Text to Motion Database

Design Document

Brendan Duke
Andrew Kohnen
Udip Patel
David Pitkanen
Jordan Viveiros

January 11, 2017

Contents

Revision History

Date	Version	Notes
January 5, 2017	0.0	File created

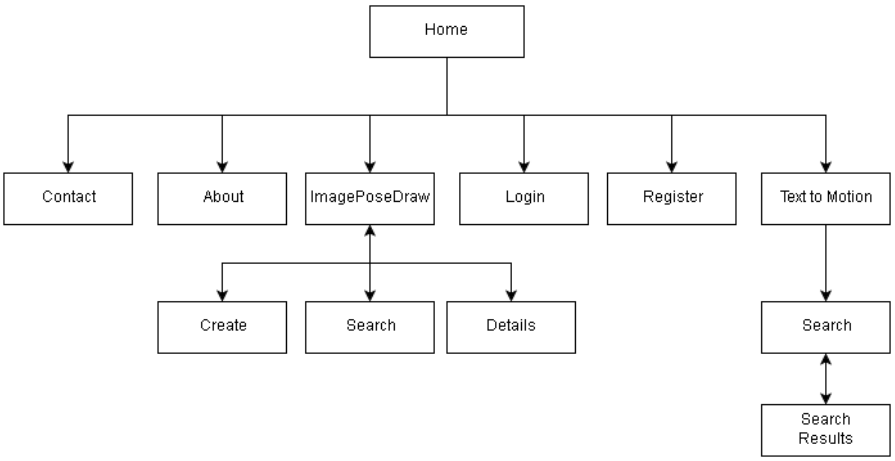
1 Overview

The Text to Motion Database aims to provide a living database of pose estimated media with word pairings and tags. The purpose of this document is to provide a detailed description of the design choices for each section of the Text to Motion Database.

2 User Experience

A user experience is the overall journey of a person on the Text to Motion Database with respect to learnability and usability. This section is organized to describe the journey between web pages and any design choices that went into the user interface in order to improve the overall experience.

Figure 2.1: McMaster Text-to-Motion Database User Expierence



2.1 User Journey

As seen in Figure 2.1, when a user comes to the Text to Motion Database they will see the home page. At first glance they see some information about the website and how it can be used along with some different page options located at the top. Each of these different pages displays something different and is hinted at within the name of the page, with ImagePoseDraw being the most ambiguous. All the pages can be accessed by an anonymous user, but in order to upload media a user must register or sign in. Each page will be described in greater detail below with additional functionality and design choices.

2.2 Home Page

As previously mentioned the first page seen on the Text to Motion Database is the home page. It contains an application description, resources that were used, and brief instructions on how to use the website. The overall design of the page was to be simple

and present the information to the user front and center so that they could explore the options given to them on their own.

2.3 About

The about page is a more granular description of the Text to Motion Databases overview, problem statement and what the intended use of the website is. Like the home page a simple design and colour scheme were chosen along with plain text for easier reading.

2.4 Contact

The contact page maintains the overall look and feel of the website with plain text and individual boxes for the contact information of each group member and supervisors. Every box contains the member's name and e-mail at minimum, with the addition of titles for each supervisor and the department for each group member.

2.5 Navigation Bar

In order to maintain an easy way to navigate the website, the navigation bar located at the top of the page contains links to each page with fixed locations regardless of which page the user is currently on. This allows the users to learn the link location and makes for a more enjoyable experience.

2.6 Log In

If the user has already made an account with the Text to Motion Database they can use the login page to access the account in order to upload images and video. While on the login page before successfully logging in the page location on the navigation bar is in the far right corner, and after logging in it is replaced by the option to log off.
From a design standpoint the login page contains two text boxes for entry, an option for the username to be remembered, a login button, and hyperlinks to register or recover a lost password. Overall it is a very standard login screen and should help a first time user navigate through without any confusion or misunderstanding.

2.7 Register

If the user has not already made an account, and wishes to do, so the option to register can be accessed from the navigation bar or the login page. It follows the same website standards that the login page does and has three text boxes for a username, password and password confirmation along with the button to complete a registration. Once a user has made an account or successfully logged in, the register option in the navigation

bar will be replaced by a greeting and take them to the account management options, which at revision 0 are not fully complete.

2.8 Text To Motion

Searching the database is not restricted by a given user's account and rather can be accessed by anyone, since it is one of the core features of the Text to Motion Database. The ability to search through the pose estimated data is done through the large search bar on the page, helping the user focus on it in order to explain the functionality of the page.

2.8.1 Search Results

Once the search bar has received input it will parse through the database to return uploads that match or have strong resemblance of the input in a column format. Each result will take the user to a separate page with the Name, Description and pose estimated media. The layout of the results shows the user what was returned without any additional information to promote the usability and accuracy of the search.

2.9 Image Pose Draw

Once the user has navigated to the page labeled ImagePoseDraw, the initial view is that of a table with names, descriptions and hyperlinks. This is currently where the most recent uploads are displayed with the name and description that were given during the upload process. The table is designed to contrast consecutive uploads through a dark and light shading in order to easy distinguish two different entries. Beyond the table, the page has two other key functions in the ability to search through the table with by the name or description, and to create new pose estimated uploads through the hyperlink above the table.

2.9.1 Create

If the option to create a new entry has been selected, the user is taken to a new page where they have the ability to upload a new image to be pose estimated and stored within the database. Choosing the image to upload can be done from a URL or internal storage, which opens a file explorer when selected. After an image has been picked, the user has to provide a name and description for the image before it has been pose estimated so that once stored the result can be retrieved from the database using a search option. Upon completion of the above steps the image can be uploaded, taking the user back to the ImagePoseDraw page with the new entry after the processing has happened.

2.9.2 Description

In order to see the uploaded media the user can use the name associated with an image to find it by searching or scrolling through alphabetically. After the upload is located, the user can edit the tags of the image, delete the image and tags, or view the pose estimated media. Deciding to edit the upload takes them to a new screen where the options to change the name or description that were previously input are given. As of revision 0, if the user wants to remove or change their uploaded image they have to first delete the previous entry using the Delete option and go through the steps of creating an upload again. If the user wants to view the selected entry the details option will take them to a new page to view this.

2.9.3 Details

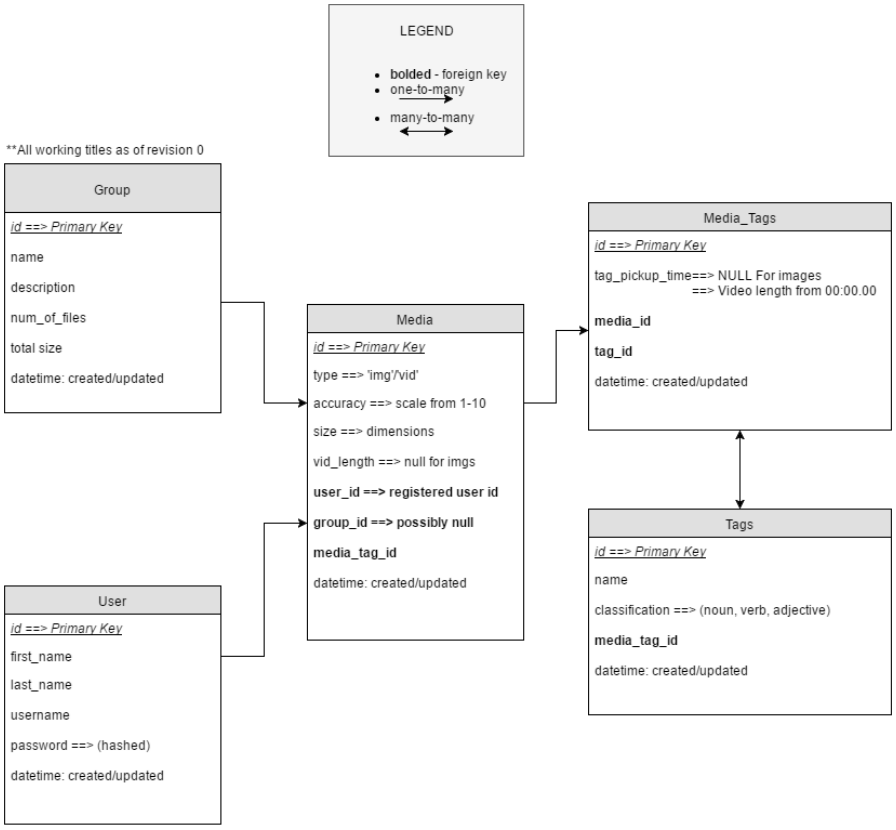
On the details page the user can see the name and description that was chosen at the time of the upload and the image that has been pose estimated to show the chin, and upper arms. Each section of the image is clearly defined with the chin and joints being represented by red circles and the upper arms being represented by two green lines connected at a joint. This shows the user where the algorithm believes the labeled sections are and the separation of colour allows for an easy understanding of the positioning.

3 Database Structure

3.1 Database Schema

The Figure below shows an entity relationship diagram for the current iteration of the database schema. The live website does not use this schema, as the full implementation of the database is still underway.

Figure 3.1: Database Entity-Relationship Diagram



3.1.1 Note For Diagram:

Note that there is an issue with the image above, the diagram has a redundancy. The 'Media' and 'Tags' table should not have a column for "media_tag_id". This column will not be mentioned in the table description and this image will be updated.

3.2 Table Description

3.2.1 Intro

For better logging, every table will contain dateTime columns for 'created_at' and 'updated_at' to store the appropriate time values. in the table descriptions below, these 2 datetime columns will be referred to as **timestamps**

primary keys will be underlined, and foreign keys will be *italicized*

3.2.2 user:

stores user credentials
(id, first_name, last_name, username, password, **timestamps**)

3.2.3 group:

stores information on a group of images/videos that were uploaded together in a group.
(id, name, description, num_of_files, total_size, **timestamps**)

Moving forward, this has the possibility to change to allow for dynamically grouping files in the database

3.2.4 media:

stores information on the actual image uploaded
(id, type = 'image'/'video', accuracy, size, vid_length == null for images, *user_id*, *group_id* == can be set to null, **timestamps**)

3.2.5 tags:

stores any word that is generated by the deep learning algorithm that describes movement or action.
(id, classification = 'noun'/'verb'/'adjective', **timestamps**)

3.2.6 media_tags:

links together media and tags to describe when a tag was picked up in an image/video
(id, tag_pickup_time == NULL for imgs, a time object for videos, *media_id*, *tag_id*, **timestamps**)

4 Module Decomposition

4.1 Text To Motion - ASP.NET Application

4.1.1 Overview

This component of the application is used to run the web interface and is responsible for linking the database and pose estimation functionality. Performing these tasks relies on 'ASP.net' and the Model, View, Controller (MVC) structure. A general description of ASP.NET and its components will be detailed below

ASP.NET core:
framework responsible for handling all http requests to a specific port (Live Website)

.NET Entity Framework
Object-Relational-Mapper (ORM) that can allow for .NET web apps to perform queries and updates on existing databases (or even create new databases with migration files). This is done through using 'Model' files.

4.1.2 Models

Each Model file represents a 'table' in a database. The Model file contains information on the columns of the 'table' and its relation to other tables in the database.
The current version of the live website does not use the relational database schema described in Section 3 of this document. A simpler schema was used for the prototype

Two Models were addded to the .NET web app

- **ApplicationUsers**
  The ApplicationUsers model will be used to add profile data to application, but is currently empty as there are no properties being stored.
  This table gets filled up when users register on the live website
- **PoseDrawnImage**
  PoseDrawnImage uses the get; set; property to store the ID, Description and Name for a given image.
  - int ID
  - string Name
  - string Description

4.1.3 Controllers

In the ASP.NET web application, every function in a 'Controller' file has a corresponding 'View' file (.cshtml) that is associated with in in the 'Views' folder. These files are written in ASP.NET's razor template syntax. Th

A View in the .NET MVC can qualify for the type of **IActionResult**, and is usually returned by the Controller function
Most of the functions inject some text into the View before rendering it. This is just done through the ASP.NET razor markup syntax, which allows for the backend controller to pass information to the view.
Reference to Razor

HTTP[Get] and HTTP[Post] methods can return objects of type **IActionResult** (sync) or **Task<IActionResult>** (async)

4.1.4 HomeController

The HomeController is a simple controller that is used to display the Index, About and Contact pages.

Function: HTTP[Get] Index()

- **Expected Arguments:**
- **Returns:**

  The View of 'Home/Index.cshtml' in the 'Views' folder of the .NET application
- **Description:**

  Calls the View of 'Index' through the MVC, in order to display the home page

Function: HTTP[Get] About()

- **Expected Arguments:**
- **Returns:**

  The View of 'Home/About.cshtml' in the 'Views' folder of the .NET application
- **Description:**

  Calls the View of 'About' through the MVC, in order to display the About page

Function: HTTP[Get] Contact()

- **Expected Arguments:**

- **Returns:**

  The View of 'Home/Contact.cshtml' in the 'Views' folder of the .NET application

- **Description:**

  Calls the View of 'Contact' through the MVC, in order to display the Contact Information

4.1.5 AccountController

The AccountController is used in order to verify Register and Login information for a user, using the HTTP Get and Post. It utilizes built in functions of 'ASP.net' but a majority are not being used for revision 0, so they are omitted below.

Function: HTTP[Get] Login(args)

- **Expected Arguments:**

  string *returnUrl* = null;

- **Returns:**

  The View of 'Account/Login.cshtml' in the 'Views' folder of the .NET application

- **Description:**

  Displays the Login page and stores the ReturnURL to be taken back into *returnUrl*

Function: HTTP[Post] Login(args)

- **Expected Arguments:**

  LoginViewModel *model*;
  string *returnUrl* = null;

- **Returns:**

  Returns the user to the previous page if complete.
  Locks the user out if the number of attempts are exceeded.
  Refreshes the page if something unexpected occurs.

- **Description:**

  Uses the async feature in order to access the account model.Email, model.Password, model.RememberMe and test the login. After which the reponse is returned in any as one of the above situations.

Function: HTTP[Get] Register(args)

- **Expected Arguments:**

  string *returnUrl* = null;

- **Returns:**

  The View of 'Account/Register.cshtml' in the 'Views' folder of the .NET application

- **Description:**

  Displays the Register page and stores the ReturnURL to be taken back into *returnUrl*

Function: HTTP[Post] Register(args)

- **Expected Arguments:**

  RegisterViewModel *model*;
  string *returnUrl* = null;

- **Returns:**

  Upon sucessful registration the user is returned to the previous page.
  If an error occured within the registration process the user is shown the error or the page is refreshed as something unexpected occured.

- **Description:**

  The function creates a new ApplicationUser and stores the email/username and password in model.Email/Username and model.Password respectivly. They are then signed in or shown the errors that may have occured during the account creation.

4.1.6 TextToMotionController

This controller is used to facilitate the 'text-to-motion' search.
As of now, this search functionality has not been implemented due to the lack of a database on the live website. Submitting a search form just passes the search query to the ASP.NET backend and into a new webpage. This will be detailed in the function definitions below

Function: HTTP[Get] Index()

- **Expected Arguments:**

- **Returns:**

  The View of 'TextToMotion/Index.cshtml' in the 'Views' folder of the .NET application

- **Description:**

  Returns the View of 'Index' through the MVC, in order to display the Search page (just a simple view with a text input)

Function:: HTTP[Post] Search(args)

- **Expected Arguments:**

  string *query*

- **Returns:**

  The View of 'TextToMotion/Search.cshtml' in the 'Views' folder of the .NET application with *query* passed into the View

- **Description:**

  Puts the value of *query* into the View. Then, Returns the View of 'Search' through the MVC, which shows the user the search term they entered (This will be built on to actually implement a search functionality)

4.1.7 ImagePoseDrawController

This Controller handles the create/view/edit/delete functionalities for images that users upload
This file also imports a shared object file to access a function defined in C. This function is a call to a C++ program that uses OpenCV and Caffe to analyze a given image and draw a skeleton overlay on the image

Reference to Caffe
Reference to OpenCV

List of Functions (IPD = ImagePoseDraw)

IPD Function 1: Task<bool>DoesImageExist(args)

- **Expected Arguments:**

  int *id*

- **Returns:**

  true if an model of **PoseDrawnImage** with id = the *id* passed into the function exists
  false if no database row found with the given *id*

- **Description:**

  This is just a simple async helper function.
  It references the Entity Model **PoseDrawnImage**

IPD Function 2: ImagePoseDrawController(args)

- **Expected Arguments:**

  ApplicationDbContext *context*
  IHostingEnvironment *environment*

- **Returns:**

  This function is a Constructor for its class and returns an object of type ImagePoseDrawController

- **Description:**

  The Constructor function is used to set the database session context and environment.
  It assigns *context* and *environment* default values determined by global variables. *environment* is used to get the absolute path when saving images

IPD Function 3: HTTP[Get] Index()

- **Expected Arguments:**

- **Returns:**

  The View of 'ImagePoseDraw/Index.cshtml' in the 'Views' folder of the .NET

- **Description:**

  Passes in the list of image names and description into the view. Then, returns the 'Index' View through the MVC, which shows a table of all of the user's uploaded images

  makes a reference to the **PoseDrawnImage** model when it queries all of the names and descriptions of the images that have been captured by the database

IPD Function 4: HTTP[Get] Details()

- int *id*

- **Returns:**

  if *id* is NOT null and a model of type PoseDrawnImage with the given *id* exists, returns the View of 'ImagePoseDraw/Details.cshtml' in the 'Views' folder of the .NET application

  else returns an error obejct

- **Description:**

  Passes in the processed image from the database into the view. Then, returns the 'Details' View, which shows an image with a skeleton overlay on top of the original

picture.
Again, this function also makes a reference to the **PoseDrawnImage** model

IPD Function 5: HTTP[Get] Create()

- **Expected Arguments:**
- **Returns:**

  The View of 'TextToMotion/Create.cshtml' in the 'Views' folder of the .NET application
- **Description:**

  Returns the View of 'Create' through the MVC, which is a form that allows a user to upload an image to run the pose estimation algorithm on

IPD Function 6: HTTP[Post] Create(args)

- **Expected Arguments:**

  PoseDrawnImage *posedImage*
  IFormFile *image* [Reference to IFormFile]
- **Returns:**

  If the process is completed with no errors, returns the View 'ImagePoseDraw/Index.cshtml'

  if any error is caught, returns the View 'ImagePoseDraw/Create.cshtml' so that the user can submit the request again as a form
- **Description:**

  the model **PoseDrawnImage** *posedImage* stores 'metadata' for the image that gets overlaid with a sketch of joint positions. This can be considered metadata because this model stores the location of the image (best case: URL, system-as-is: file path to SQLite DB on the live website server)

  The *image* is the uploaded image that will have a skeleton overlay drawn onto it

  The function inserts a new entry into the pose estimation database table by creating a new row with a unique ID (in ASP.NET, this inclues built-in error checks). The function takes in *image* and saves it to a file path, and the *posedImage* model has an attribute/value that stores the file path that the image is saved in. In order to actually analyze the *image*, the function calls the "estimate_pose_wrapper" function from a shared object file. This shared object file's function (originally written in C++) takes in the raw image, writes the skeleton overlay outline of joints onto the raw image and returns it.

The object referred to in the shared object file will be discussed in detail in the next section.
Once the image has been processed and overwritten to include the pose estimation data (joint positions), *posedImage* is saved to the database

IPD Function 7: HTTP[Get] Edit()

- **Expected Arguments:**
- **Returns:**

  The View of 'ImagePoseDraw/Edit.cshtml' in the 'Views' folder of the .NET application
- **Description:**

  Returns the View of 'Edit' through the MVC, which displays the page where the Name and Description of an upload can be manipulated.

IPD Function 8: HTTP[Post] Edit(args)

- **Expected Arguments:**

  int *id*
  PoseDrawnImage *image*
- **Returns:**

  If the process is completed with no errors, returns the View 'ImagePoseDraw/Index.cshtml'

  else returns back to the Edit page so the user can re-submit the form
- **Description:**

  id refers to the id of the **PoseDrawnImage** model to update
  Takes in text input for the new 'Name' and/or 'Description' via the HTTP Request
  Updates the content of the pose-drawn image database entry (columns for Name and Description).

IPD Function 9: HTTP[Get] Delete()

- **Expected Arguments:**

  int *id*
- **Returns:**

  if the The View of 'ImagePoseDraw/DElet.cshtml' with the upload associated with the *id* removed if confirmation was successful, otherwise returns to the ImagePoseDraw page with the uploads still there.

- **Description:**

  Calls await Details(id) in order to confirm the removing of the upload.

IPD Function 10: HTTP[Post] DeleteConfirmation(args)

- **Expected Arguments:**

  int *id*

- **Returns:**

  If the process is completed with no errors, returns the View 'ImagePoseDraw/Index.cshtml', which will no longer show the upload associated with the *id*

- **Description:**

  id refers to the id of the **PoseDrawnImage** model to delete
  Uses the database _context and **PoseDrawnImages** model in order to remove the image when confirmed. Then redirects to the ImagePoseDraw/Index.cshtml page.

4.2 Flowing Convnets - Human Pose Estimation

4.2.1 Overview

This component of the project actually renders the skeleton overlay onto an image submitted to the website.
Mapping out the joints of a person in an image requires the use of image manipulation and deep learning libraries. As of now, this proces is based on a research paper and is implemented with **Caffe** and **OpenCV** in **C++**. (*The parameters for functions given below will reference 'caffe' and 'cv' types in c++)

4.2.2 Shared Object File

The website is able to take an uploaded image and process it by using a shared object file (.so). The web app can make function calls to functions in the shared object file and pass in images as the parameters.

The C++ file **"estimate_pose.cpp"** contains all of the functions that interface with Caffe and OpenCV. The C file **"estimate_pose_wrapper.c"** is used to wrap the C++ function in C, and create the shared object file so that the C++ function can be accessed from the website.

4.2.3 Pose Estimation C Program (estimate_pose_wrapper.c)

The file **"estimate_pose_wrapper.c"** just contains 1 function. This function references a C++ function in **estimate_pose.cpp**

Function: int32_t estimate_pose_wrapper(args)

- **Expected Arguments:**

  void **image*
  uint32_t **size_bytes*
  uint32_t *max_size_Bytes*

- **Returns:**

  if *image* is processed and saved, returns:
  int32_t size (describing size of file that was uploaded)

  else returns error object

- **Description:**

  this function makes a direct call to the C++ function **"estimate_pose_from_c"** in **"estimate_pose.cpp"** and simply returns the result of that C++ function call. The C++ function takes in the same args as this function

4.2.4 Pose Estimation C++ Program (estimate_pose.cpp)

This C++ Program file contains 8 functions. All of these functions take in objects from the 'openCV'(cv) and 'Caffe'(caffe) libraries as arguments

The key function in this file is **estimate_pose_from_c**, and most of the functions serve as helpers to this function

List of Functions and Descriptions (PE = Pose Estimation):

References for Objects used in C++ functions

- **References for OpenCV objects**

  Reference to cv::Mat
  Reference to cv::Point
  Reference to cv::InputArray

- **References for Caffe Objects**

  Reference to caffe::Blob
  Reference to caffe::Net

- **References for Other Objects**

  Reference to boost::shared_ptr

PE Function 1: void channels_from_blob(args)

- **Expected Arguments:**

  std::vector<cv::Mat> *channels*
  boost::shared_ptr<caffe::Blob> *blob*
  int32_t *width*
  int32_t *height*

- **Returns:**

  void (saves data into *channels*)

- **Description:**

  The *blob* object contains concatenated mulit-channel data
  The *channels* object is empty to begin with
  This function converts the raw data in a Caffe blob into a 'container of channels' (vector of openCV matrices)
  The *width* and *heigth* parameters let the program know what the dimensions of the channels are in the *blob*
  The extracted information from *blobs* is saved into the *channels* vector

This function is just used as a helper for other functions

PE Function 2: void copy_image_to_input_blob(args)

- **Expected Arguments:**

  caffe::Net <float> *heatmap_net*
  cv::Mat *image*

- **Returns:**

  void (saves data into *heatmap_net*)

- **Description:**

  This function converts the *image* object from OpenCV BGR format to 32-bit-floating point RGB format and copies the image to the input blob of *heatmap_net*. It lso divides the input layer of the *heatmap_net* from a multi-channel array into several single-channel arrays by calling a helper function
  image is the image that will serve as the input layer to the caffe network
  heatmap_net is the caffe network that will get its input layer filled with the RGB pixel data from *image*

  This function makes a call to **PE-cpp Function 1** when it splits up the newly updated image in *heatmap_net*'s input layer into several 'input_channels'

PE Function 3: void get_joints_from_network(args)

- **Expected Arguments:**

  cv::Point **joints*
  cv::Size *channel_size*
  caffe::Net<float> *heatmap_net*

- **Returns:**

  void (saves data into *joints*)

- **Description:**

  This function uses the *heatmap_net*'s "conv5_fusion" layer to get a set of joint locations for that heatmap. (This layer is derived from the research paper used)
  The joint locations get saved into **joints*
  The *channel_size* is used to maintain the accuracy of the position of the joints relative to the image as the image matrix is resized multiple times

  This function makes a call to **PE-cpp Function 1** when it uses the *heatmap_net* to save all of the joint locations in a 'joint_channel' vector of cv::Mat objects

PE Function 4: void draw_skeleton(args)

- **Expected Arguments:**

  cv::Mat *image*
  cv::Point **joints*

- **Returns:**

  void (saves image data into *image*)

- **Description:**

  This function uses the joint_locations described in **joints* to draw an upper-body skeleton on the *image* matrix passed in.
  image is the image to draw the skeleton overlay on
  **joints* contain the locations for the set of joints (wrists, elbows, shoulders and head)

PE Function 5:
std::unique_ptr<caffe::Net<float>>init_pose_estimator_network(args)

- **Expected Arguments:**

  std::string *model*
  std::string *trained_weights*

- **Returns:**

  std::unique_ptr<caffe::Net<float>> heatmap_net
  pointer to an object that represents a whole caffe network

- **Description:**

  This function creates a Caffe network and copies over the trained layers from a given option for *trained_weights*
  For this application, a caffe network is initialized with the default settings:
  (*model* = 'MODEL_DEFAULT', *trained_weights* = TRAINED_WEIGHTS_DEFAULT)

PE Function 6: void image_pose_overlay(args)

- **Expected Arguments:**

  caffe::Net<float> *heatmap_net*
  cv::Mat *image*

- **Returns:**

  void (saves to *image*)

- **Description:**

This function processes the *image* passed in using the *heatmap_net* to draw a skeleton on the openCV Matrix

Details on the actions taken by the function:

1. Resizes *image* to 256x256
2. calls **PE Function 2** to copy the image into the input layer of the *heatmap_net*
3. after allowing the network to extract some data, declares an array object of cv::Point called *joints*
4. calls **PE Function 3** to load in joint locations into *joints*
5. converts image to a format so that it can be drawn on by the program
6. calls **PE Function 4** to draw the skeleton overlay on top of the *image*

PE Function 7: void square_image_with_borders(args)

- **Expected Arguments:**

  cv::Mat *image_mat*

- **Returns:**

  void (saves image data to *image_mat*)

- **Description:**

  If the image's dimensions do not fit a square (length != width), this function makes it so that the image dimensions are expanded so that it fits into a square
  This is to avoid distorting the image drastically when the image is resized to 256x256

  This is just a simple helper function to pre-process the image

PE Function 8: int32_t estimate_pose_from_c(args)

- **Expected Arguments:**

  void **image*
  uint32_t **size_bytes*
  unit32_t *max_size_bytes*

- **Returns:**

  if *image* is processed and saved, returns:
  int32_t size (describing size of file that was uploaded)

  else returns error object

- **Description:**

  This function is the key method in this file. This function overwrites the contents of the memory allocated for **image* so that a given image is updated to show the skeleton overlay on that image

  This function creates a new Caffe network and calls a lot of helper functions needed to process the *image*

  Details on the actions taken by the function:

  1. calls **PE Function 5** to create a new caffe Network, stores new network in *heatmap_net*
  2. uses **image* and **size_bytes* to create a cv::InputArray object to represent the uploaded image
  3. creates a cv::Mat image matrix, and decodes the contents of the cv::InputArray object into the newly created matrix object
  4. calls **PE Function 7** to pre-process the image matrix to reaffirm that the image is square
  5. calls **PE Function 6** using *heatmap_net* and the image matrix so that the image_matrix includes the skeleton overlay
  6. converts and compresses the image_matrix into a png
  7. finally, overwrites the contents of **image* with the newly created image that has the pose estimation 'skeleton overlay'

4.3 Features In Development

4.3.1 Pose Estimation For Videos (C++)

Since a video can be considered as just a set (or array) of images, this pose estimation algorithm can be executed on videos as well as images. At this point in the project,a C++ program does exist that references a function from the file **"estimate_pose.cpp"** to analyze videos, but the code has not been finalized. It is also far too slow to be used from a web interface.

Since this functionality is not a part of the system yet, this module was not given a formal declaration/definition. In addition to the reasons mentioned above, the Caffe submodule as a whole will soon be replaced by a newer deep learning framework described in the section below (TensorFlow). So this module is not going to be included in the final revision of this document

4.3.2 TensorFlow

Moving Forward, the Caffe deep learning framework (C++) will be replaced by TensorFlow. TensorFlow is a newer framework with more support for developers compared to Caffe. TensorFlow is a library in Python that can generate dataflow graphs needed for neural networks and deep learning algorithms.

Thanks to the modularity of the project design, replacing the deep learning framework will not drastically change the logic of the web application. The web application would just refer to a different kind of executable when processing an uploaded image (it refers to a shared object file as of now).

Current development of the TensorFlow backend is focused on training a neural network from scratch, or re-training a standard network such as VGG-16. The re-training can use parameters learned by the network on a task with massive datasets available, such as image classification on ImageNet. Certain layers of the network can then be re-trained to solve the task at hand, human pose estimation, taking advantage of the image recognition features learned by the network on the larger dataset.

The focus of the TensorFlow backend is to improve the existing methods by experimenting with state of the art results in human pose estimation using a common framework (that being TensorFlow). Once completed this code could be contributed back to the open source community, so that others can also run the TensorFlow implementation of current human pose estimation research methods.

Optical flow will also be added to the TensorFlow single-frame human pose estimation, in order to extend those estimations to video. An additional optical flow layer in the network will use the context information from adjacent frames to better predict joint positions over time in video, as compared with concatenating the results of making predictions on many individual frames.

4.3.3 Standalone HTTP Server

In addition to the ASP.NET website, the idea is to have a dedicated HTTP server that is separate from the .net website that can take in HTTP requests and run the pose estimation algorithm and save the image/video to the database.

The main purpose of this dedicated server is to optimize mass-uploads of large sets of images/videos. This server would not have to deal with the overhead of running an entire MVC framework, and would be better suited for uploads that take longer. This server could also be configured to be optimized for GPU-based execution of the pose estimation algorithm.
Having a separate site would also prevent the possibility of overloading the main website with too many long-polling requests. This is a definite possibility if there are a lot of users and activity.

There is no module or code for this standalone server yet, but it might be implemented using Python and HTTP Requests

5 Communication Protocol

This application uses the HTTP (HyperText Transfer Protocol) to communicate between the web interface and the server. There is the possibility of using HTTPS for securse communication, but this is not a strong requirement

6 Development Details

6.1 Languages of implementation

- **C#**: ASP.NET Core MVC Framework
- **C++**: Pose Estimation program based on research paper
- **C**: used to wrap the C++ function so that it can be made into a shared object file that can be called from the web app
- **SQLite**: to store database of images

supporting frameworks/plugins:

- Bootstrap
- JQuery JS
- DataTables JS

6.2 Languages of Features in development

- **Python**: Tensorflow
- **MySQL**: for the database schema described in **section 3**

6.3 Software

- **Caffe**: to run pose estimation on heatmaps generated from images
- **OpenCV**: for image manipulation so that uploaded images can be converted into types accepted by Caffe

6.4 Software - for features in development

- **Tensorflow:** to replace Caffe as the deep learning framework
- **Sphinx Search:** indexing tool for SQL databases that can speed up search queries. will be included once the SQL database is up and running

6.5 Hardware

- **Ubuntu Server - 16.0.4:** hosts the live website at this link