Text to Motion Database

Detailed Design

Brendan Duke
Andrew Kohnen
Udip Patel
David Pitkanen
Jordan Viveiros

April 9, 2017

Contents

Revision History

Date	Version	Notes
January 5, 2017	0.0	File created
April 9, 2017	1.0	Final document revision

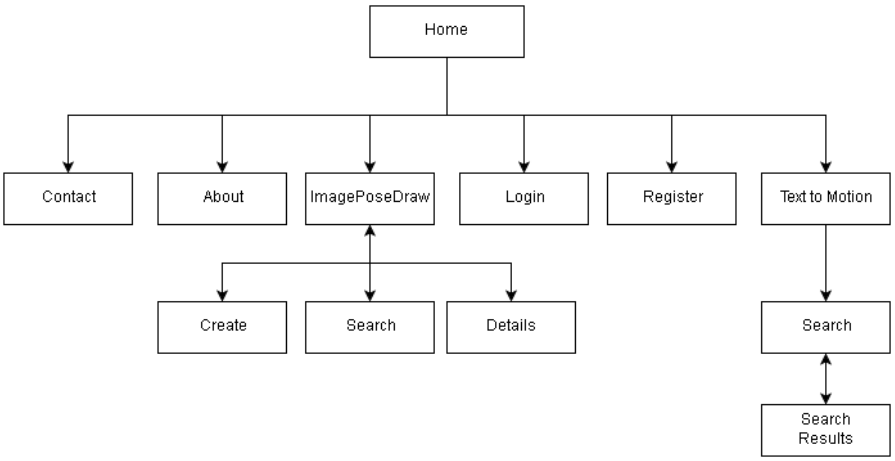
1 Overview

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.

2.10 Media Upload

The process that a verified user can go through in order to upload and interact with the Text-to-Motion Database is enumerated below.

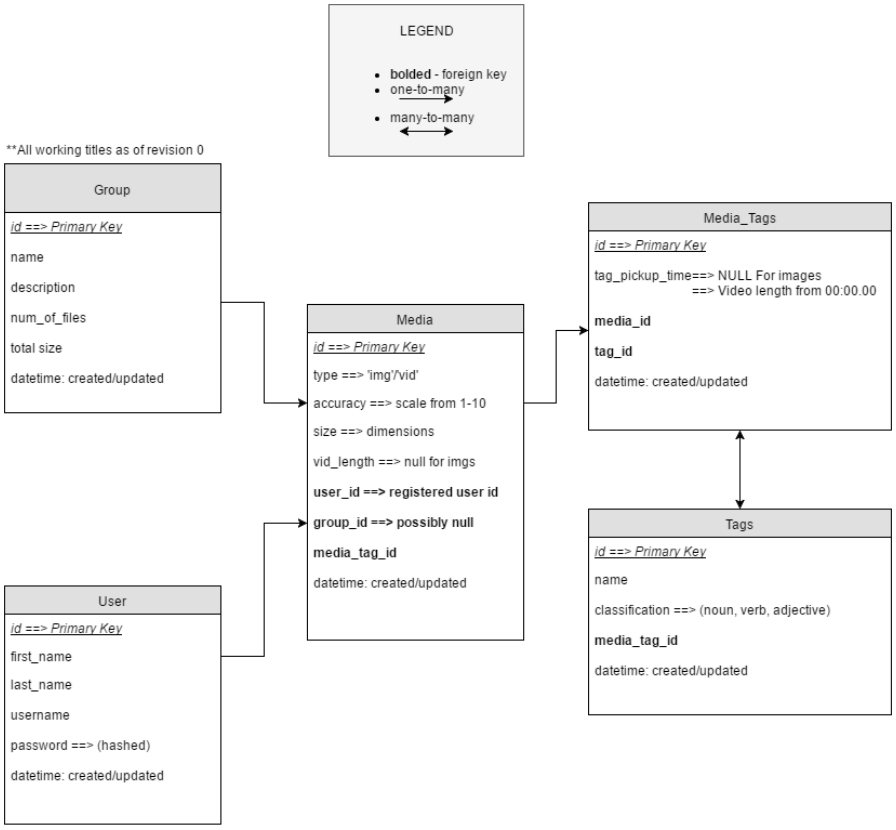
1. The user must first be logged into the website in order to upload an image or video.
2. Using the web interface allows the user multiple options with respect to uploading images and video.
3. The ASP.NET Core web server accesses the TensorFlow backend via HTTP request in order to run pose estimation.
4. The ASP.NET Core web server converts the human pose annotations into the required HDF5 format for storage in the database.
5. The database takes in search queries from the ASP.NET Core web server and receives uploads from the pose estimation process.

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



3.2 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.

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'.

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 imageReference 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 tf-http-server - Human Pose Interface Server

4.2.1 Overview

This component of the project actually provides an HTTP interface through which human pose estimation can be run on images and video. The tf-http-server is written in python and interfaces with the TensorFlow and the human_pose_model directly.

4.2.2 tf_http_server.py

Function: _get_image_joint_predictions(args)

- **Expected Arguments:**

  image: JPEG image in raw bytes format.

  session: TF session to run inference in.

  image_bytes_feed: Placeholder tensor to feed image into.

  logits_tensor: Output logits tensor, corresponding to heatmaps of joint positions inferred by the network.

- **Returns:**

  The joint predictions of a single image in a JSON string.

- **Description:**

  This function does joint position inference on a single image, and returns the resultant joint predictions in a JSON string.

  The returned JSON string has the following format:

```
[{"r_ankle": [0.5, 0.5], "r\_knee": [1.0, 2.0], ...},
 {"r_ankle": [0.25, 0.2], "r\_knee": [0.1, 0.5], ...}].
```

I.e. it is a JSON array of dictionaries, where the keys are names of joints from `JOINT_NAMES_NO_SPACE`, and the values are two-element arrays containing the `[x, y]` coordinates of the joint prediction.

These `[x, y]` coordinates are in a space where the range `[-0.5, 0.5]` represent the range, in the padded image, from the far left to the far right in the case of x, and from the top to the bottom in the case of y.

Function: TFHttpRequesthandlerFactory(args)

- **Expected Arguments:**

  session

  image_bytes_feed

  logits_tensor

  resized_image_tensor

- **Returns:**

  This function returns a subclass of the `http.server.BaseHTTPrequestHandler`.

- **Description:**

  This function is required to call the subclass of the `http.server.BaseHTTPrequestHandler` as it allows the function to call extra parameters which are local to the class. All of this is required as the constructor expects a certain function signature.

Function: _respond_with_joints(args)

- **Expected Arguments:**

  self

  image

- **Returns:**

  Returns a 200 OK HTTP response message with response data as a JSON string containing the inferred joint position.

- **Description:**

  Takes the image and performs joint inference on it in order to store it within the JSON that is returned containing the inferred joint position.

Function: do_GET(args)

- **Expected Arguments:**

  self

- **Returns:**

  A JSON string of the estimated joints for the image URL that has been passed as an option parameter.

- **Description:**

  This representation of an HTTP GET request will take an image URL as a JPEG and perform joint inference on the image in order to save it in as a JSON string that will later be returned.

Function: do_POST(args)

- **Expected Arguments:**

  self

- **Returns:**

  A JSON string that represents the inferred joint position.

- **Description:**

  This function is a HTTP POST request containing a JPEG in base 64 in order to decode the base 64 image and perform joint inference in order to store it as a JSON string representing the joint position.

Function: _get_joint_position_inference_graph(args)

- **Expected Arguments:**

  image_bytes_feed

  batch_size

- **Returns:**

  A constructed computation graph that will be used to run human pose inference on.

- **Description:**

  The function sets up the computation graph that will decode a JPEG from the input *image_bytes_feed* in order to pad and resize the image to the desired shape. Which is then used to run the human pose inference on it using the "Two VGG-16s cascade" model.

Function: run()

- **Expected Arguments:**

- **Returns:**

- **Description:**

  Starts the server in order to handle the HTTP request by listening on an SSL-wrapped socket at port 8765 of the localhost. On start-up a joint inference computation graph is setup, all model weights are restored and a session where the graph can be run is initialized.

4.3 human_pose_model - Training and Inference Software Suite

4.3.1 Overview

The human_pose_model module provides a completely standalone software suite for the training, evaluation and deployment of human pose inference models. In order to provide this software suite the human_pose_model is set up in a hierachy that utilizes `train.py`, `evaluate.py`, `write_tf_record.py`, `input_pipeline.py`, `networks.py`, `dataset.py`, and `pose_utils.py`.

4.3.2 input_pipeline.py

Sets up an input pipeline that reads example protobufs from all TFRecord files, decodes and preprocesses the images. The input pipeline is setup with a sequence of queues so that preprocessing and file-reading can be parallelized across many hardware threads.

Function: _setup_example_queue(args)

- **Expected Arguments:**

  filename_queue

  num_readers

  input_queue_memory_factor

  batch_size

- **Returns:**

  A dequeue operation that will dequeue one Tensor containing an input example from `examples_queue`.

- **Description:**

  Sets up a randomly shuffled queue containing example protobufs, read from the TFRecord files in `filename_queue`.

Function: _parse_example_proto(args)

- **Expected Arguments:**

  example_serialized

  image_dim

- **Returns:**

  Returns a tuple containing a raw image reshaped to the image dimensions in float32 format, sparse indices, and sparse joints.

- **Description:**

  Parses an example proto in order to return the tuple above.

Function: _distort_colour(args)

- **Expected Arguments:**

  distorted_image

  thread_id

- **Returns:**

  The image after being distorted.

- **Description:**

  Distorts the brightness, saturation, hue and contrast of an image randomly using the *thread_id*.

Function: _decode_binary_maps(args)

- **Expected Arguments:**

  binary_maps

  image_dim

- **Returns:**

  The input with a reshaped tensor so that TensorFlow can utilize the input.

- **Description:**

  Decodes a binary map from its post-decompression format and reshapes the tensor.

Function: _flip_with_left_right_permutation(args)

- **Expected Arguments:**

  maps

- **Returns:**

  A flipped map of the input.

- **Description:**

  Flips the ground truth maps, accounting for the fact that when mirrored the images left and right will already have been swapped.

Function: _maybe_flip_maps(args)

- **Expected Arguments:**

  maps

  should_flip

- **Returns:**

  The input map that may be flipped.

- **Description:**

  Conditionally flip ground truth maps left-right.

Function: _randomly_flip(args)

- **Expected Arguments:**

  image

  binary_maps

  heatmaps

- **Returns:**

  The (possibly) flipped map.

- **Description:**

  Randomly flips the input and set of joint-maps left or right.

Function: _randomly_rotate(args)

- **Expected Arguments:**

  image

  binary_maps

  heatmaps

  max_rotation_angle

- **Returns:**

  The randomly rotated input.

- **Description:**

  Randomly rotates inputs between +/- of the "max_rotation_angle".

Function: _distort_image(args)

- **Expected Arguments:**

  decoded_image

  binary_maps

  heatmaps

  image_dim

  thread_id

  max_rotation_angle

- **Returns:**

  A tuple containing joint-maps and image post slipping, rotation, and colour distortion.

- **Description:**

  Randomly distorts the image from 'parse_example' by randomly rotating, randomly flipping left and right, and randomly distorting the colour of the image.

Function: _parse_and_preprocess_example_eval(args)

- **Expected Arguments:**

  heatmap_stddev_pixels

  example_serialized

  num_preprocessed_threads

  image_dim

- **Returns:**

  The x and y joints.

- **Description:**

  Works as `_parse_and_preprocess_example_train` without the image distortion or heatmap creation.

Function: _get_joints_normal_pdf(args)

- **Expected Arguments:**

  dense_joints

  std_dev

  coords

  expand_axis

- **Returns:**

  A set of 1-D Normal distributions.

- **Description:**

  Creates the set of 1-D Normal distributions with means equal to the elements of `dense_joints` and deviations equal to the value of `std_dev`.

Function: _get_joint_heatmaps(args)

- **Expected Arguments:**

  heatmap_stddev_pixels

  image_dim

  x_dense_joints

  y_dense_joints

- **Returns:**

  Joint heatmaps.

- **Description:**

  Calculates a set of confidence maps for the joints given by `x_dense_joints` and `y_dense_joints`.

Function: _get_is_visible_weights(args)

- **Expected Arguments:**

  parse_joint_indices

  is_visible_list

  weights

- **Returns:**

  A set of per-joint weights.

- **Description:**

  Calculates a set of per-joint weights, which are 1 if the joint annotation is both and present and unoccluded.

Function: _maybe_flip_weights(args)

- **Expected Arguments:**

  weights

  should_flip

- **Returns:**

  Weight joints that were conditionally flipped.

- **Description:**

  Conditionally permutes weights left-right.

Function: _parse_and_preprocess_example_train(args)

- **Expected Arguments:**

  example_serialized

  num_preprocess_threads

  image_dim

  heatmap_stddev_pixels

  max_rotation_angle

- **Returns:**

  A list of lists, one for each thread, where each inner list contains decoded image with colours scaled to range [-1,1] as well as the sprase joint ground truth vectors.

- **Description:**

  Parses Example protobufs containing input images and their ground truth vectors and preprocesses those images, returning a vector with one preprocessed tensor per thread.

Function: _setup_batch_queue(args)

- **Expected Arguments:**

  images_and_joint_maps

  batch_size

  num_preprocess_threads

- **Returns:**

  A batch queue of images, binary_maps, heatmaps, and weights.

- **Description:**

  Sets up a batch queue that returns.

Function: setup_eval_input_pipeline(args)

- **Expected Arguments:**

  batch_size

  num_preprocess_threads

  image_dim

  heatmap_stddev_pixels

  data_filenames

- **Returns:**

  The input pipeline to be used for model evaluation.

- **Description:**

  Sets up an input pipeline for model evaluation.

Function: setup_train_input_pipeline(args)

- **Expected Arguments:**

  FLAGS

  data_filename

- **Returns:**

  `(images, heatmaps, weights, batch_size)`: List of image tensors with first dimension (`shape[0]`) equal to `batch_size`, along with lists of dense vectors of heatmaps, ground truth vectors (`joints`), dense vectors and the batch size.

- **Description:**

  Sets up an input pipeline that reads example protobufs from all TFRecord files, assumed to be named `train*.tfrecord` (e.g. `train0.tfrecord`), decodes and preprocesses the images.

  There are three queues: `filename_queue`, contains the TFRecord filenames, and feeds `examples_queue` with serialized example protobufs. Then, serialized examples are dequeued from `examples_queue`, preprocessed in parallel by `num_preprocess_threads` and the result is enqueued into the queue created by `batch_join`. A dequeue operation from the `batch_join` queue is what is returned from `preprocess_images`. What is dequeued is a batch of size `batch_size` containing a set of, for example, 32 images in the case of `images` or a sparse vector of floating-point joint co-ordinates (in range `[0,1]`) in the case of `joints`.

4.3.3 train

This sub-module of `human_pose_model` does human pose inference model training.

Function: train()

- **Expected Arguments:** None.
- **Returns:** None.
- **Description:** Trains a human pose estimation network to detect and/or regress binary maps and/or confidence maps of joint co-ordinates (`NUM_JOINTS` sets of (x, y) co-ordinates).

  Here we only take files $0-N$. For now files are manually renamed as `train[0-N].tfrecord` and `valid[N-M].tfrecord`, where there are $N+1$ train records, $(M-N)$ validation records and M + 1 records in total.

4.3.4 networks

The `networks` sub-module contains a sub-module called `inference`, which contains all of the human pose inference model imports, various loss functions as well as a function to do inference. Besides `inference`, the `networks` sub-module also contains model definitions themselves (e.g. `vgg_bulat` for the VGG-16 model).

Function: inference()

- **Expected Arguments:**

  images: Mini-batch of preprocessed examples dequeued from the input pipeline.

  heatmaps: Confidence maps of ground truth joints.

  weights: Weights of heatmaps (tensors of all 1s if joint present, all 0s if not present).

  is_visible_weights: Weights of heatmaps/binary maps, with occluded joints zero'ed out.

  gpu_index: Index of GPU calculating the current loss.

  scope: Name scope for ops, which is different for each tower (tower_N).

- **Returns:** Tensor giving the total loss (combined loss from auxiliary and primary logits, added to regularization losses).
- **Description:** Sets up a human pose inference model, computes predictions on input images and calculates loss on those predictions based on an input dense vector of joint location confidence maps and binary maps (the ground truth vector).

  TF-slim's `arg_scope` is used to keep variables (`slim.model_variable`) in CPU memory. See the training procedure block diagram in the TF Inception README.

4.3.5 dataset

The dataset contains the `mpii_read` sub-module, which is a module for reading the MPII Human Pose Dataset.

Function: mpii_read()

- **Expected Arguments:**

  mpii_dataset_filepath: The filepath to the .mat file provided from the MPII Human Pose website.

  is_train: Read training data (True), or test data (False)?

- **Returns:** Parsed `MpiiDataset` object from `parse_mpii_data_from_mat`.

- **Description:** Reads the MPII dataset from its provided `.mat` format.

  Note that the images are assumed to reside in a folder one up from the .mat file that is being parsed (i.e. ../images).

parse_mpii_data_from_mat

- **Expected Arguments:**

  mpii_dataset_mat: A dictionary of MATLAB structures loaded using `scipy.io.loadmat`. The arguments `struct_as_record = False` and `squeeze_me = True` must be set in the `loadmat` call.

  mpii_images_dir: The path of the directory where all the MPII images are stored.

  is_train: Parse training data (True), or test data (False)?

- **Returns:** Returns: An `MpiiDataset` Python object corresponding to `mpii_dataset_mat`.

- **Description:** Parses the training data out of `mpii_dataset_mat` into a `MpiiDataset` Python object.

  To save time during debugging sessions, you can manually get `mpii_dataset_mat` using `scipy.io.loadmat` once, and then iteratively call this function as you make changes, without reloading the .mat file.

4.4 Features In Development

4.4.1 TensorFlow

The Caffe deep learning framework (C++) has already been 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 has not drastically changed the logic of the web application. The web application refers to

a different kind of executable when processing an uploaded image (it previously referred to a shared object).

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.

4.4.2 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 server, which is configured to be optimized for GPU-based execution of the pose estimation algorithm, is to off-load human pose inference computations from the MVC web server.

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.

5 Communication Protocol

This application uses the HTTPS (HyperText Transfer Protocol Secure) to communicate between the web interface and the server. HTTP over SSL is used due to browser requirements of HTTPS in order to use the webcam.

The browser's webcam is needed for development of the human pose estimation app, which is a simple demo of the Text-to-Motion human pose estimation technology as suggested by Dr. He. The web app demo is currently under development.

6 Development Details

6.1 Languages of implementation

- **C#**: ASP.NET Core MVC Framework
- **Python**: 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

- **Tensorflow:** to replace Caffe as the deep learning framework
- **OpenCV**: for image manipulation so that uploaded images can be converted into types accepted by Caffe

6.4 Software - for features in development

- **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 https://brendanduke.ca.

7 Reference Links

7.1 ASP.NET Reference List

Reference to ASP.NET
- Reference to IFormFile
- Reference to Razor

7.2 Caffe Reference List

Reference to Caffe
- Reference to caffe::Blob
- Reference to caffe::Net

this function is related to caffe so was included in this section:
- Reference to boost::shared_ptr

7.3 OpenCV Reference List

Reference to OpenCV
- Reference to cv::InputArray
- Reference to cv::Mat
- Reference to cv::Point