the angle vector estimates come from the same or multiple instructors. Let $\mathbf{U}$ be the set of k expert video feeds. We can get an aggregate expert angle vector, $\mathbf{E} = \Phi_{\text{agg}} \in R^8$, by taking the empirical mean of all angle vectors from the k experts.

$$\mathbf{E} = \Phi_{\text{agg}} = \frac{1}{k}\sum_{i=1}^{k} \Phi_k$$

ANGLE (ϕ)	JOINT OF INTEREST	SUPPORTING JOINT
ϕ_1	RIGHT HAND	RIGHT ARM
ϕ_2	RIGHT ARM	TORSO
ϕ_3	LEFT ARM	TORSO
ϕ_4	LEFT HAND	LEFT ARM
ϕ_5	RIGHT LEG	RIGHT FOOT
ϕ_6	LEFT LEG	RIGHT FOOT
ϕ_7	HIPS	N/A
ϕ_8	SHOULDERS	N/A

TABLE II
ANGLE-JOINT NAME MAPPING

Our goal is to find how much Φ differs from $\mathbf{E}$ and in what elements they differ: we correct the dimension along which they differ most. To better identify which dimension of Φ to correct, we need to make stronger assumptions.

First, since we have k expert angle vectors, we actually have a distribution in the space of angle vectors. If we assume a normal distribution over the angle vectors, then we can assume the following:

$$\Phi_k \sim \mathcal{N}(\mu, \mathbf{\Sigma})$$

Thus, the mean vector $\mu \in R^8$ contains the mean value for all eight angles of interest in the expert distribution and the covariance matrix $\mathbf{\Sigma} \in R^{8\times 8}$ contains the variance for the eight angles of interest along the diagonal. Therefore, let $\rho = \text{diag}(\mathbf{\Sigma}) \in R^8$.

We can then rewrite each element of our user angle vector, Φ, in terms of the expert mean and expert standard deviation, effectively finding z-scores in the statistical sense.

$$\phi_i = \mu_i + \alpha_i(\sqrt{\rho_i})$$

$$\alpha_i = \frac{\phi_i - \mu_i}{\sqrt{\rho_i}}$$

We are left with eight scalar α values each representing how many standard deviations away from the expert a user lies. We then identify which user angle differs most from the expert distribution; this is the angle we want to correct.

$$\phi_{\text{fix}} = \phi_i \in \arg\max_i |\alpha_i|$$

In order to prompt the user to correct her behavior, we need to tell her which joint to correct and how to correct it. To start, we fill in the statement below, replacing [DIRECTION] with either "Increase" or "Decrease" and replacing [JOINT] with the joint name from Table II. To get the direction of the correction, we leverage the sign of the α value in question: positive alpha corresponds to "Decrease" and negative alpha corresponds to "Increase". This output is then sent to a Text-To-Speech engine to then play back to the user.

"[DIRECTION] the angle between your [JOINT] and your [SUPPORTING JOINT]"

If ϕ_7 or ϕ_8 need to be corrected, we use the following verbiage.

"Level your [JOINT]"

In essence, we take advantage of the variance in the expert angle distribution to chose which joint to correct. We then leverage that variance ranking to correct the user's pose. Every time the user provides a frame V we correct at most one joint. To facilitate termination, we impose a lower bound on $|\alpha_i|$.

If $|\alpha_i| \leq \epsilon$ holds, then we do not consider correcting the i-th angle. We intend to set epsilon to 1: this implies that we only correct movements to be within one standard deviation of the expert mean. Once all eight angles satisfy the aforementioned condition, we show the following message.

"Great job, you've mastered this pose!"

C. Temporal Pose Estimation

Once we have corrections for individual images, we look to extend the methodology to allow for corrections of more dynamic movements captured in video. To do a real time correction of dynamic movements is both unfeasible due to the processing time required by the pose estimation algorithm, as well as unnecessary, since the user will not be able to process feedback in a meaningful way while performing a movement. Therefore, we decided on providing feedback after the movement is performed, in a way that is clear and easily digestible for the user.

The way that we accomplish this is by correcting the key frames of a dynamic movement. For our purposes, we define key frames as the most important still frames within a movement. They are the poses contained in dynamic movements that dancers transition in and out of during the course of a full move. For example, during the plie -a beginner dance move- the dancer will transition in and out of 4 key frames: one at the start with the hands positioned downwards, one in which their arms are lifted in front with the knees bent out, one in which their legs straighten out again and their arms are lifted straight to their sides, and one in which they return to the original pose with the hands positioned downwards. Analogous frames for a push-up would be the position at the top with the arms extended, and the position at the bottom with the arms bent. By correcting each of these frames individually, we hope to correct the overall form of the movement, which is just these movements strung together. It is worthy to note that this is how dance teachers traditionally correct movements as well. By mimicking this routine, we hope to provide as close a replacement for a real dance teacher as possible.

In order to perform this key frame correction, we must first be able to do two things. The first is to identify which key frames are necessary to correct within a given move. This we do by having trained experts look at a given video of an instructor performing a move, and manually identify the frames which they think are key. We take these frames to be the underlying data set for the move.

Once we have the instructor key frames identified, we then need a way to find the frames within the user video that we would like to correct. In order to do this, we create a frame matching algorithm that utilizes the distances between frames to find the closest match to a given key frame. One way to take these distances would be to compare the raw coordinates of

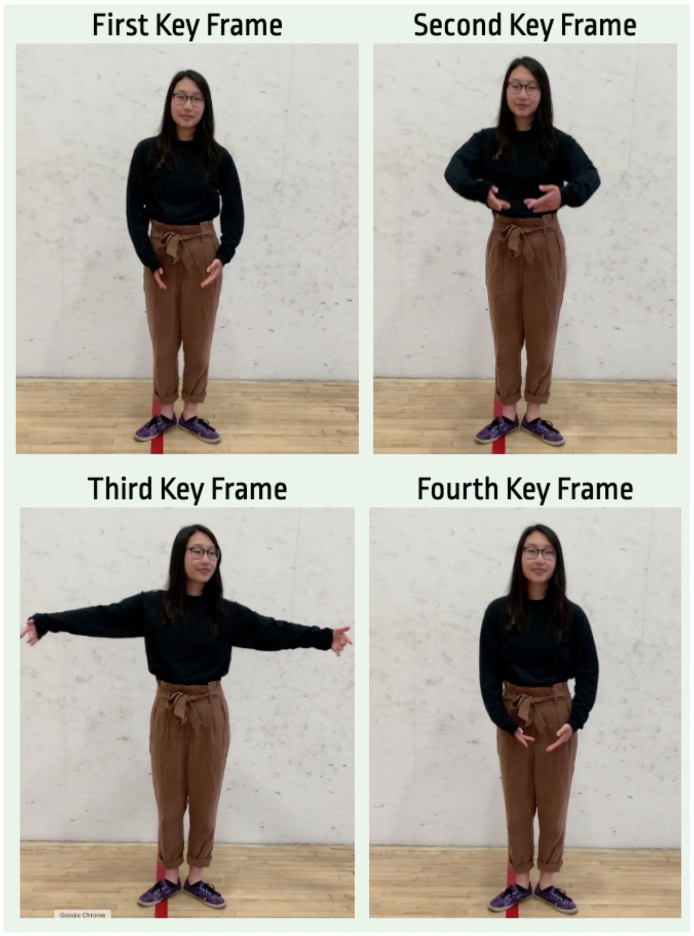


Fig. 4. Example Key Frames for Port de Bras Movement

the joints created by the pose estimation algorithm. However, this would be heavily influenced by the size, position, and orientation of the user. Furthermore, this would not align with the way in which we find the corrections, in which we consider only the pose angles.

Therefore, we first convert the poses to their angle domain, and find the distances between frames there. Like before, this allows us to bypass any translations, rotations, or scaling within the data. After experimenting with multiple distance metrics, we found the ℓ_2 distance to be most faithful to how polled dancers interpreted distances between poses.

With a distance metric specified, we then are able to compare instructor key frames to those of the user within their video. We identify the user key frame by sweeping through segments of the video and finding the minimum distance between the instructor frame and the user's. We then search the next segment of the video for the next key frame, repeating until all key frames have been matched with a corresponding user frame. Once the user frames frames have been identified, we then run the original image correction algorithm on them and output the corrections sequentially.

IV. Design Trade Studies

We found three subsystems to have considerable trade-off. First: would it be faster to run our pose estimation on AWS or on our local computers? Second: would OpenPose or AlphaPose be faster? Third: would Node or Flask be a better backend for our system? We now describe and expand on these trade offs.

A. *Local Computer vs. AWS*

Our local computers were not optimized for efficient pose estimation. Unfortunately, pose estimation using CPUs is horridly slow. We explored ways to get our estimation under our desired metric of doing the pose estimation on a video in under 30 seconds. As such, we decided to explore running our pose estimation on a GPU where we would get the speedup we need to meet the metric.

We worked on getting an initial pose estimation implementation of AlphaPose up on AWS. Similar to when AlphaPose runs locally, we run AlphaPose over a set of frames and give the visualize the resulting jsons as a graph. This AlphaPose computation is accelerated by the GPUs on the AWS instance; even though we only used one GPU at a time (multithreading would not be useful for our pose estimation use case given the short length of the video - small number of frames - and given that aggregating the results of each core would add to the irreducible time for these instances, see below), we performed estimation on a 40 second video about 20 times as fast.

By running our code on AWS, we move an image from 20 seconds for estimation to about 8 seconds for estimation. This speedup considerably helped decrease the latency of the pipeline as a whole. To that end, it was natural for us to design our system with AWS for pose estimation instead of unrealistic local computation that not only hindered efficiency but also harmed the experience of a person using KUB.

B. *OpenPose vs. AlphaPose*

We decided to run a comparison between AlphaPose and OpenPose for running pose estimation on the AWS instance. We needed to know which pipeline would be more effective for our use case. We note that our pipeline has a 8 second up and down time that is irreducible but any other time is added due to the slow pose estimation. As such, we wanted to explore if OpenPose is faster for our use case on a GPU.

OpenPose runs smoothly but has much overhead if we want to retrofit it to optimize for our task. Running vanilla OpenPose led to a respectable estimation time for a 400 frame video (on one GPU), but was still over our desired metric of doing the computation in under 30 seconds. Though the times were comparable at first, when we added flags to our AlphaPose command to reduce the detection batch size of the pose estimation network, set the number of people to look for in the resulting frames, and reduce the overall joint estimate confidence interval, we were able to get blazing fast estimation from AlphaPose (20 seconds + 8 seconds for the up down), which considerably beats OpenPose and rivals state of the art pose estimation in industry. This means we also hit our metric of doing end to video pose correction in under 30 seconds.

Rather than taking 29 seconds for a video and 15 seconds for an image, we then wanted to break sub-20 second inference for a video and sub-10 second inference for an image. The best place to shave off time was in the pose estimation by increasing the speed of AlphaPose and decreasing the frame rate of the original video. It turned out that the UI saved the video as a *.webm file and AlphaPose did not take in this type. As such, we had to use a conversion function (ffmpeg was the one we picked) to convert it from *.webm to *.mp4. Unfortunately, this conversion actually expanded instead of compressed the video which led to even slower pose estimation by AlphaPose.

By setting a reduced frame rate flag, we were able to subsample in the video and then run the shorter video through the pose estimation network (with the relaxed confidence, lower number of people, and an increased detection batch). This sub-sampling would effectively reduce the pose redundancy and decrease the latency of the overall pipeline. With these changes, we got the video estimation down to 1 second for a 4 second long *.webm file (with added time for the ffmpeg call with the sub-sampling). This finding not only advocates for AlphaPose over OpenPose but also AWS over local computation.

C. Node vs. Flask

The tricky part of the UI was in determining how to build a tool that allows us to respect the interface of the server whilst being agile to deploy on the web. We initially used no web-backend. As researchers and full stack developers, our web development experience was mediocre at best. We found that HTML and CSS alone for a website would be insufficient for end users even if our backend and video pipeline was blazing fast. To that end, we began to learn and leverage web backends that adhere to a particular server client relationship. We started by using Node.js alone in the HTML/CSS framework itself. Unfortunately this suffered from multiple issues. Firstly, we struggled to gain root access to the web camera and then failed in capturing/saving the given video. Moreover, we also struggled when developing with respect to the server-client relationship.

With the issues of Node.js adding up, we decided to switch over to the Flask framework: a classic web development framework that would give us the agility we need to develop a server-client API that makes it easy to download/save videos and to initiate a bash script that sends the requisite files to AWS for computation. With Flask, we not only had an easier time implementing our KUB front end but also had a smoother experience in integrating it with our AWS instance, where AlphaPose was running for pose estimation. We used another bash script to run the corrections which then return to the Flask framework where the corrections are displayed with relative ease. In essence, switching to Flask instead of Node made integration with our robust back end feasible.

V. System Description

On a high level, our system has to accomplish a few things:

1) Capture a video feed of the user
2) Extract pose estimates from the user feed
3) Compare the user poses with stored instructor poses
4) Decide what movements need the most immediate correction
5) Communicate the feedback to the user

A. Video Feed

The video feed for our project is captured by whatever camera is on the user's device. For our purposes, we are using the stock Macbook Pro web camera since we would like to ensure that this works even with low quality web-cameras. However, the program works with any web-camera.

B. Pose Extraction

We initially thought that we would be using OpenPose as our estimation algorithm. We then briefly changed to PoseNet due to its ease of use, but ultimately settled on using AlphaPose since it provided robust pose estimations that were computed quicker than the above methods with higher confidence.

During the project design phase we also had envisioned the program running solely on the CPU. However, we quickly realized that that would be much too slow for a user to wait for. Therefore we moved the pose calculations to a AWS instance that we communicate with through our application. The system sends the user video to the instance after it has finished recording, waiting for the estimation to finish and for the JSON file containing the pose estimates to be sent back.

C. Comparison and Feedback

Once the JSON is received by the application, it runs the algorithms outlined above to calculate the corrections for the user. For still poses, the application only does a single pose comparison between between the user image and the instructor pose. However, for dynamic movements, the application runs the frame matching algorithm to find the frames within the user feed that are most similar to the key frames, and then run the image corrections between each of these pairs.

The corrections are saved locally as images, and are piped into the UI via the Flask framework.

D. Data Creation and Storage

In order to conduct the comparisons, we needed a large amount of instructor data. This is in order to get a better sampling for the variance and mean of each joint within a key frame or pose. We gathered this data by asking several dancers to come to a controlled environment where we could guarantee that there would be no interference from other people entering the frame or from poorly lit/colored areas affecting the pose estimation. We then asked each dancer to perform moves from a given list multiple times apiece. We recorded each move, and asked that the dancers repeat any moves they thought were off.

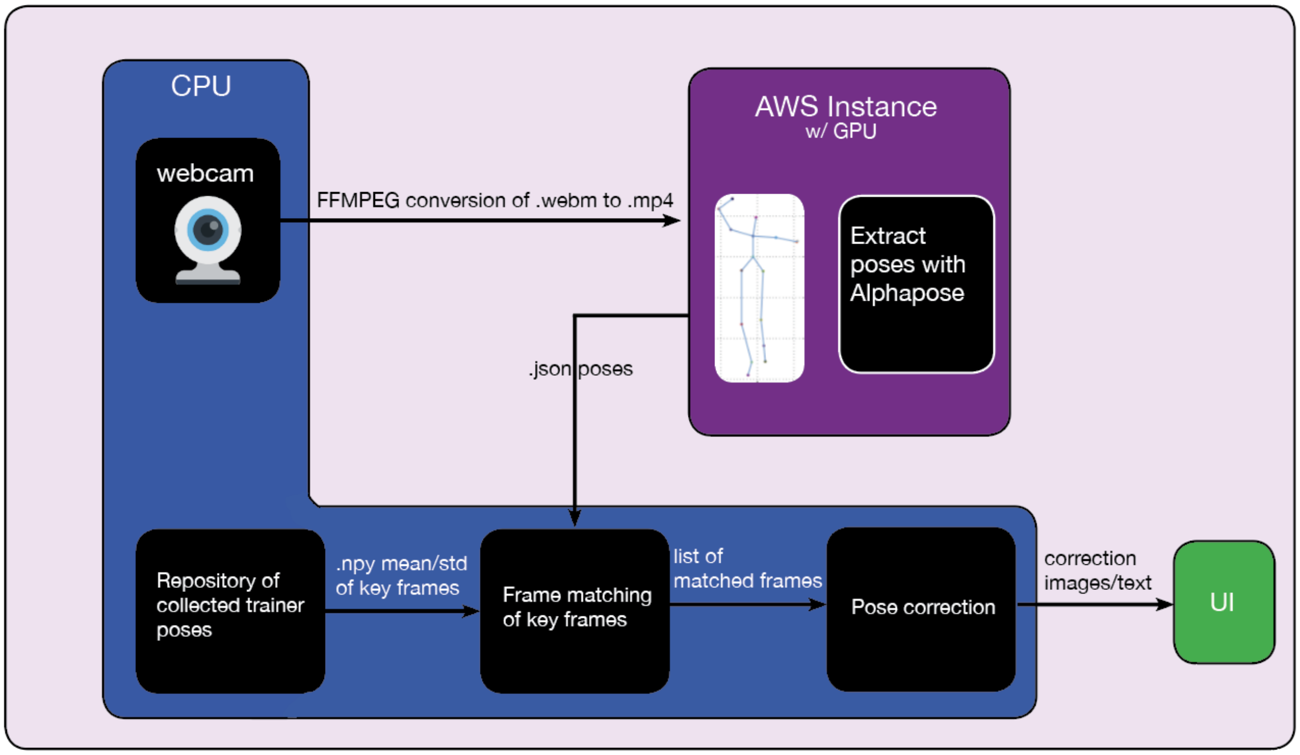

CPU
webcam
FFMPEG conversion of .webm to .mp4
AWS Instance
w/ GPU
Extract poses with Alphapose
.json poses
Repository of collected trainer poses
.npy mean/std of key frames
Frame matching of key frames
list of matched frames
Pose correction
correction images/text
UI

Fig. 5. Block diagram of system components

We ran both the still as well as the videos through the pose estimator to convert them to raw poses. We then converted them further into the angular domain. For the images we took the variance and mean of the data for each of the poses and stored them in our database based on their move type. For the videos we had Kristina label the number as well as the location of the key frames. We then took the mean and variance of the angles for the key frames and saved them in our database.

E. Flask Framework

We previously thought that we would be creating our application without a framework. However, we ran into issues with accessing local files, as well as running certain backend python scripts. In order to fix these issues we ported our application to the Flask Framework that runs on python. This enabled us to execute our backend scripts in a much more simplistic way, and allowed us to access our downloads in which we were storing the user images and videos. Flask further allowed us to test and deploy our design faster, allowing us to iterate more quickly.

F. AWS Instance

We realized running AlphaPose locally was quite expensive and time consuming. We then updated our pipeline to run on an AWS instance. In the end, this updated video pipeline ran in 11 seconds total (including up-down to the instance and ffmpeg) and ran in 8 seconds for an image. Unfortunately, the AWS instance we used for this was a P3 instance (which had an unsustainable cost of $12/hr). So we settled for the normal P2 instance (which had a cheap cost of $0.90/hr). This pipeline on the P2 ran a video through in 15 seconds and an image through in 9 seconds. Both of these times far surpassed our original metrics.

VI. User Interface Design

Let's imagine user Bob as he interacts with KUB Trainer. When he first opens the web application, he is welcomed and is asked to select which dance pose or movement he wants to learn. He decides to choose the first arabesque tendu pose and is then taken to a screen with a demonstration of the pose by KUB. Bob changes his mind, and decides to choose a different pose. He goes back to the initial selection screen, but decides to choose the same pose again. After watching KUB perform the pose he wants to learn, he's ready to do the pose. He starts the pose capture and is given 10 seconds to back up from his web camera and to get into clear framing AS the countdown ends, he tries to copy KUB's demonstration, which is still up on the screen next to his own camera feed that acts as both a mirror and a way to record his joint data. After up to 30 seconds, KUB gives him a correction. Bob wants to see KUB demonstrate once again to better understand his correction, so he goes back to the earlier screen where KUB performs his pose. He gets a better visualization of what his first arabesque tendu should look like, so he tries the pose again. This time, he gets a different correction and chooses to try another time, applying this new correction. After being

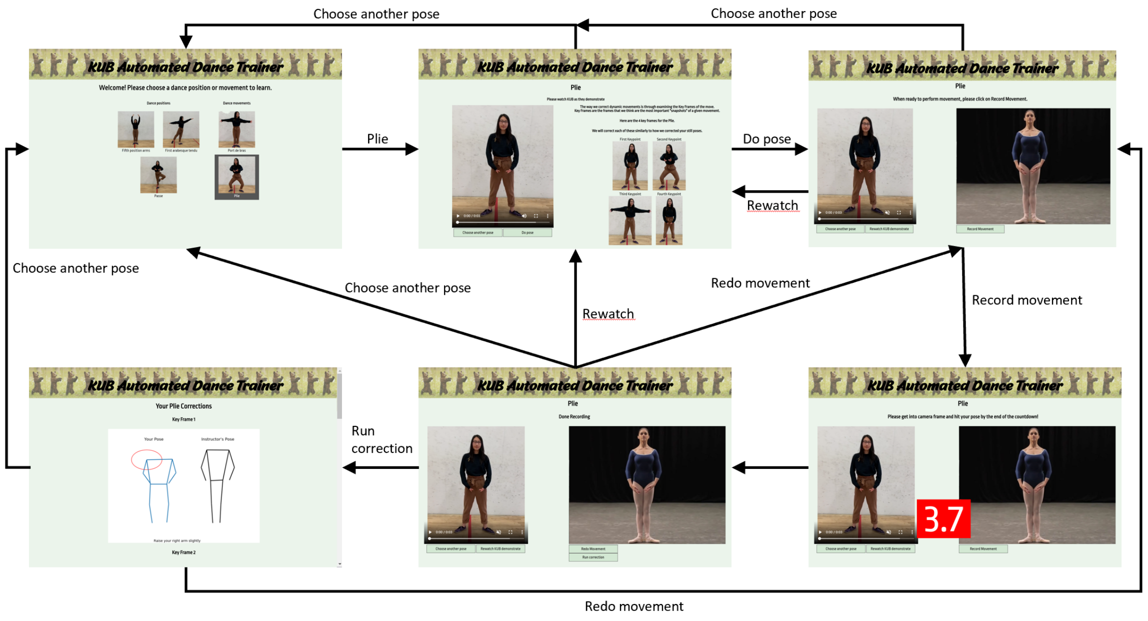


Fig. 6. User flow

satisfied with his performance on the current pose, he decides to try something new and goes back to the initial selection screen. Bob wants to try a dance movement instead of a pose this time, so he chooses a different movement than before. He is guided to a similar KUB demonstration, except this time KUB also specifies key frames that Bob should focus on hitting throughout the movement. After the same countdown as before to perform the movement, he performs the new movement as KUB's demonstration plays for reference. KUB gives him a corrections on each of the key frames of his chosen movement up to 30 seconds after he completes the movement. He continues using KUB Trainer to learn how to dance.

Our UI is designed to have as few moving parts as possible so that the user can delve into learning their movements without having to learn how to use the app first. The app consists of 4 main pages.

The landing page simply show the moves that we currently have available, separated into categories of dance poses and dance movements. Next to each option, we have an example image of what each pose is, or an example image of one of the key frames of each movement.

Once a user selects a particular move, it takes them to the instruction page for that movement. Here, they are able to see a demonstration from the instructor as either a picture of the pose or a video of the movement. These are shown mirrored to the user so that the screen acts the same way as a mirror in a dance studio would. For movements, they can also learn what the key frames are, which are basically the important poses the chosen movement goes through that the user should focus on hitting. The user may decide to return to the movement selection page at any point, allowing them to change their move.

Once the user is satisfied that they know what to do, they can select the "DO POSE" option that takes them to the performance page. Here, the user can still view the KUB demonstration, but can also view themselves in a mirrored video feed. When ready, the user initiates the count down of 10 seconds, backs up from the web camera, and makes sure all of their joints are within the frame. At the end of the countdown, a picture is taken for poses or a video is taken for movements as the demonstration video plays. Once the media is captured, it gets saved and the user can start the correction.

The program calculates what instructions need to be made. When calculation is done, the user is taken to a new screen with visualizations of the corrections. Dance movements show up to four corrections for each key frame whereas dance poses show up to four corrections for the individual pose. Afterwards, the user can either choose to repeat the movement or return to the home page to choose another movement.

VII. Project Management

A. Schedule

In our original schedule from the design report, we allotted some slack time at the end to account for tasks that took longer than predicted, and the slack time only ran until the final presentation instead of the final demonstration. Throughout the course of the semester, some of the initial tasks laid out took longer than predicted, either due to them being more complicated than initially thought or due to project members' busy schedules. More tasks also were added as

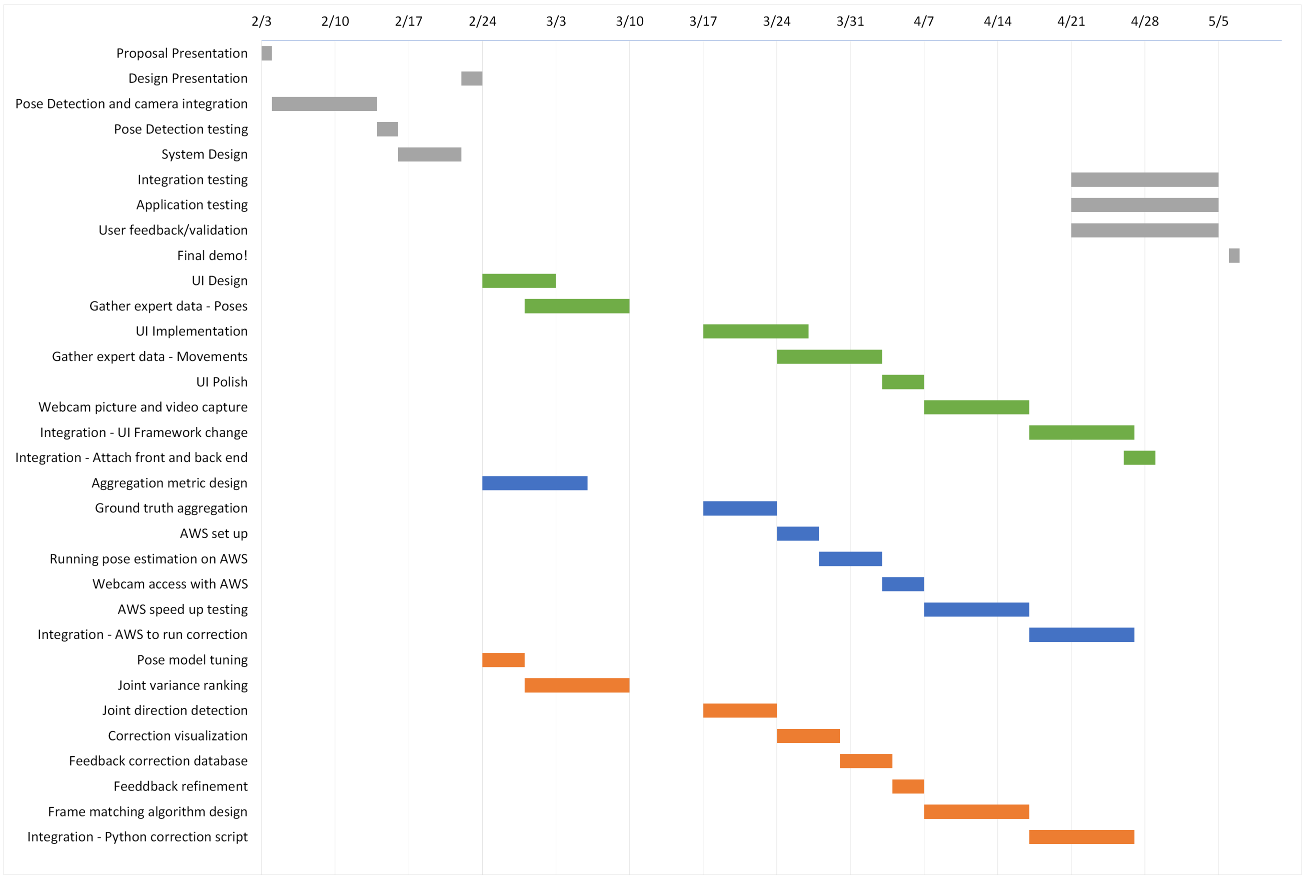
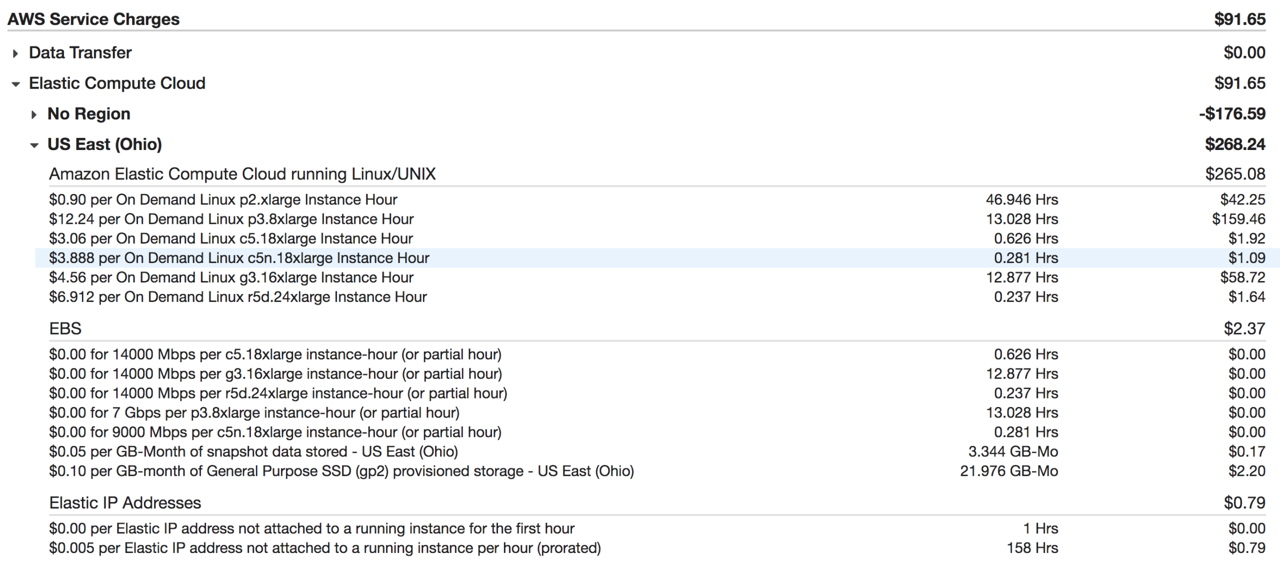


Fig. 7. Schedule



AWS Service Charges		**$91.65**
Data Transfer		**$0.00**
Elastic Compute Cloud		**$91.65**
No Region		**-$176.59**
US East (Ohio)		**$268.24**
Amazon Elastic Compute Cloud running Linux/UNIX		$265.08
$0.90 per On Demand Linux p2.xlarge Instance Hour	46.946 Hrs	$42.25
$12.24 per On Demand Linux p3.8xlarge Instance Hour	13.028 Hrs	$159.46
$3.06 per On Demand Linux c5.18xlarge Instance Hour	0.626 Hrs	$1.92
$3.888 per On Demand Linux c5n.18xlarge Instance Hour	0.281 Hrs	$1.09
$4.56 per On Demand Linux g3.16xlarge Instance Hour	12.877 Hrs	$58.72
$6.912 per On Demand Linux r5d.24xlarge Instance Hour	0.237 Hrs	$1.64
EBS		$2.37
$0.00 for 14000 Mbps per c5.18xlarge instance-hour (or partial hour)	0.626 Hrs	$0.00
$0.00 for 14000 Mbps per g3.16xlarge instance-hour (or partial hour)	12.877 Hrs	$0.00
$0.00 for 14000 Mbps per r5d.24xlarge instance-hour (or partial hour)	0.237 Hrs	$0.00
$0.00 for 7 Gbps per p3.8xlarge instance-hour (or partial hour)	13.028 Hrs	$0.00
$0.00 for 9000 Mbps per c5n.18xlarge instance-hour (or partial hour)	0.281 Hrs	$0.00
$0.05 per GB-Month of snapshot data stored - US East (Ohio)	3.344 GB-Mo	$0.17
$0.10 per GB-month of General Purpose SSD (gp2) provisioned storage - US East (Ohio)	21.976 GB-Mo	$2.20
Elastic IP Addresses		$0.79
$0.00 per Elastic IP address not attached to a running instance for the first hour	1 Hrs	$0.00
$0.005 per Elastic IP address not attached to a running instance per hour (prorated)	158 Hrs	$0.79

Fig. 8. Bill of Materials

we implemented more features, went through testing to meet our metrics, found bugs and problems along the way, and polished the application. Due to all of this, our project schedule morphed and changed along with our tasks, responsibilities, and personal schedule to ultimately cover the extra week between presentations and demonstrations. Also, all of our initially planned slack time was spent working on these extra tasks and on integration. See Figure 7 for a breakdown of the final schedule.

B. Team Member Responsibilities

To divide up the work, Kristina was mainly in charge of the web application design and implementation, which includes getting user feedback to iterate on both the UI design and the instructor feedback and making sure all the UI elements work and flow together. She also worked with Brian to integrate the algorithms and data with the front end, making sure the back end scripts had access to local files and could run properly. She was also responsible for gathering the expert dance trainer data as well as identifying key frames of each movement, since she has the dance expertise.

Brian and Umang worked on the algorithm design and implementation, as well as the data pipelining. While they mostly worked together, Umang focused more on designing the aggregation metric and aggregation of the ground truth data that Kristina gathered, as well as the language model that translates the raw joint correction into an understandable statement. Umang also generated the aforementioned z-scores and took the lead on our AWS integration. That included running initial tests to see if AWS would give us the necessary speed up, setting up the instances for our script to run, and editing the script to work with AWS. Brian focused on the joint-specific tasks of getting the joint variance ranking from those z-scores Umang generated, detecting and displaying the direction of joint correction, the sign of the α value, and running the algorithm in script form that could be called from the front end. He took the lead on the design and implementation of the temporal variant of our procedure, with the frame matching algorithm and updating his correction algorithm to work on videos. Brian and Kristina also worked together to integrate her front end with his back end.

All team members worked together to test integration, the actual correction of joints, and the overall application. All team members, of course, also worked together to create all presentations, reports, and posters about our project.

C. Bill of Materials

KUB is a web application using a Flask Framework with JavaScript, HTML, and CSS. The PyTorch version of AlphaPose, a multi-person pose estimator, is used to detect and track location and movement of joints of a person from a JPG picture or MP4 video and to save the information in JSON format. Algorithms and processing are written in Python. A MacBook Pro 2018, 2.2 Ghz, i7, 16GB RAM, 256GB SSD and its web camera is used to run the web application. Processing of pictures and videos are done on an AWS instance in order to utilize a GPU to optimize AlphaPose's runtime. See Figure 8 for a breakdown of AWS service charges from both testing and demonstration.

D. AWS Usage and Thanks

We would like to thank Amazon for donating AWS credits to us for the development of our project. The credits were spent on accelerating our AlphaPose pose estimation script so that we could return our dance corrections in a reasonable time. We consumed the entirety of the 200 allotted dollars (plus a bit more on accident). The speed up ended up being a very crucial element of the project.

E. Risk Management

For risk management, we had a pretty loose strategy that worked well with our group's communication style and overall approach to the project. Our biggest risk was our own individual commitments taking up time to work and ease to work together. Between extra-curricular activities that take up odd hours of the day and week, research conventions and presentations, many Ph.D visits, and a full course load, we had to communicate, plan, and schedule well in order to get our tasks completed.

Another big risk that would affect performance of the project was the pose estimation processing speed. Once we realized that this was an issue, it became the highest priority task in order to mitigate the issue. We were able to alter our design to incorporate AWS for the desired speed up without too much negative affect on the rest of the project.

Once we realized we had to use AWS, a risk that arose was that we actually had to track budget. In our original design, the only possible cost for the project was an external web camera, but early testing led us to decide that our laptop web camera was sufficient for our use and easier for a user. Once we introduced AWS, however, we had to track usage and cost, making sure to turn off instances when they were unneeded. We knew it would be too easy to over-spend, so it again became an important task to monitor this.

A final risk affected integration, and that was that we had different developmental tools. Two members were using Macbooks whereas one member was using a Windows laptop. Since we knew this from the beginning, we made sure to be aware and familiar of what methods and work-arounds were system-specific so that it wouldn't be as big of an issue during integration.

VIII. Related Work

When we first created our project idea, we looked at other applications on the market that were in a similar field: giving you at home access to a personal trainer in a physical activity. Most of these applications were created to have a database of workouts a user could choose from that fits their desired fitness goals and level of experience. Some of these also tracked users' workouts so a user could see statistics and log their exercise. This was the first "type" of application we found, and was the most abundant. The second "type" is what we accomplished in our project: an application that gave feedback and corrections to the user like an actual personal trainer would. We found only one that accomplished this goal, and it had its own constraints in that the speed at which you perform each movement had to be the same as the on screen visual or else there would be no useful feedback. In our application's specific field of dance however, only a couple applications of

”type” 1 exist and we found none of ”type” 2. So while there are other applications on the market in a related field, there aren’t any that are too similar.

In our capstone class however, there were two other similar projects. One was for weight-training and fitness and one was for yoga. Both offered corrections, but all three were different in implementation and presentation. It was fun to see the difference and similarities in how we all approached a similar problem.

IX. Summary

Overall, our system did meet the design specifications that we set, and our project does work the way we intended. With our budget and time, we of course are limited in the speed of processing; a more expensive machine on AWS and a lot more work on our part would result in faster processing, but for the time frame and our budget, our system runs an average 15 seconds in processing videos, which is already significantly under our 30 second specification. If we were to have more time, it would be effective to speed up our current processing time.

We also limited the number of dance poses and movements that our system supports in order to account for time and access to gather expert trainer data, the amount of time needed to test our correction algorithms on each of the poses, and the time needed to design and implement UI to accommodate many more pose and movement choices. With more time, we could increase the number and difficulty level of poses and movements included in the system to allow for greater variety and to allow more experienced dancers to also benefit from the application.

Another limitation is with our training data, since we were starting from scratch. While the data we do have includes very experienced dancers, they are not professional ballerinas performing all of these ballet moves. With more time, we could gather more data to add to our data set, and with even more time, we could even include professionals in the data. This would add a level of qualification and authority to the corrections and feedback given to the user.

After a semester of working on a new project with new teammates, we have learned a lot of lessons that others could take away from. A project management lesson we learned is that every task takes longer than you think it will, especially since we are students and have a split workload instead of full-time employees that dedicate 40 hours a week to create products. Even if you think or know that one task will take a certain amount of time, budget for more because you never know what will pop up during the time frame allotted for each task. Also, it would be useful to use a form of scheduling that works for the team itself, instead of using the format of schedules that is required from the class. Even though we were warned about it, our team also faced a lot of integration issues. Specifically for us, we found that if we had documented our software dependencies and had done more testing on our own components, integration would have been less problematic. Lastly, know how each teammate operates. All three of us are constantly late, so we learned throughout the semester that we should set meeting times much earlier than when we think we should start working so that we are actually productive in the time we allotted.

References

[1] Zhe Cao, Gines Hidalgo, Tomas Simon, Shih-En Wei, and Yaser Sheikh. Openpose: Realtime multi-person 2d pose estimation using part affinity fields. *CoRR*, abs/1812.08008, 2018.

[2] Yuliang Xiu, Jiefeng Li, Haoyu Wang, Yinghong Fang, and Cewu Lu. Pose flow: Efficient online pose tracking. *CoRR*, abs/1802.00977, 2018.

[3] Bo Pang, Kaiwen Zha, and Cewu Lu. Human action adverb recognition: ADHA dataset and A three-stream hybrid model. *CoRR*, abs/1802.01144, 2018.