Facemap

Release 1.0.2

Carsen Stringer & Atika Syeda

Feb 12, 2024

BASICS:

Facemap is a framework for predicting neural activity from mouse orofacial movements. It includes a pose estimation model for tracking distinct keypoints on the mouse face, a neural network model for predicting neural activity using the pose estimates, and also can be used compute the singular value decomposition (SVD) of behavioral videos.

For more details, please see our paper and twitter thread.

CHAPTER

ONE

INSTALLATION

Please see the Github readme for the latest installation instructions

1.1 Common installation issues

- If you have `pip` issues, there might be some interaction between pre-installed dependencies and the required Facemap dependencies. Please upgrade pip as follows:

```
python -m pip install --upgrade pip
```

- While running `python -m facemap`, if you receive the error: `No module named PyQt5.sip`, then try uninstalling and reinstalling pyqt5 as follows:

```
pip uninstall pyqt5 pyqt5-tools
pip install pyqt5 pyqt5-tools pyqt5.sip
```

- If you are on Yosemite Mac OS, PyQt doesn't work, and you won't be able to install Facemap. More recent versions of Mac OS are supported. The software has been heavily tested on Ubuntu 18.04, and less well tested on Windows 10 and Mac OS. Please post an issue if you have installation problems.

1.2 Dependencies

Facemap (python package) relies on these awesome packages:

- pyqtgraph
- pyqt6
- numpy (>=1.13.0)
- scipy
- opencv
- numba
- natsort
- pytorch
- matplotlib
- tqdm
- UMAP

1.3 MATLAB package installation

The matlab version supports SVD processing only and does not include the orofacial tracker. The package can be downloaded/cloned from github (no install required). It works in Matlab 2014b and above. The Image Processing Toolbox is necessary to use the GUI. For GPU functionality, the Parallel Processing Toolbox is required. If you don't have the Parallel Processing Toolbox, uncheck the box next to "use GPU" in the GUI before processing. Note this version is no longer supported.

CHAPTER

TWO

GUI

2.1 Starting the GUI

The quickest way to start is to open the GUI from a command line terminal:

```
python -m facemap
```

2.2 Command line usage

Run `python -m facemap --help` to see usage of the following optional parameters:

-h, --help	show this help message and exit
--ops	options
--movie	Absolute path to video(s)
--keypoints	Absolute path to keypoints file (*.h5)
--proc_npy	Absolute path to proc file (*_proc.npy)
--neural_activity	Absolute path to neural activity file (*.npy)
--neural_prediction	Absolute path to neural prediction file (*.npy)
--tneural	Absolute path to neural timestamps file (*.npy)
--tbehavior	Absolute path to behavior timestamps file (*.npy)
--savedir	save directory
--autoload_keypoints	Automatically load keypoints in the same directory as the movie
--autoload_proc	Automatically load *_proc.npy in the same directory as the movie

2.3 Using the GUI

The GUI can be used for the processing keypoints and SVD of mouse behavioral videos. The GUI can also be used for predicting neural activity using the behavioral data. For more details on each feature, see the following tutorials:

2.3.1 Pose tracking (GUI)

The latest python version is integrated with Facemap network for tracking 14 distinct keypoints on mouse face and an additional point for tracking paw. The keypoints can be tracked from different camera views (see examples).

Generate keypoints

Follow the steps below to generate keypoints for your videos:

1. Load video
    - Select `File` from the menu bar
    - For processing single video, select `Load video`. Alternatively, for processing multiple videos, select `Load multiple videos` to select the folder containing the videos. (Note: Pose estimation for multipl videos is only supported for videos recorded simultaneously i.e. have the same time duration and frame rate).

   (Optional) Set output folder
    - Use the file menu to `Set output folder`.
    - The processed keypoints (`*.h5`) and metadata (`*.pkl`) will be saved in the selected output folder or folder containing the video (by default).
2. Process video(s)
    - Check `Keypoints` for pose tracking.
    - Click `process`.
    - Note: The first time facemap runs for processing keypoints it downloads the latest available trained model weights from our website.
3. Set ROI/bounding box for face region
    - A dialog box for selecting a bounding box for the face will appear. Drag the red rectangle to select region of interest on the frame where the keypoints will be tracked. Please ensure that the bouding box is focused on the face where all the keypoints will be visible. See example frames here. If a 'Face (pose)' ROI has already been added then this step will be skipped.
    - Click `Done` to process video. Alternatively, click `Skip` to use the entire frame region. Monitor progress bar at the bottom of the window for updates.
4. View keypoints
    - Keypoints will be automatically loaded after processing.
    - Processed keypoints file will be saved as `[videoname]_FacemapPose.h5` in the selected output folder.

Visualize keypoints

To load keypoints (*.h5) for a video generated using Facemap or other software in the same format (such as DeepLabCut and SLEAP), follow the steps below:

1. Load video
    - Select `File` from the menu bar
    - Select `Load video`
2. Load keypoints
    - Select `Pose` from the menu bar
    - Select `Load keypoints`
    - Select the keypoints (*.h5) file
3. View keypoints
    - Use the "Keypoints" checkbox to toggle the visibility of keypoints.
    - Change value of "Threshold (%)" under pose settings to filter keypoints with lower confidence estimates. Higher threshold will show keypoints with higher confidence estimates.

Finetune model to refine keypoints for a video

To improve keypoints predictions for a video, follow the steps below:

1. Load video
    - Select `File` from the menu bar
    - Select `Load video`
2. Set finetuned model's output folder
    - Select `Pose` from the menu bar
    - Select `Finetune model`
    - Set output folder path for finetuned model
3. Select training data and set training parameters
    - Set `Initial model` to use for training. By default, Facemap's base model trained on our dataset will be used for fine-tuning. Alternatively, you can select a model previously finetuned on your own dataset.
    - Set `Output model name` for the finetuned model.
    - Choose `Yes/No` to refine keypoints prediction for the video loaded and set `# Frames` to use for training. You can also choose proportion of random vs. outlier frames to use for training. The outlier frames are selected using the `Difficulty threshold (percentile)`, which determines the percentile of confidence scores to use as the threshold for selecting frames with the highest error.
    - Choose `Yes/No` to add previously refined keypoints to the training set.
    - `Set training parameters` or use default values.
    - Click `Next`
4. Refine keypoints
    - If a ROI/bounding box was not added, then a dialog box for selecting a bounding box for the face will appear. Drag the red rectangle to select region of interest on the frame where the keypoints will be tracked.

- Click `Done` to process video. Alternatively, click `Skip` to use the entire frame region. Monitor progress bar at the bottom of the window for updates.
- Drag keypoints to refine predictions. Use `Shift+D` to delete a keypoint. Right click to add a deleted keypoint. Use `Previous` and `Next` buttons to change frame. Click `Help` for more details.
- Click `Train model` to start training. A progress bar will appear for training updates.

5. Evaluate training
   - View predicted keypoints for test frames from the video loaded. For further refinement, Click `Continue training` that will repeat steps 3-5.
   - Click `Save model` to save the finetuned model. The finetuned model will be saved as `*.pt` in the selected output folder.
6. Generate keypoints using the finetuned model
   - Use the `Pose model` dropdown menu to set the finetuned model to use for generating keypoints predictions.
   - (Optional) Change "Batch size" under pose settings.
   - Click `Process` to generate keypoints predictions. See *Generate keypoints* for more details.

2.3.2 SVD processing and ROIs

Choose a type of ROI to add and then click "add ROI" to add it to the view. The pixels in the ROI will show up in the right window (with different processing depending on the ROI type - see below). You can move it and resize the ROI anytime. You can delete the ROI with "right-click" and selecting "remove". You can change the saturation of the ROI with the upper right saturation bar. You can also just click on the ROI at any time to see what it looks like in the right view.

By default, the "multivideo" box is unchecked. If you check it, then the motion SVD or movie SVD is computed across ALL videos - all videos are concatenated at each timepoint, and the SVD of this matrix of ALL_PIXELS x timepoints is computed. If you have just one video acquired at a time, then it is the SVD of the full video.

To compute motion SVD and/or movie SVD, please check one or both boxes in the GUI before hitting process.

If you want to open the GUI with a movie file specified and/or save path specified, the following command will allow this: ~~~ python -m facemap –movie '/home/carsen/movie.avi' –savedir '/media/carsen/SSD/' ~~~ Note this will only work if you only have one file that you need to load (can't have multiple in series / multiple views).

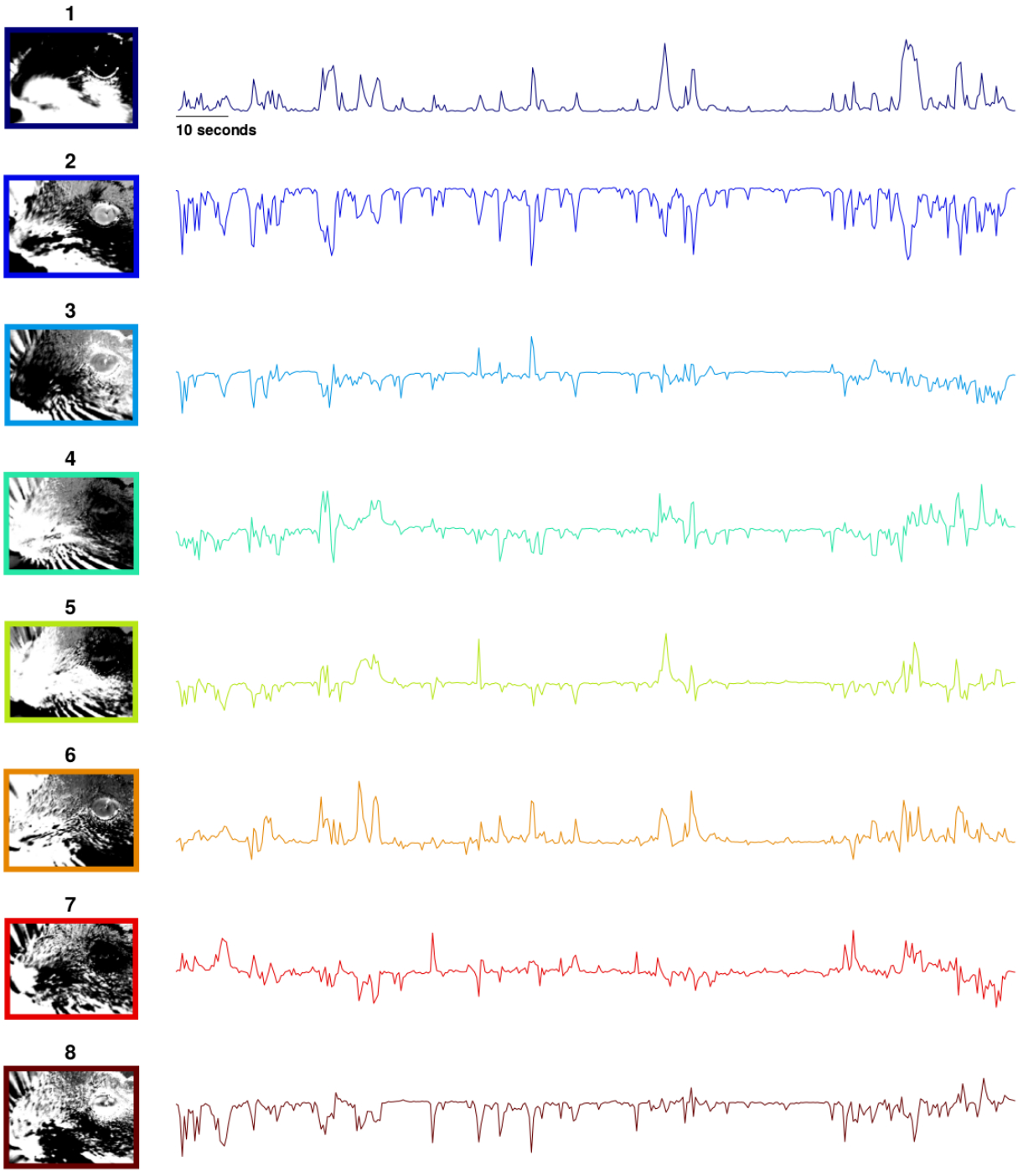
ROI types

Motion SVD

The motion/movie SVDs (small ROIs / multivideo) are computed on the movie downsampled in space by the spatial downsampling input box in the GUI (default 4 pixels). Note the saturation set in this window is NOT used for any processing.

The motion M is defined as the abs(current_frame - previous_frame), and the average motion energy across frames is computed using a subset of frames (*avgmot*) (at least 1000 frames). Then the singular vectors of the motion energy are computed on chunks of data, also from a subset of frames (15 chunks of 1000 frames each): *uMotMask*. These are the motion masks that are then projected onto the video at all timepoints (done in chunks of size nt=500):

Example motion masks *uMotMask* and traces *motSVD*:


1
10 seconds
2
3
4
5
6
7
8

The SVDs can be computed on the motion or on the raw movie, please check the corresponding box for "motion SVD" and/or "movie SVD" before hitting process to compute one or both of these.

We found that these extracted singular vectors explained up to half of the total explainable variance in neural activity in visual cortex and in other forebrain areas. See our paper for more details.

We also compute the average of M across all pixels in each motion ROI and that is returned as the **motion**. The first **motion** field is non-empty if "multivideo SVD" is on, and in that case it is the average motion energy across all pixels in all views.

Pupil computation

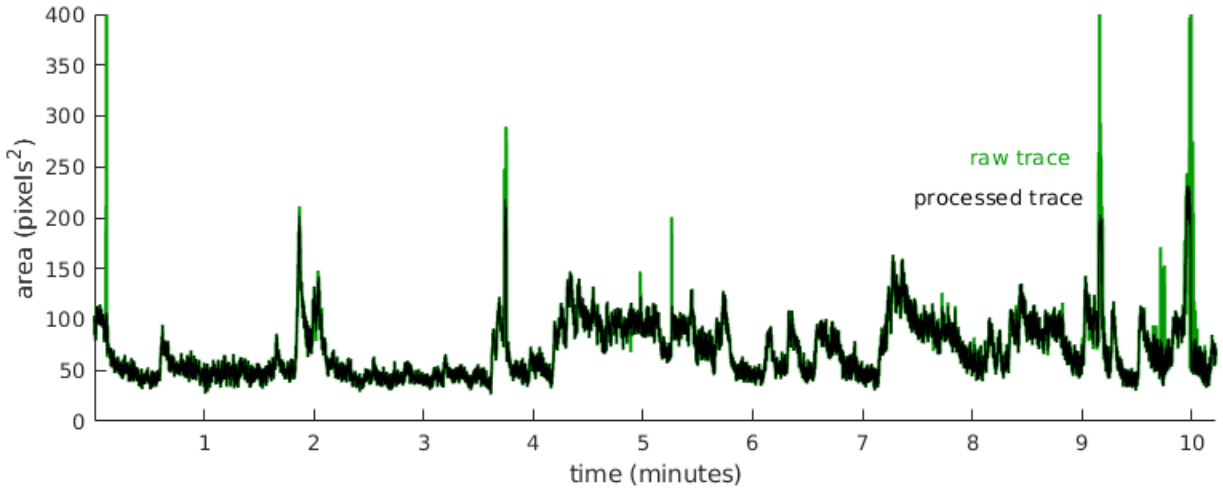
The minimum pixel value is subtracted from the ROI. Use the saturation bar to reduce the background of the eye. The algorithm zeros out any pixels less than the saturation level (I recommend a *very* low value - so most pixels are white in the GUI).

Next it finds the pixel with the largest magnitude. It draws a box around that area (1/2 the size of the ROI) and then finds the center-of-mass of that region. It then centers the box on that area. It fits a multivariate gaussian to the pixels in the box using maximum likelihood (see pupil.py).

After a Gaussian is fit, it zeros out pixels whose squared distance from the center (normalized by the standard deviation of the Gaussian fit) is greater than 2 * sigma^2 where sigma is set by the user in the GUI (default sigma = 2.5). It now performs the fit again with these points erased, and repeats this process 4 more times. The pupil is then defined as an ellipse sigma standard deviations away from the center-of-mass of the gaussian. This is plotted with '+' around the ellipse and with one '+' at the center.

If there are reflections on the mouse's eye, then you can draw ellipses to account for this "corneal reflection" (plotted in black). You can add as many of these per pupil ROI as needed. The algorithm fills in these areas of the image with the predicted values, which allows for smooth transitions between big and small pupils.

This raw pupil area trace is post-processed. The trace is median filtered with a window of 30 timeframes. At each timepoint, the difference between the raw trace and the median filtered trace is computed. If the difference at a given point exceeds half the standard deviation of the raw trace, then the raw value is replaced by the median filtered value.



Blink computation

You may want to ignore frames in which the animal is blinking if you are looking at pupil size. The blink area is defined the number of pixels above the saturation level that you set (all non-white pixels).

Running computation

The phase-correlation between consecutive frames (in running ROI) are computed in the fourier domain (see running.py). The XY position of maximal correlation gives the amount of shift between the two consecutive frames. Depending on how fast the movement is frame-to-frame you may want at least a 50x50 pixel ROI to compute this.

2.3.3 Neural activity prediction

This tutorial shows how to use the deep neural network encoding model to predict neural activity based on mouse orofacial behavior.

To process neural activity prediction using pose estimates extracted using the keypoint tracker:

1. Load or process keypoints (see pose tracking tutorial).
2. Select `Neural activity` from file menu.
3. Click on `Launch neural activity window`.
4. Select `Load neural activity` (2D-array stored in *.npy) and (optionally) timestamps for neural and behavioral data (1D-arrays stored in*.npy) then click `Done`.
5. Once the neural data is loaded, click on `Run neural predictions`.
6. Select `Keypoints` as input data and select one of the options for output of the model's prediction, which can be `Neural PCs` or neural activity. CLick on `Help` button for more information.
7. The predicted neural activity (*.npy) file will be saved in the selected output folder.

To predict neural activity using SVDs from Facemap:

1. Load or process SVDs for the video. (see SVD tutorial).
2. Follow steps 2-5 above.

Note: a linear model is used for prediction using SVDs.

Predicted neural activity will be plotted in the neural activity window. Toggle `Highlight test data` to highlight time segments not used for training i.e. test data. Further information about neural prediction, including variance explained can be found in the saved neural prediction file (*.npy).

CHAPTER

THREE

INPUTS

Facemap supports grayscale and RGB movies. The software can process multi-camera videos for pose tracking and SVD analysis. Movie file extensions supported include:

'.mj2','.mp4','.mkv','.avi','.mpeg','.mpg','.asf'

Here are some example movies.

3.1 Processing multiple movies

Please note:

- simultaneous videos: user can load videos of different dimensions but must have the number of frames for each video
- sequential videos: user can load videos with varying number of frames but the dimensions of the frames/videos must match

If you load multiple videos, the GUI will ask *"are you processing multiple videos taken simultaneously?"*. If you say yes, then the script will look if across movies the **FIRST FOUR** letters of the filename vary. If the first four letters of two movies are the same, then the GUI assumed that they were acquired *sequentially* not *simultaneously*.

Example file list: + cam1_G7c1_1.avi + cam1_G7c1_2.avi + cam2_G7c1_1.avi + cam2_G7c1_2.avi + cam3_G7c1_1.avi + cam3_G7c1_2.avi

"are you processing multiple videos taken simultaneously?" ANSWER: Yes

Then the GUIs assume {cam1_G7c1_1.avi, cam2_G7c1_1.avi, cam3_G7c1_1.avi} were acquired simultaneously and {cam1_G7c1_2.avi, cam2_G7c1_2.avi, cam3_G7c1_2.avi} were acquired simultaneously. They will be processed in alphabetical order (1 before 2) and the results from the videos will be concatenated in time. If one of these files was missing, then the GUI will error and you will have to choose file folders again. Also you will get errors if the files acquired at the same time aren't the same frame length (e.g. {cam1_G7c1_1.avi, cam2_G7c1_1.avi, cam3_G7c1_1.avi} should all have the same number of frames).

Note: if you have many simultaneous videos / overall pixels (e.g. 2000 x 2000) you will need around 32GB of RAM to compute the full SVD motion masks.

You will be able to see all the videos that were simultaneously collected at once. However, you can only draw ROIs that are within ONE video. Only the "multivideo SVD" is computed over all videos.

3.2 Batch processing

Load a video or a set of videos and draw your ROIs and choose your processing settings. Then click "save ROIs". This will save a **_proc.npy* file in the output folder. Default output folder is the same folder as the video. Use file menu to change path of the output folder. The name of saved proc file will be listed below "process batch" (this button will also activate). You can then repeat this process: load the video(s), draw ROIs, choose settings, and click "save ROIs". Then to process all the listed **_proc.npy* files click "process batch".

3.3 Data acquisition info

3.3.1 IR illumination

For recording in darkness we use IR illumination at 850nm, which works well with 2p imaging at 970nm and even 920nm. Depending on your needs, you might want to choose a different wavelength, which changes all the filters below as well. 950nm works just as well, and probably so does 750nm, which still outside of the visible range for rodents.

If you want to focus the illumination on the mouse eye or face, you will need a different, more expensive system. Here is an example, courtesy of Michael Krumin from the Carandini lab: driver, power supply, LED, lens, and lens tube, and another lens tube.

3.3.2 Cameras

We use ptgrey cameras. The software we use for simultaneous acquisition from multiple cameras is BIAS software. A basic lens that works for zoomed out views here. To see the pupil well you might need a better zoom lens 10x here.

For 2p imaging, you'll need a tighter filter around 850nm so you don't see the laser shining through the mouse's eye/head, for example this. Depending on your lenses you'll need to figure out the right adapter(s) for such a filter. For our 10x lens above, you might need all of these: adapter1, adapter2, adapter3, adapter4.

CHAPTER

FOUR

OUTPUTS

4.1 ROI and SVD processing

SVD processing saves two outputs: a *.npy file and a *.mat file. The output file contains the following variables:

- **filenames**: A 2D list of video filenames - a list within the 2D list consists of videos recorded simultaneously whereas sequential videos are stored as a separate list
- **Ly**, **Lx**: list of frame length in y-dim (Ly) and x-dim (Lx) for each video taken simultaneously
- **sbin**: spatial bin size for SVDs
- **Lybin**, **Lxbin**: list of number of pixels binned by sbin in Y (Ly) and X (Lx) for each video taken simultaneously
- **sybin**, **sxbin**: coordinates of multivideo (for plotting/reshaping ONLY)
- **LYbin**, **LXbin**: full-size of all videos embedded in rectangle (binned)
- **fullSVD**: bool flag indicating whether "multivideo SVD" is computed
- **save_mat**: bool flag indicating whether to save proc as *.*mat* file
- **avgframe**: list of average frames for each video from a subset of frames (binned by sbin)
- **avgframe_reshape**: average frame reshaped to size y-pixels by x-pixels
- **avgmotion**: list of average motion computed for each video from a subset of frames (binned by sbin)
- **avgmotion_reshape**: average motion reshaped to size y-pixels by x-pixels
- **iframes**: an array containing the number of frames in each consecutive video
- **motion**: list of absolute motion energies across time - first is "multivideo" motion energy (empty if not computed)
- **motSVD**: list of motion SVDs - first is "multivideo SVD" (empty if not computed) - each is of size number of frames by number of components (500)
- **motMask**: list of motion masks for each motion SVD - each motMask is pixels x components
- **motMask_reshape**: motion masks reshaped to: y-pixels x x-pixels x components
- **motSv**: array containing singular values for motSVD
- **movSv**: array containing singular values for movSVD
- **pupil**: list of pupil ROI outputs - each is a dict with 'area', 'area_smooth', and 'com' (center-of-mass)
- **blink**: list of blink ROI outputs - each is nframes, the blink area on each frame
- **running**: list of running ROI outputs - each is nframes x 2, for X and Y motion on each frame
- **rois**: ROIs that were drawn and computed:

- rind: type of ROI in number
- rtype: what type of ROI ('motion SVD', 'pupil', 'blink', 'running')
- ivid: in which video is the ROI
- color: color of ROI
- yrange: y indices of ROI
- xrange: x indices of ROI
- saturation: saturation of ROI (0-255)
- pupil_sigma: number of stddevs used to compute pupil radius (for pupil ROIs)
- yrange_bin: binned indices in y (if motion SVD)
- xrange_bin: binned indices in x (if motion SVD)

4.1.1 Loading outputs

The *.npy saved is a dict which can be loaded in python as follows:

```
import numpy as np
proc = np.load('filename_proc.npy', allow_pickle=True).item()
print(proc.keys())
motion = proc['motion']
```

These *_proc.npy* files can be loaded in the GUI (and is automatically loaded after processing). The checkboxes on the lower left panel of the GUI can be used to toggle display of different traces/variables.

4.2 Keypoints processing

Keypoints processing saves two outputs: a *.h5 and a *metadata.pkl file.

- *.h5 file contains: Keypoints stored as a 3D array of shape (3, number of bodyparts, number of frames). The first dimension of size 3 is in the order: (x, y, likelihood). For more details on using/loading the *.h5 file in python see this tutorial.
- ***metadata.pkl file: contains a dictionary consisting of the following variables:**
  - batch_size: batch size used for inference
  - image_size: frame size
  - bbox: bounding box for cropping the video [x1, x2, y1, y2]
  - total_frames: number of frames
  - bodyparts: names of bodyparts
  - inference_speed: processing speed

  To load the pkl file in python, use the following code:

```
import pickle
with open('filename_metadata.pkl', 'rb') as f:
    metadata = pickle.load(f)
```

(continues on next page)

(continued from previous page)

```
print(metadata.keys())
print(metadata['bodyparts'])
```

4.3 Neural activity prediction output

The output of neural activity prediction is saved in *.npy file and optionally in *.mat file. The output contains a dictionary with the following keys:

- predictions: a 2D array containing the predicted neural activity of shape (number of features x time)
- test_indices: a list of indices indicating sections of data used as test data for computing variance explained by the model
- variance_explained: variance explained by the model for test data
- plot_extent: extent of the plot used for plotting the predicted neural activity in the order [x1, y1, x2, y2]

CHAPTER

FIVE

TUTORIALS

See example tutorials for processing keypoints and refining keypoints model.

CHAPTER

SIX

FACEMAP API GUIDE

6.1 Pose estimation

class facemap.pose.pose.**Pose**(*filenames=None*, *bbox=[]*, *bbox_set=False*, *resize=False*, *add_padding=False*, *gui=None*, *GUIobject=None*, *net=None*, *model_name=None*)

Pose estimation for single video processing

Parameters

- **filenames** (*2D-list*) – List of filenames to be processed.
- **bbox** (*list*) – Bounding box for cropping the video [x1, x2, y1, y2]. If not set, the entire frame is used.
- **bbox_set** (*bool*) – Flag to indicate whether the bounding box has been set. Default is False.
- **resize** (*bool*) – Flag to indicate whether the video needs to be resized
- **add_padding** (*bool*) – Flag to indicate whether the video needs to be padded. Default is False.
- **gui** (*object*) – GUI object.
- **GUIobject** (*object*) – GUI mainwindow object.
- **net** (*object*) – PyTorch model object.
- **model_name** (*str*) – Name of the model to be used for pose estimation. Default is None which uses the pre-trained model.

load_model()

Load model for keypoints prediction. Uses default model unless set_model is used to update the model name and load the model state.

predict_landmarks(*video_id*, *frame_ind=None*)

Predict keypoints/landmarks for all frames in video and save output as .h5 file. If frame_ind is specified, only predict keypoints/landmarks for those frames. :param video_id: Index of video in filenames list to be used for prediction. :type video_id: int :param frame_ind: List of frame indices for keypoints/landmarks prediction. :type frame_ind: list

save_data_to_hdf5(*data*, *video_id*, *selected_frame_ind=None*)

save_data_to_hdf5: Save data to an HDF5 file

Parameters

- **data** (*2D-array*) – Data to save (nframes x nbodyparts x 3)
- **selected_frame_ind** (*list*) – Indices of selected frames

save_dict_to_hdf5(*h5file*, *path*, *data_dict*)

Saves dictionary to an HDF5 file. Adapted from https://github.com/talmolab/sleap/blob/391bc0421fe3820ddd6b5d07e31311d60b129fe3/sleap/util.py#L116 Calls itself recursively if items in dictionary are not *np.ndarray*, *np.int64*, *np.float64*, *str*, or bytes. Objects must be iterable. :param h5file: The HDF5 filename object to save the data to.

Assume it is open.

Parameters

- **path** – The path to group save the dict under.
- **data_dict** – The dict containing data to save.

Raises
ValueError – If type for item in dict cannot be saved.

Returns
None

save_model(*model_filepath*)

Save model to file :param model_filepath: Path to save model weights :type model_filepath: str

set_model(*model_selected=None*)

Set model to use for pose estimation

Parameters
model_selected (*str, optional*) – Path of trained model weights to use. Default value of None sets the model_name to Base model/facemap_model_state which uses pre-trained weights.

train(*image_data*, *keypoints_data*, *num_epochs*, *batch_size*, *learning_rate*, *weight_decay*, *bbox*)

Train the model :param image_data: Array of images of shape (nframes, Ly, Lx) :type image_data: ND-array :param keypoints_data: Array of keypoints of shape (nframes, nkeypoints, 2) :type keypoints_data: ND-array :param num_epochs: Number of epochs for training :type num_epochs: int :param batch_size: Batch size for training :type batch_size: int :param learning_rate: Learning rate for training :type learning_rate: float :param weight_decay: Weight decay for training :type weight_decay: float

Returns
model – Trained/finetuned model

Return type
torch.nn.Module

6.2 SVD processing

facemap.process.**process_blink_ROIs**(*t*, *nt0*, *img*, *ivid*, *rois*, *blind*, *blinks*)

docstring

facemap.process.**process_pupil_ROIs**(*t*, *nt1*, *img*, *ivid*, *rois*, *pupind*, *pups*, *pupreflector*)

docstring

facemap.process.**process_running**(*t*, *n*, *nt1*, *img*, *ivid*, *rois*, *runind*, *runs*, *rend*)

docstring

facemap.process.**run**(*filenames*, *sbin=1*, *motSVD=True*, *movSVD=False*, *GUIobject=None*, *parent=None*, *proc=None*, *savepath=None*)

Process video files using SVD computation of motion and/or raw movie data. :param filenames: List of video files to process. Each element of the list is a list of

filenames for video(s) recorded simultaneously. For example, if two videos were recorded simultaneously, the list would be: [['video1.avi', 'video2.avi']], and if the videos were recorded sequentially, the list would be: [['video1.avi'], ['video2.avi']].

Parameters

- **sbin** (*int*) – Spatial binning factor. If sbin > 1, the movie will be spatially binned by a factor of sbin.
- **motSVD** (*bool*) – If True, compute SVD of motion in the video i.e. the difference between consecutive frames.
- **movSVD** (*bool*) – If True, compute SVD of raw movie data.
- **GUIobject** (*GUI object*) – GUI object to update progress bar. If None, no progress bar will be updated.
- **parent** (*GUI object*) – Parent GUI object to update progress bar. If None, no progress bar will be updated.
- **proc** (*dict*) – Dictionary containing previously processed data. If provided, parameters from the saved data, such as sbin, rois, sy, sx, etc. will be used.
- **savepath** (*str*) – Path to save processed data. If None, the processed data will be saved in the same directory as the first video file.

Returns
savename – Path to saved processed data.

Return type
str

6.3 Transforms

facemap.pose.transforms.**adjust_bbox**(*prev_bbox*, *img_yx*, *div=16*, *extra=1*)

Takes a bounding box as an input and the original image size. Adjusts bounding box to be square instead of a rectangle. Uses longest dimension of prev_bbox for final image size that cannot exceed img_yx :param prev_bbox: bounding box positions in order x1, x2, y1, y2 :type prev_bbox: tuple of size (4,) :param img_yx: image size for y and x dimensions :type img_yx: tuple of size (2,)

Returns
bbox – bounding box positions in order x1, x2, y1, y2

Return type
tuple of size (4,)

facemap.pose.transforms.**adjust_keypoints**(*xlabels*, *ylabels*, *crop_xy*, *padding*, *current_size*, *desired_size*)

Adjust raw keypoints (x,y coordinates) obtained from model to plot on original image :param xlabels: x coordinates of keypoints :type xlabels: ND-array :param ylabels: y coordinates of keypoints :type ylabels: ND-array :param crop_xy: initial coordinates of bounding box (x1,y1) for cropping :type crop_xy: tuple of size (2,) :param padding: padding values for bounding box (x1,x2,y1,y2) :type padding: tuple of size (4,)

Returns

- **xlabels** (*ND-array*) – x coordinates of keypoints
- **ylabels** (*ND-array*) – y coordinates of keypoints

facemap.pose.transforms.**adjust_keypoints_for_padding**(*xlabels*, *ylabels*, *pads*)

Adjust keypoints for padding. Adds padding to the top and left of the image only. :param xlabels: x coordinates of keypoints :type xlabels: ND-array :param ylabels: y coordinates of keypoints :type ylabels: ND-array :param pads: padding values for bounding box (y1,y2,x1,x2) :type pads: tuple of size (4,)

Returns

- **xlabels** (*ND-array*) – x coordinates of keypoints after padding
- **ylabels** (*ND-array*) – y coordinates of keypoints after padding

facemap.pose.transforms.**augment_data**(*image*, *keypoints*, *scale=False*, *scale_range=0.5*, *rotation=False*, *rotation_range=10*, *flip=True*, *contrast_adjust=True*)

Augments data by randomly scaling, rotating, flipping, and adjusting contrast :param image: image of size nchan x Ly x Lx :type image: ND-array :param keypoints: keypoints of size nkeypoints x 2 :type keypoints: ND-array :param scale: whether to scale the image :type scale: bool :param scale_range: range of scaling factor :type scale_range: float :param rotation: whether to rotate the image :type rotation: bool :param rotation_range: range of rotation angle :type rotation_range: float :param flip: whether to flip the image horizontally :type flip: bool :param contrast_adjust: whether to adjust contrast of image :type contrast_adjust: bool

Returns

- **image** (*ND-array*) – image of size nchan x Ly x Lx
- **keypoints** (*ND-array*) – keypoints of size nkeypoints x 2

facemap.pose.transforms.**crop_image**(*im*, *bbox=None*)

Crop image to bounding box. :param im: image of size [(Lz) x Ly x Lx] :type im: ND-array :param bbox: bounding box positions in order x1, x2, y1, y2 :type bbox: tuple of size (4,)

Returns
im – cropped image of size [1 x Ly x Lx]

Return type
ND-array

facemap.pose.transforms.**get_crop_resize_params**(*img*, *x_dims*, *y_dims*, *xy=(256, 256)*)

Get cropped and resized image dimensions Input:-

img: image x_dims: min,max x pos y_dims: min,max y pos xy: final (desired) image size

Output:-
x1: (int) x dim start pos x2: (int) x dim stop pos y1: (int) y dim start pos y2: (int) y dim stop pos resize: (bool) whether to resize image

facemap.pose.transforms.**get_cropped_imgs**(*imgs*, *bbox*)

Preproccesing of image involves: conversion to float32 in range0-1, normalize99, and padding image size to be compatible with UNet model input :param imgs: images of size [batch_size x nchan x Ly x Lx] :type imgs: ND-array :param bbox: bounding box positions in order x1, x2, y1, y2 :type bbox: tuple of size (4,)

Returns
cropped_imgs – images of size [batch_size x nchan x Ly' x Lx'] where Ly' = y2-y1 and Lx'=x2-x1

Return type
ND-array

facemap.pose.transforms.**get_random_factor**(*factor_range*)

Get a random factor within the range provided. :param factor_range: factor range in order min, max :type factor_range: tuple of size (2,)

Returns
factor – random factor

Return type
float

facemap.pose.transforms.**pad_img_to_square**(*img*, *bbox=None*)

Pad image to square. :param im: image of size [c x h x w] :type im: ND-array :param bbox: bounding box positions in order x1, x2, y1, y2 used for cropping image :type bbox: tuple of size (4,)

Returns

- **im** (*ND-array*) – padded image of size [c x h x w]
- **(pad_w, pad_h)** (*tuple of int*) – padding values for width and height

facemap.pose.transforms.**pad_keypoints**(*keypoints*, *pad_h*, *pad_w*)

Pad keypoints using padding values for width and height. :param keypoints: keypoints of size [N x 2] :type keypoints: ND-array :param pad_h: height padding :type pad_h: int :param pad_w: width padding :type pad_w: int

Returns
keypoints – padded keypoints of size [N x 2]

Return type
ND-array

facemap.pose.transforms.**preprocess_img**(*im*, *bbox*, *add_padding*, *resize*, *device=None*)

Preproccesing of image involves:

1. Conversion to float32 and normalize99
2. Cropping image to select bounding box (bbox) region
3. padding image size to be square
4. Resize image to 256x256 for model input

Parameters

bbox: tuple of size (4,)
bounding box positions in order x1, x2, y1, y2

add_padding: bool
whether to add padding to image

resize: bool

whether to resize image

Returns

postpad_shape: tuple of size (2,)
shape of padded image

pads: tuple of size (4,)
padding values for (pad_y_top, pad_y_bottom, pad_x_left, pad_x_right)

facemap.pose.transforms.**randomize_bbox_coordinates**(*bbox*, *im_shape*, *random_factor_range=(0.1, 0.3)*)

Randomize bounding box by a random amount to increase/expand the bounding box region while staying within the image region. :param bbox: bounding box positions in order x1, x2, y1, y2 :type bbox: tuple of size (4,) :param im_shape: image shape in order Ly, Lx :type im_shape: tuple of size (2,) :param random_factor_range: range of random factor to use for expaning bounding box :type random_factor_range: tuple of size (2,)

Returns
bbox – randomized bounding box positions in order x1, x2, y1, y2

Return type
tuple of size (4,)

facemap.pose.transforms.**rescale_keypoints**(*xlabels*, *ylabels*, *current_size*, *desired_size*)

Rescale keypoints to original image size :param xlabels: x coordinates of keypoints :type xlabels: ND-array :param ylabels: y coordinates of keypoints :type ylabels: ND-array :param current_size: current size of image (h,w) :type current_size: tuple of size (2,) :param desired_size: desired size of image (h,w) :type desired_size: tuple of size (2,)

Returns

- **xlabels** (*ND-array*) – x coordinates of keypoints
- **ylabels** (*ND-array*) – y coordinates of keypoints

facemap.pose.transforms.**resize_image**(*im*, *resize_shape*)

Resize image to given height and width. :param im: image of size [Ly x Lx] :type im: ND-array :param resize_shape: desired shape of image :type resize_shape: tuple of size (2,)

Returns
im – resized image of size [h x w]

Return type
ND-array

facemap.pose.transforms.**resize_keypoints**(*keypoints*, *desired_shape*, *original_shape*)

Resize keypoints to desired shape. :param keypoints: keypoints of size [batch x N x 2] :type keypoints: ND-array :param desired_shape: desired shape of image :type desired_shape: tuple of size (2,) :param original_shape: original shape of image :type original_shape: tuple of size (2,)

Returns
keypoints – keypoints of size [batch x N x 2]

Return type
ND-array

PYTHON MODULE INDEX

f

INDEX