Real-time detection and movement prediction of behaving *Pogona Vitticeps*

Itai Turkenich Tal Eisenberg

Mark Shein-Idelson lab

A module for an experimental system built by MA student Reggev Eyal

Submitted to: Amir Globerson

BA project for the Sagol School of Neuroscience, Computer Science - Psychology track

October 2020

Introduction

Many animals rely on hunting live prey for their survival. Tracking and catching prey requires predicting the future location of the observed prey, and using this information to plan and execute hunting behavior. By investigating this type of behavior, insights regarding the more general question of how external moving stimuli are represented and modelled in the brain, as well as the plasticity of these models, can be revealed.

In a recent experiment, Yoo et al. investigated these issues with macaque monkeys. The monkeys were trained to play a computer game, where they tried to catch a virtual “prey” with a “self” avatar, that the monkeys controlled using a joystick. The “prey” avatar was programmed to escape using an algorithm that maximizes its distance from the monkey controlled avatar (Yoo et al., 2020).

The following report describes an attempt to implement a similar escape mechanism for a virtual prey displayed on a screen. The animal used is a Central Bearded Dragon (*Pogona vitticeps*), and thus the escape mechanism we implemented uses images recorded at real time. The experiment was designed and built by MA student Reggev Eyal, and our project involved building one module of the experimental system that allows it to react to the animal’s behavior in real-time. Apart from this module, we had no role in designing the experimental paradigm and system, or building other parts of the experimental system.

Our module is composed of two separate parts - detection and prediction, therefore the following paper is also separated into two main parts, each describing related works, our chosen solution, various considerations and constraints, and experimental results.

Experimental Setup and System Overview

The experimental setup consists of a small arena equipped with a touch screen, where the animal can behave freely. Four cameras record grayscale images of the animal in the arena from different viewing angles at a rate of 60 frames per second (fps). In a typical experiment, a virtual prey is displayed moving on the screen, causing the animal to engage in hunting behavior, and try to “catch” the virtual prey by touching the screen with its tongue. The touch event is then registered by the screen and recorded.

A computer system running custom software modules is responsible for collecting data from the cameras, controlling the images displayed on the screen, and rewarding the animal with food

upon successful hunts. In order to analyze and predict the trajectory of the animal's movement, the program sends the video frames from one of the cameras to our real-time prediction module. Based on the images, the real-time module tries to predict where and when the animal is going to touch the screen, and sends the prediction back. According to the prediction, the movement of the virtual prey can be adjusted so that it will try to escape its hunter. In terms of latency, our goal was to produce a prediction in less than 16ms, so that the system can keep up with the maximal acquisition speed of the camera, and to make sure the prey can change its course as soon as possible.

Our implementation detects the location of the animal head in the image, and uses the location of the head over time to predict the future motion of the head, and use that information to predict future screen touch events.

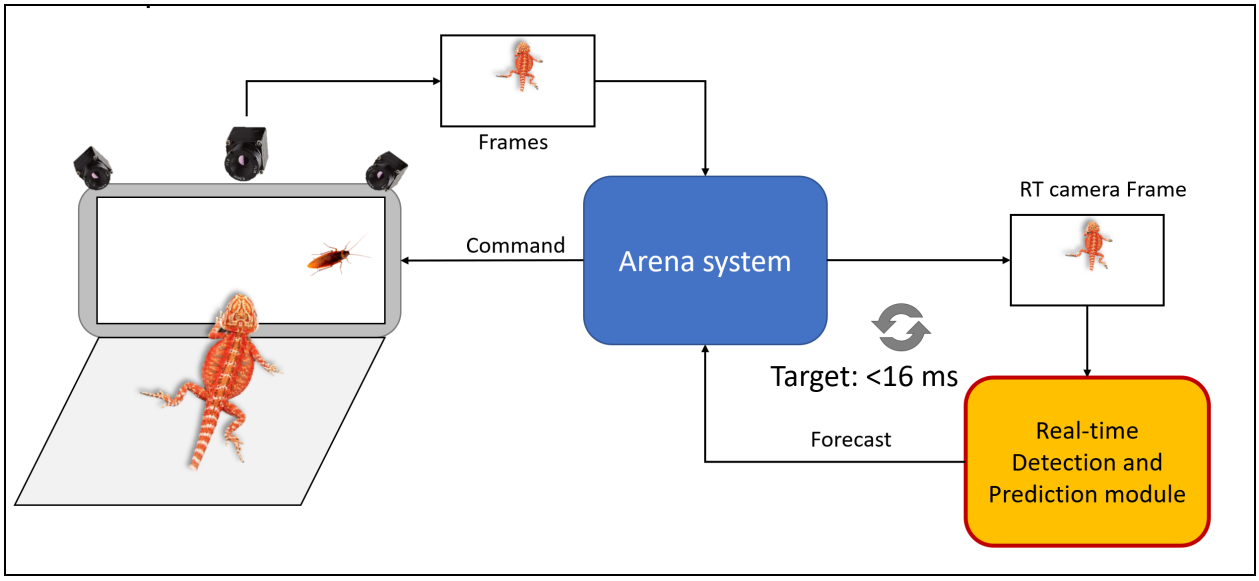

Figure 1: The experimental system and the role of the real-time module

Related Works and Possible Solutions

Detection

This section details some deep learning based tools from the computer vision literature, that can be used to identify the location of the animal, without any physical markers. As in many tasks in computer vision, the algorithms rely on Convolutional Neural Networks (CNNs), though their inner workings and output differ. We aimed for an "off the shelf" detector that could be integrated

relatively easily into our code, as detection is a common problem that arises in many applications in different domains.

Supervised object detection: This class of algorithms detect instances of predefined objects in an image with an axes aligned bounding box. Training these networks requires manually labelling images containing the required classes. There are many types of object detection algorithms, and the main tradeoff is between inference latency and accuracy. Recent object detectors can operate robustly in real time (>30 fps inference, single image batch), for example, YOLO-v4 (Bochkovskiy et al., 2020) and EfficientNet (Tan & Le, 2020).

"Single shot" object tracking: Another relevant class of algorithms track instances of arbitrary objects with "single shot" learning. The tracked object is labelled by a bounding box (manually or by other means) in the first frame of a stream, and the network tracks its position, relying on the similarity between consecutive frames. Example for a recent such tracker is SiamMask (Chen & Tsotsos, 2019). These algorithms' main advantage is that manual labelling of a dataset for each custom object is not required. On the downside, these algorithms are prone to "drift" of the detection, which is problematic for tracking an object reliably for a long duration.

Supervised pose estimation: Lastly, another relevant class of algorithms estimates the location of predefined joints of some object, mainly of behaving humans and animals. These algorithms' output is rich and complex, which is more useful than a bounding box for some use cases. Furthermore, using multiple viewpoints, a 3D pose of the animal can be produced. On the downside, manually labelling the data is more complex, and networks that perform this task reliably in real time were not available at the time of research. Recently developed systems such as DeepLabCut-Live (Kane et al., 2020) and EthoLoop (Nourizonoz et al., 2020) claim to have made animal pose estimation possible in real time.

Prediction

In contrast to the detection part, predicting screen touching by the animal is a problem more specific to the experimental setting used in the lab, and to the unique properties of the motion exhibited by the animal. Below we briefly review some related literature, though it's worth noting that modelling and predicting movement in a high spatio-temporal resolution in real-time is a relatively new task, as appropriate and widely available computational tools and hardware have only begun to emerge in recent years.

Human 3D pose prediction: Algorithms from this class predict the next poses of a 3D human skeleton in a sequence, where the input is past 3D poses. The networks can be a recurrent neural

network (RNN; Martinez et al., 2017) or, more recently, a Transformer network (Aksan et al., 2020). This area of the literature seemed the most relevant for our problem, as it involves predicting the motion of several joints. We didn't find similar literature regarding animal pose prediction.

Pedestrian trajectory prediction: Another class of algorithms that solve a somewhat similar problem are networks that predict the trajectories of (possibly multiple) pedestrians. The input is a sequence of 2D coordinates, and the output is a sequence of 2D coordinates (Becker et al., 2019). Like 3D pose prediction, these networks mostly rely on RNNs. They significantly differ from our problem, as the movement itself is slower, sampled at a lower rate, and there's an emphasis on predicting the trajectories of multiple pedestrians together, taking into account the relations between different agents, which is a complication that is irrelevant for our problem.

Modelling and predicting animal behaviour and movement: As noted, not much literature exists regarding modeling and predicting movements of animals at the spatio-temporal resolution that we were interested in. Some literature exists regarding modelling more general animal behaviour, such as migratory patterns of birds or movements of colonies of ants (see review in Wijeyakulasuriya et al., 2020).

Overview of the approach

Detection

We chose to use the YOLO-v4 network in it's C-based Darknet implementation to detect the head of the animal. This implementation claimed to be the fastest out of all available object detectors at the time of research, which was the main factor for choosing it (more technical details regarding the detector and it's integration will follow). Other factors were the robustness of the detection - the bounding box output is relatively simple, and a preliminary test we conducted with a small dataset of the animal's head yielded good results. We chose to use this framework, and not PyTorch or TensorFlow, as the implementations in those frameworks are significantly slower.

As to single-shot object tracking networks, their reported inference times (particularly of SiamMask and its predecessors) were slower than those of YOLO-v4. Additionally, the fact that the first frame of each trial needs to be hand labelled makes the use of the system more cumbersome. More importantly, preliminary tests conducted with the SiamMask algorithm revealed a "drift" phenomenon, as the bounding box slowly deviated from the head of the animal to other parts of its body, or to random objects in the arena. In contrast, the YOLO-v4 network detects the head in every frame independently of previous frames, which is more robust.

Finally, we chose not to use pose estimation, mainly because the inference times of the popular DeepLabCut system (Mathis et al., 2018) were irrelevant for real time inference at the time of research. Detecting the exact structure of the animal's head (by detecting the nose and both ears, for example), would be more useful for analyzing the head's movement, as it will make it possible to infer not only the location, but also the angle of the head. This is a main downside of the axis aligned bounding box, as it introduces various geometrical artifacts into the motion data. For example, a rotation of the head in its place can not be accurately detected.

Prediction

We decided to predict the future trajectory of the motion of the animal, conditioned on the previous detections. This formulates the problem as sequence to sequence (seq2seq) learning, which is typically solved using recurrent neural networks (RNN; Sutskever et al., 2014). These architectures were prevalent in both the 3D human pose prediction literature and pedestrian trajectory literature. Transformer models (Vaswani et al., 2017) could also be useful in solving this problem, but their inference time made them irrelevant. Additionally, we used a Kalman filter with a constant velocity model to predict future states of the system, as a sort of a baseline predictor.

Alternative formulations of the problem were predicting the time and location of the screen touching directly from a sequence of coordinates. The first and more important reason we chose not to use this formulation is lack of data. This experimental paradigm and setting is novel, and the animals are still in the process of learning the task. Another reason is a theoretical one: the system is supposed to allow some interactive manipulation of the stimulus present on the screen, in a way that could change the animal's behavior. Thus, the behavior that the predictor was trained on could differ from future behaviors. While predicting the actual coordinates of the trajectory is a more general and harder problem, it can also have more uses, and it's agnostic to the experimental setting.

Overview of the solution

Each frame of the real-time camera is passed to the detector, which returns a single 2D bounding box, if the animal is present and the detection confidence is above some threshold. The bounding box coordinates are then corrected for lens barrel distortion, and transformed to a constant coordinate system, in which the screen in the arena lies on the X-axis (full technical details

regarding the correction and the transformation will follow). After processing each frame, the past m detections are used to predict the next n detections. If the Y value of some of the points in the trajectory cross a certain threshold, a screen touch (hit) is predicted. Currently, the RNNs produce unsatisfactory predictions, and further behavioral data and research are needed. The Kalman filter based prediction is able to produce a crude prediction of the motion.

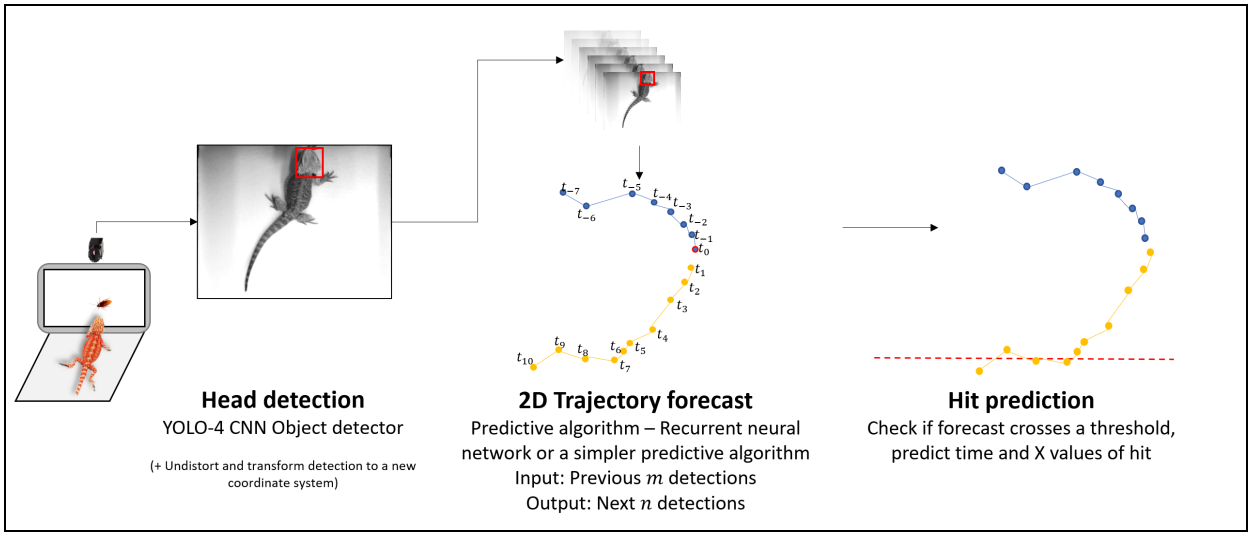

Figure 2: Overview of the real-time detector-predictor module

Detection

Framework and dataset

The detector uses the default YOLOv4 architecture with a CSPDarknet53 backbone network as described by Bochkoviskiy et al. (2020). We used the authors' original implementation based on the Darknet framework (https://github.com/AlexeyAB/darknet). The network was trained to detect the Pogona's head using a custom labelled dataset, and then integrated into the system through darknet's Python API.

In order to train the model to detect the new object we used transfer learning (Pan & Yang, 2010), where the model is initialized with pre-trained weights, and then further trained on a custom dataset. This procedure makes it possible to achieve good results with a much shorter training session, and on a much smaller dataset than would be required when training from scratch. In our case the initial weights were based on the Microsoft COCO object detection dataset (Lin et al., 2014).

The custom dataset consists of 3,650 images (3,312 train images, and 338 validation images), containing a single object class. 2,256 images contained a single instance of the lizard's head, and were manually annotated with appropriate bounding boxes. 1,592 of them consisted of frames taken using the same camera that is used in the real-time system, while the rest were based on videos recorded with mobile phones. In most of these images the lizard is viewed from above to approximate the viewing angles that are later used for detection. However, the dataset does contain a variety of different poses and viewing angles to make sure the model is able to generalize to novel poses.

In order to prevent false positive detections, we used 1,394 negative examples where the object was not present. About half of those were various images of birds and insects randomly chosen from the ImageNet dataset (Deng et al., 2009). The other half consisted of images taken using the system's camera where the lizard was not present in the frame.

Since the real-time system uses a grayscale camera, all of the dataset images were converted to grayscale before training. Nonetheless, we had to use a model configuration with 3 color channels in order to match the configuration that was used when the pre-trained weights were generated.

We trained the model for 10,000 batches, each batch consisting of 64 images, and used the training parameters recommended by the algorithm designers. Training lasted about eight hours using an Nvidia RTX 2080ti GPU with the C-based implementation of Darknet. The model with the best validation-set accuracy was selected for use in the real-time system. Training consisted of several sessions, where after each session we examined the model, and added undetected images to the training set until achieving satisfactory results on unseen data.

Results

Since the system is only required to detect a single object instance of a single class in each frame, we were mostly interested in measuring the bounding box accuracy, as well as the fraction of false positive and false negative detections. Object detection models which detect multiple classes are usually evaluated using the mean average precision metric (mAP), however this seemed redundant in our case.

Bounding box accuracy was measured on an additional test-set of 1,210 images sampled from experiment video recordings, using the intersection-over-union metric (IoU), which is the intersection of the predicted and ground-truth area of the bounding boxes, divided by their union. The model achieved a median IoU value of 0.86. The 5th percentile accuracy is 0.77, and the 95th percentile accuracy is 0.94 (see figure 3).

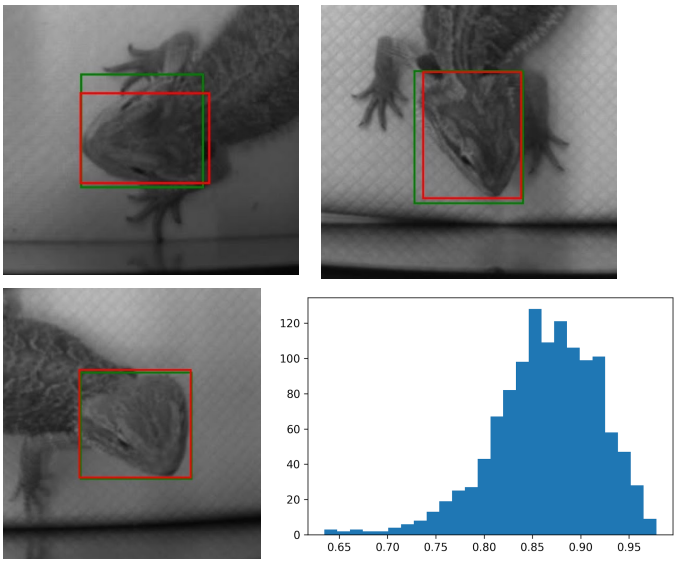

Figure 3: IoU distribution and example detections. The ground-truth bounding box is green, and the predicted bounding box is red. Top-left - 5th percentile; Top-right - median; Bottom-left - 95th percentile detection example. Bottom-right - IoU distribution.

The fraction of *false negative* detections was calculated over the entire experimental dataset at the time of testing, containing around half a million video frames. The detector was tested using a confidence threshold of 0.8. Each detection is assigned a confidence value by the detector that estimates how likely the object belongs to a class, the lizard's head in our case. Approximately 0.6% of the frames contained a lizard and were undetected by the model when using this threshold. This number can be reduced even more by lowering the threshold, although this would theoretically increase the risk of false positive detections. Including the undetected images in the training set and retraining the model would probably also reduce the fraction of false positives. False negative images come in roughly three categories - difficult poses that were not present in the training set, images where it is hard to differentiate between the lizard body and its head due to body pose or lighting issues, and some images with unusual objects in the frame, such as the experimenter's hand which sometimes confused the detector.

False positive detections, where an object which is not the animal's head is erroneously detected, were not found in the experimental dataset when using a confidence threshold of 0.8.

The object detector average latency is 11.2ms or 89fps, when tested on the experiment system hardware (Intel Core i7-9700 CPU, 3.00GHz, Nvidia RTX 2080ti GPU). This is achieved when frames are resized to a resolution of 416x416 before passing them to the model. The latency can be lowered by resizing to even lower resolutions, however this could result in lower accuracy. By using

the tkDNN-TensorRT implementation on NVIDIA GPUs, the latency could be theoretically improved to up to 104fps without loss of accuracy (Gatti, 2017/2020). Using the YOLO-v4-tiny configuration is another option for dramatically lowering the latency. This configuration uses a much smaller network, and can reportedly achieve 443fps on images with 416x416 resolution on the same GPU. It should be tested whether using lower resolution or YOLO-v4-tiny would keep the accuracy high enough for this specific simple dataset and relatively undemanding setting.

Data collection

In this section we'll detail the data collection and processing for analysis and training.

Correction and transformation of the data

The input of the module are grayscale 1440×1080 frames captured by a FLIR Firefly camera at approximately 60fps. The detection of the head is performed on the input frame, and the coordinates of the bounding box are corrected and transformed, both during online inference and offline data analysis.

The image is captured from a relatively close distance, and thus significant barrel distortion is introduced by the lens. Most importantly, the screen, seen in the bottom of the frames in the figures below, seems curved. To solve this issue, a constant camera matrix was computed and used to undistort the coordinates of each detection.

After correcting for the lens distortion, the coordinates are transformed using an homography transformation to a coordinate system in which the bottom of the screen approximately lies on the X-axis. Another reason for transforming the data was to correct for any small movements of the camera itself from trial to trial. For this purpose, we placed four Aruco markers, which are predefined markers used for image calibration. The resulting rectangle is the 2D plane in which the bounding box moves. It is a simplification of the actual movement, and restricts the analysis to 2D motion of the head, while disregarding movement on the Z-axis. Transforming the data into a 3D space would have required combining input from multiple cameras, which was both more complicated and seemed unnecessary to achieve our goal.

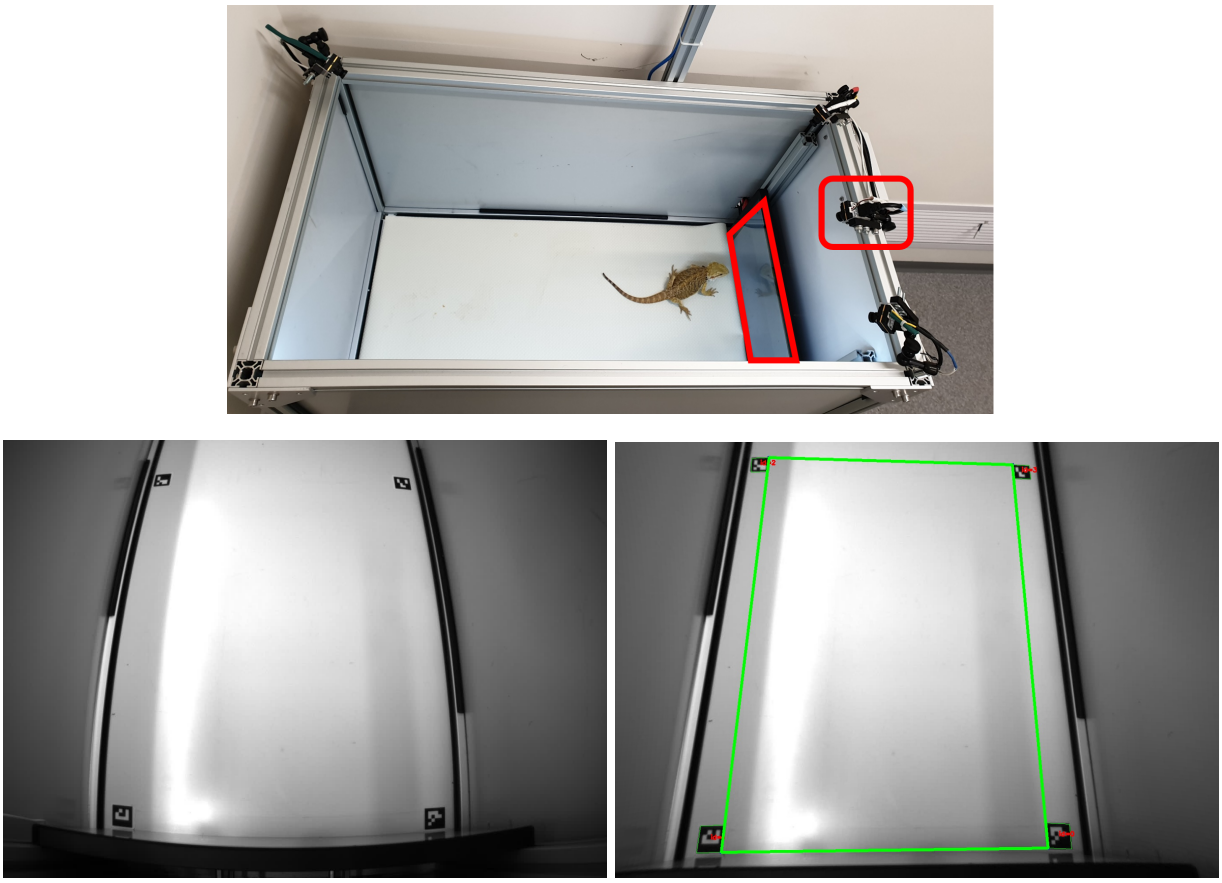

Figure 4: Top - The experimental arena in the lab. Marked in red are the touch screen and the real-time camera. Left - The RT frame as it is recorded by the camera. The curvature of the screen can be seen in the bottom of the frame. Right - the lens corrected frame, with lines drawn between the four Aruco markers, detected automatically by the system. The bottom line is the X-axis of the new coordinate system, and marks the edge of the screen.

Description of the data

For the offline analysis of the data, the bounding boxes of the head of the animal for each frame from each trial were saved. Each trial is at most a few minutes long, which translates to several thousands of detections.

The screen on which the stimulus is displayed is a touch screen, and any touching of the screen is recorded separately. The time of the frame acquisition and the time of the touching of the screen can then be compared, and detections where a touching occurred can be marked, and used for analysis combining both data sources. It's worth noting that the timings recorded for the acquisition of the frame and the touching of the screen somewhat differ , due to technical issues related to the screen and to the arena system. As noted before, we did not use the screen touchings for training

the model, as their quantity was relatively small, so we did not investigate these technical issues any further.

To use the data for solving the sequence to sequence formulation of the problem, pairs of sequences were created from the data by rolling two sliding windows over the bounding boxes of each trial. Formally, the data is:

$\{(x_{(i,\ i+m-1)}, y_{(i+m,\ i+m+n-1)}) \mid x \in R^{m\times 4},\ y \in R^{n\times 4}, 0 \le i \le L-m-n\}$, where the subscripts mark the interval of detections that comprises the sequences, and L is the number of frames in the trial, m is the length of the input, and n is the length of the output, which are important parameters that we'll discuss later on. Each detection is a pair of points $\{(x_1, y_1), (x_2, y_2)\}$ which are the top left and bottom right of the transformed bounding box, respectively..

Data metrics

We sought to understand some of the features of the sequence data. For that purpose, we used several metrics:

Average speed - average L_2 norm of the first difference vectors $\frac{1}{k}\sum \left\| x_i - x_{i-1} \right\|_2$, of the bottom-right corner. The units are pixels on the corrected and transformed frame. This serves as a basic and intuitive measurement for motion in a sequence. But further analysis revealed that the bounding box sometimes jitter when the animal does not move, which required finding other metrics to find sequences in which the animal actually moves.

Pearson's r - we tried computing the absolute correlation between the x and y values of the coordinates in the sequence, as a measure of the linearity of the movement. This metric disregards the temporal relation between the points, and thus was not very helpful.

Average Angle Difference ("Zigzagity") - average angle between consecutive velocity vectors: $\frac{1}{k}\sum cos^{-1}(\frac{u_i \cdot v_i}{|u_i| \cdot |v_i|})$, where $u_i = x_i - x_{i-1}, v_i = x_{i-1} - x_{i-2}$. This aims to remove any sequences that contain mainly sensor jitter and not actual movement, and also to find sequences that contain "interesting" movements, i.e., straight or slightly curved lines, whose zigzagity values are relatively small. This metric proved the most useful for our purposes.

From 223 trials, we collected approximately 1M detections, from which we generated approximately 860K sequences. This analysis refers to x, y sequences where $m = n = 20$ detections, which corresponds to approximately 330ms for each of the x, y sequences. It should be noted that these sequence lengths are somewhat arbitrary, and we'll discuss the output length issue later on.

The majority of the data, around 80%-90%, is relatively motionless, and some of it is sensor noise. We reached this conclusion by sampling sequences according to their Zigzagity scores, and inspected them qualitatively. This is due to the fact that the animal did not move, or moved very little, most of the time.

Below are the distributions of these metrics, computed on the x sequences, alongside relevant quantiles to demonstrate the scarcity of the data. As visible in both the speed and zigzagity, the distributions seem to be composed of a Gaussian distribution, which probably contains the static noise in the data, and a tail, which comprises actual motion data. Qualitative sampling and inspection of the data revealed that mostly sequences with Zigzagity of about 1.7 or less are actual motion data, which is about 10-15% of the data. Significant movements seem to have scores of less than 1, which accounts for only 2-3% of the currently available data - only several thousand sequences, and an even smaller number of actual "motion events", as the sequences overlap when generated by the sliding window method.

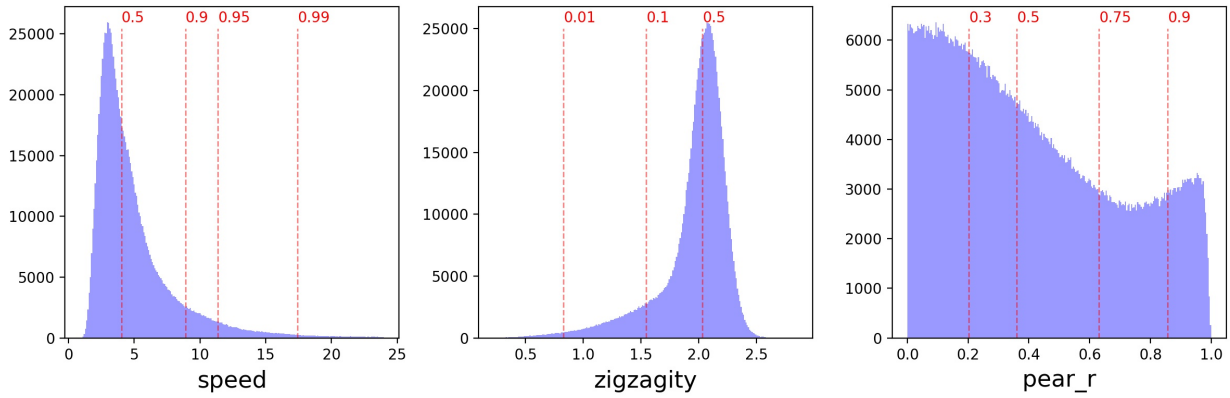

Figure 5: distribution of the various metrics on the entire sequence data, n=20. Pearson's r correlations are in absolute values

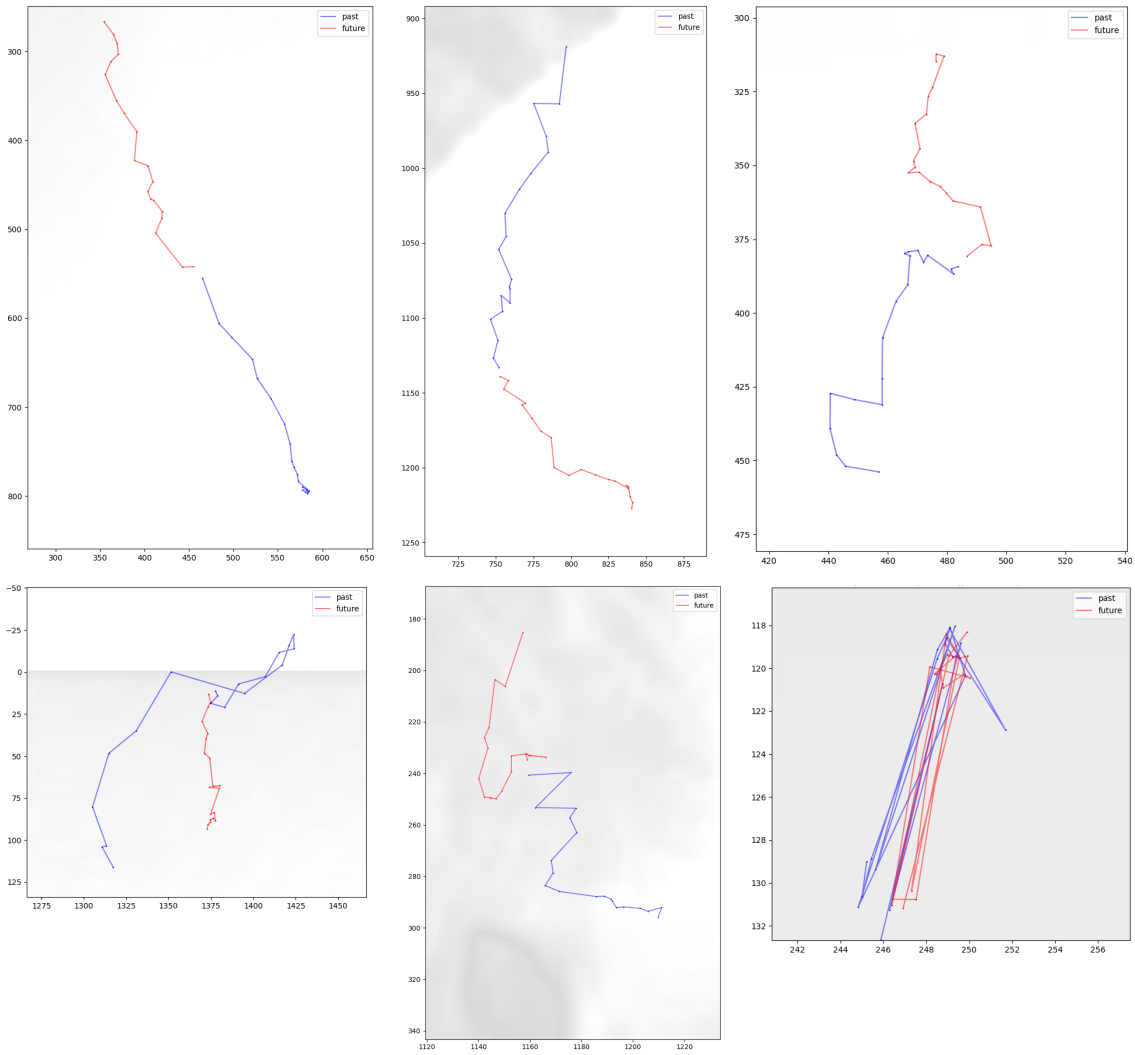

Figure 6: Examples of sequences with different zigzagity values. Top-left - 0.2-0.3, top-middle - 0.6-0.7, top-right - 0.8-0.9, bottom-left - 1.3-1.4, bottom-middle - 1.5-1.6, bottom-right - 2.1-2.2, this is an example of sensor jitter which comprises a large part of the dataset.

Screen Touching Prediction

Trajectory Prediction

As noted before, to predict future screen touches by the animal, we used an algorithm that predicts the future trajectory of the head of the animal, based on previous detections. We mainly aimed to model and predict the sharp and fast movements of the animal as it tries to capture the prey that appears on the screen. From the data available up until now, the animal initiates a

movement of the head towards the screen from a static posture, and such movements usually last about 150-350ms from the initial acceleration to touching of the screen. These movements are not necessarily linear, and can be curved, depending on the animal posture in relation to the screen, and probably other unknown factors.

We decided to predict the motion of the bounding box itself, and not only of a single coordinate, such as the centroid of the bounding box, as it seemed that the bounding box can capture information regarding the shape of the head as it moves, and supply additional features that the learning algorithms can use.

Kalman Filter based trajectory prediction

One simple solution for the trajectory prediction problem is using Kalman filtering. A Kalman filter is a discrete time recursive algorithm that estimates the state of a system with a known linear dynamic model based on noisy measurements at each time step. Assuming a gaussian measurement noise, the filter can optimally minimize the mean square measurement error over time. Using the filtered (estimated) position and velocity of the bounding box, it's possible to produce an extrapolation of the trajectory which is based on a very simple physical model of the motion.

In order to use the filter for trajectory prediction we use the estimated state vector $u_t = (x_t, \dot{x}_t, y_t, \dot{y}_t)$ for each bounding box corner at each time step, and project it to the future by using the state transition matrix F. The matrix transforms the state vector to the state vector of the next time step, assuming the simple linear dynamical model holds, i.e. $u_{t+1} = F u_t$. By recursively applying this equation, a series of state vectors $u_{t+i} = F^i u_t$ can be calculated. Predicting the trajectory using this method produces a very crude estimation of the actual future trajectory, since the movement of the animal head is quite complex and nonlinear. However, it could suffice for the experimental requirements.

We implemented the Kalman predictor by using two kalman filters, one for each corner of the bounding box. This was done to simplify the covariance matrices of the process and measurement error variables. Assuming the x and y coordinates of each point are independent variables with identically distributed errors, this allows using scalar covariance matrices. At each time step the current detection is passed through the filter, and a trajectory forecast is generated based on the current estimated state vector. We tried two dynamical models - a constant velocity model, and a constant acceleration model. When using the constant velocity model, the predicted trajectory is simply a straight line in the direction of the current estimated velocity vector, while the constant

acceleration model can produce various quadratic trajectories, which usually vastly overshoot the actual trajectory.

Since the distributions of the detector measurement error, and model error are unknown, we used a hyperparameter optimization library (https://github.com/facebook/Ax) to search for variance parameters for each covariance matrix. The parameters were optimized to minimize the average displacement error (ADE; see in the next section) between the predicted trajectory and the true future trajectory over all of the collected experiment trajectory data.

RNN based trajectory prediction

In an effort to achieve better predictions of the nonlinear trajectories produced by the movement of the lizard head, we additionally experimented with several recurrent neural network (RNN) based trajectory predictors.

RNNs (Elman, 1990) are a class of artificial neural networks that use recurrent connections, that provide the network with memory capabilities, and allow it to store representations of temporal data. Thus RNNs achieve much better results in time-series prediction tasks in comparison with feed-forward networks. RNNs are optimized using backpropagation through time (Werbos, 1990), which requires calculating a long chain of gradients with respect to the loss of the entire sequence of network outputs. This makes optimization difficult due to the vanishing gradient problem (Bengio et al., 1994). Long Short Term Memory (LSTM; Hochreiter & Schmidhuber, 1997) and Gated Recurrent Unit (GRU; Cho et al., 2014) are two RNN variants designed to solve this problem by using "forget" gates that allow the network to selectively persist only relevant information about past inputs.

Predicting future trajectories by using data of past trajectories amounts to a sequence to sequence learning problem, similar, for example, to machine translation. RNNs can be used to perform such tasks by using an encoder-decoder architecture, where one network encodes the input sequence into a fixed size vector, and a second network decodes the vector to produce the output sequence (Sutskever et al., 2014).

We experimented with several variants of this encoder-decoder architecture. In each case, an input sequence $(x_i, ..., x_{i+m-1}) \in R^{m \times 4}$ of past bounding box coordinates is first transformed to a sequence of velocity vectors $(x_{i+1} - x_i), ..., (x_{i+m-1} - x_{i+m-2}) \in R^{m-1 \times 4}$, and then passed as input to the encoder. The encoder then processes each vector in the sequence recurrently, and outputs its

hidden state vector containing a representation of the whole sequence until that point. The last hidden state vector is passed to the decoder.

We tried two different decoder architectures - an RNN decoder (GRU or LSTM), and a decoder consisting of a single linear layer. The former was inspired by Martinez et al. (2017), where a similar network is used to predict human poses, and the latter was inspired by Becker et al. (2018), which uses a linear decoder for pedestrian trajectory prediction. The RNN decoder uses the last output of the encoder together with the last velocity vector of the input to recurrently produce a sequence of predicted velocity vectors. In each iteration the previous velocity, and hidden state vectors are used to produce the next hidden state vector. Then, a linear layer transforms the vector into a new velocity vector, which is used as the input for the next iteration. The linear decoder simply takes the last output of the encoder and passes it to the linear layer. Its output vector is reshaped to produce the complete output sequence in parallel.

In each case, once the sequence of velocity vectors is produced, it is transformed to a sequence of absolute position coordinates using the formula:

$$\hat{y_t} = \sum_{k=m}^{m+t} v_k + x_m \text{ for } t \in (1,\ n),$$

where $\hat{y_t}$ is the *t*-th output vector, v_k is the *k*-th velocity vector starting from the first input velocity vector, and m is the length of the input sequence.

Additionally, we tried using the absolute position coordinates as additional input features by concatenating each position vector to its corresponding velocity vector.

We experimented with both LSTM and GRU networks for the encoder and decoder networks. We also tried using encoder-decoder networks with tied weights, where both networks are RNNs with shared weights between them.

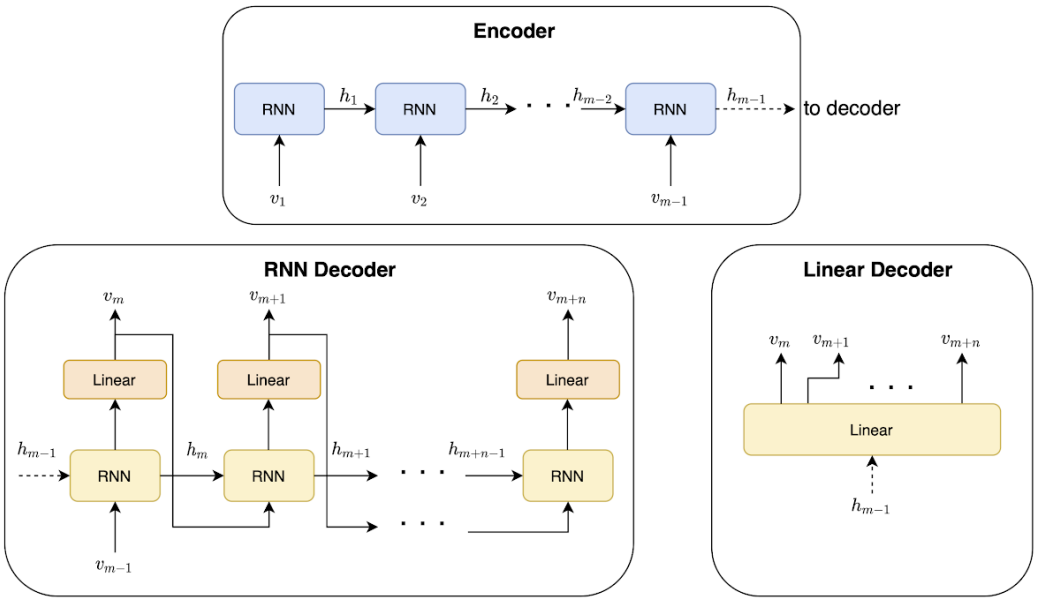

Figure 7: Decoder-encoder artificial neural network architectures.

Training process

Since most of the dataset contains static trajectories with little motion, we only selected training sequences with certain properties, according to the metrics described in the Data Collection section. We aimed to find subsets of the data containing actual motion data, and not static noise. We split the dataset to a training, validation, and test sets, making sure trials were divided evenly according to the proportion of sequences that were selected from each trial.

We also experimented with various techniques related to the training procedure itself, and not to the network structure.

Loss functions: The main loss function we used was the average displacement error (ADE), defined as the average L_2 norm between the predicted coordinate and the ground truth coordinates $\frac{1}{k}\Sigma||y_i - \hat{y}_i||_2$ .This was also the main evaluation metric we used to compare the different models. We also tried various variants of this loss:

- **Weighted ADE**, where the error term of each timestep was weighted according to its place. We aimed to give greater weight to errors made close to the end of the prediction, as the prediction becomes harder when advancing in time.
- **Randomized ADE**, where we computed the errors up until some random index in the sequence, drawn uniformly from the interval [1, prediction length]. We aimed that the loss

will be influenced by both short and long term errors, as the errors naturally grow when advancing in time.

- **FDE**, final displacement error, where the error term is $||y_n - \hat{y}_n||_2$ for each sequence

Hyperparameter tuning: We used AX, a hyper-parameter optimization library, to try to find optimal hyper-parameters.

Scheduled sampling: Inspired by Bengio et al. (2015), we tried feeding the ground truth data instead of the prediction at random positions in the decoder sequence. When using the scheduled sampling regime, we toss a coin with some probability, and according to the result, replace the prediction generated in the previous iteration by the actual velocity from the ground truth data. The probability decays as the training progresses, according to some function.

Results

We experimented with multiple configurations involving the RNN architecture and training hyper-parameters. Despite our efforts, the results we achieved with the RNN based trajectory predictor, and consequently, with the hit predictor, were unsatisfactory. This is true, regardless of the experimental requirements from the hit predictor, which are not yet clearly defined, and regardless of the way the hit predictor is evaluated. This stems from the fact that currently, the RNN based trajectory predictor's behavior is erratic, and it's hard to understand what makes it behave so unpredictably. In contrast, the Kalman based predictor will probably produce a large amount of false positives, as the prediction is crude and usually overshoots beyond the actual trajectory. However, its behaviour is known and the results are somewhat predictable.

After conducting many experiments, it seems implausible to us that some unknown configuration of the hyperparameters can lead to significantly improved results, and due to the very high dimensionality of the hyperparameters space, it's infeasible to conduct an exhaustive search and reach a more definitive conclusion.

Assuming this is the case, there are several possible reasons for these results. First, the amount of usable data in the dataset might not be large enough to interpolate and generalize to unseen situations. Second, the input features we used might have been insufficient to predict the animal's behavior. Lastly, it's possible that even with better features, and more data, it would still not be possible to predict the motion trajectories of the animal, as its movement could be too unpredictable in the high spatio-temporal resolution we used.

To demonstrate, below is a comparison between networks that significantly differ in their complexity: a small network with a single GRU layer (in both the encoder and the decoder) and a

small 32-dimensional hidden vector; a network with 2 GRU layers and a 200-dimensional hidden vector; and a relatively large network, with 3 GRU layers and a 600-dimensional hidden vector. The models were trained on a subset of the training set containing sequences with a "Zigzagity" of less than 1.5. Figure 8 shows both qualitative and quantitative results of the 3 models (chosen according to the lowest validation set accuracy), and the Kalman prediction. It is evident that the models do not significantly differ on the ADE metric on the test-set, and that the predictions are erratic for all models, and as such, no better than the more crude but predictable Kalman based prediction.

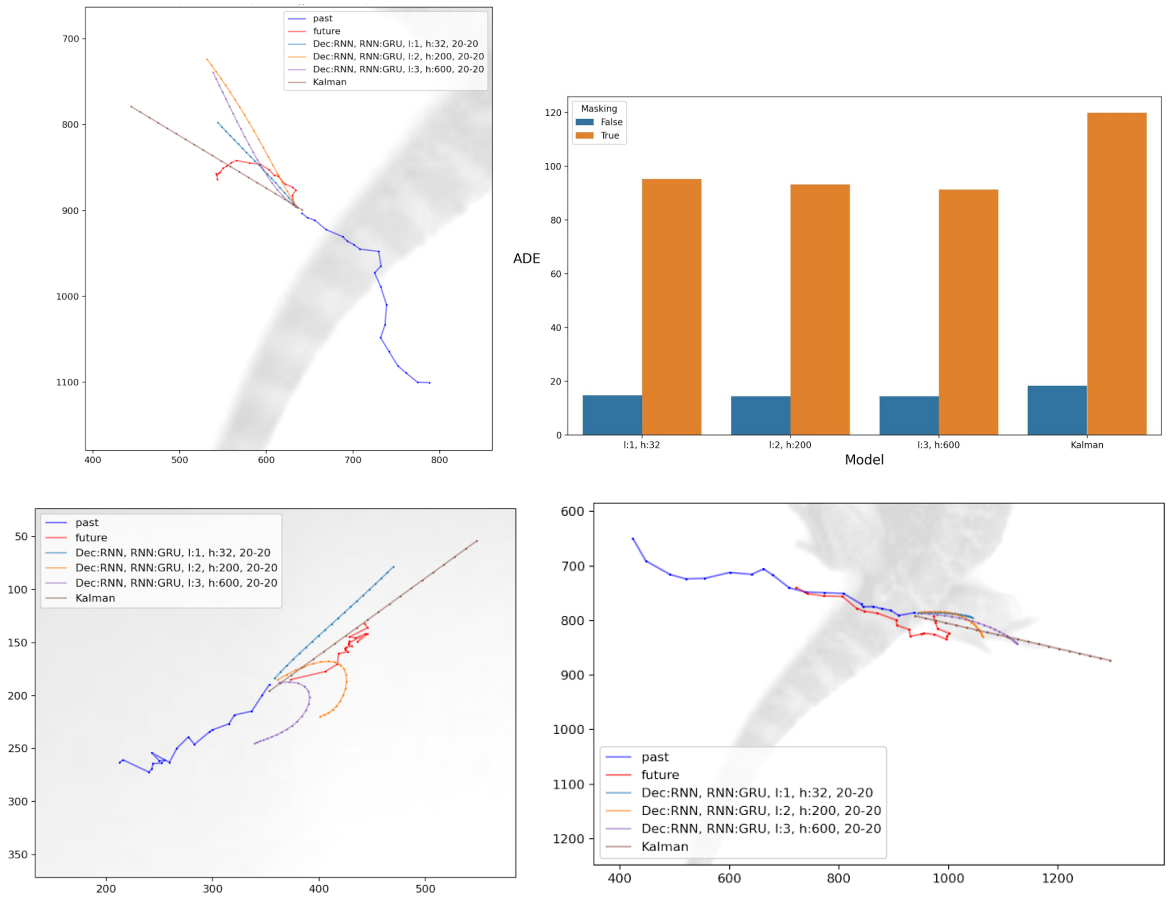

Figure 8: Top-right - ADE evaluation results for each model on both the whole test-set (blue), and the subset on which the networks were trained (orange). The ADE of the subset is much higher than for the whole dataset since the dataset contains mostly static sequences. Top-left, bottom - examples of predicted trajectories produced by each model. "past" indicates the input sequence, "future" indicates the ground-truth output.

Further support for the importance of the data and its properties, can be seen when comparing models trained on various subsets of data. We compared 3 identical models, that were trained on: a very small subset of the data, containing only high speed motion; a random subset containing 35% of the data, and thus contains mostly sequences with no motion; and a subset containing sequences

with a “Zigzagity” of less than 1.5. As evident in figure 9, the model that was trained on high speed sequences only tends to overshoot significantly, even when the input sequence contains little to no motion.

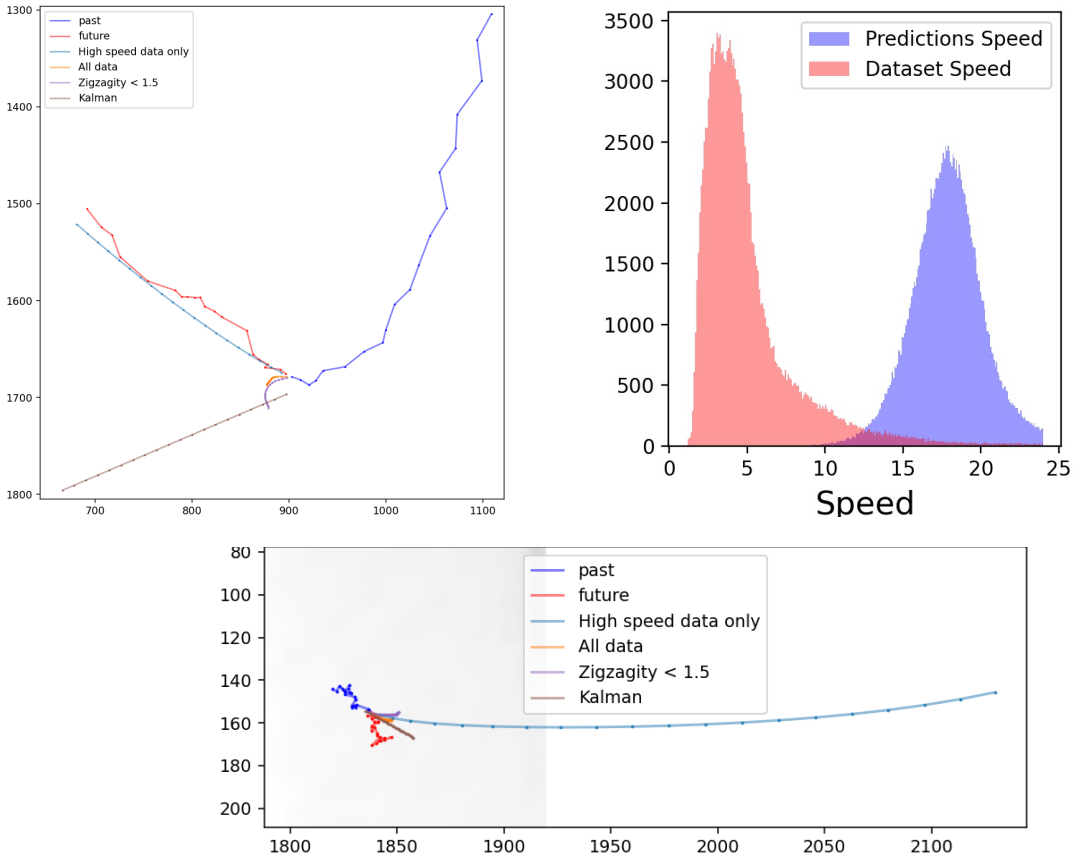

Figure 9: Top-left: An example of a high-speed sequence. The high-speed model prediction is relatively accurate. Top-right: Comparison of the average speed distribution of the ground-truth vs. the predicted trajectories of the high-speed model. Bottom: An example of a static sequence. The high-speed model predicts a fast motion trajectory.

Throughout our experiments we used sequences of 20 time steps for both input and output. Preliminary tests revealed that the predictor achieves slightly higher ADE scores when using an input sequence length of 20 versus shorter ones. The difference between the predictions and the ground-truth naturally grows as the sequence progresses. This important hyperparameter determines how early a hit prediction can be given to the system, a timeframe that needs to be determined by the experimental requirements, which are still not clear. It also influences the way the model is trained, as longer output lengths result in larger error terms, which in turn produce different gradients, in contrast to shorter predictions. These important issues need to be further explored.

The inference time of the trajectory predictors varies with its architecture. The Kalman predictor runs on the CPU and its inference time is practically negligible. The RNN based predictor with the linear decoder predicts the entire output sequence at once, and is much faster than the RNN decoder. The inference time of the RNN based predictor with an RNN decoder, depends on the output sequence length, as it needs to generate the output iteratively and cannot be optimized, as the output of the previous iteration is required as the input of the next iteration. It should be noted that the time constraints limited the complexity of the models we could use.

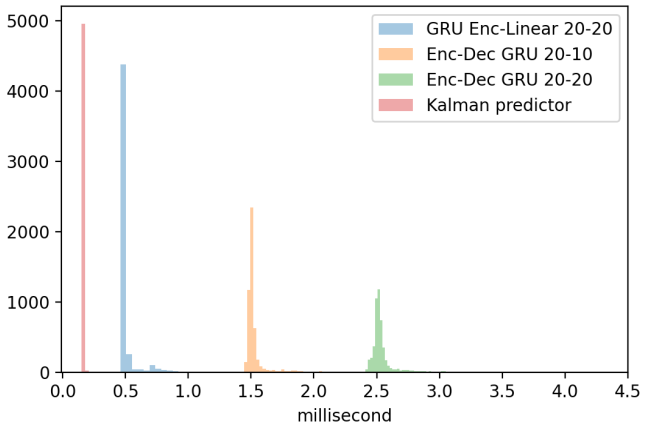

Figure 10: Inference time distribution for several trajectory predictors.

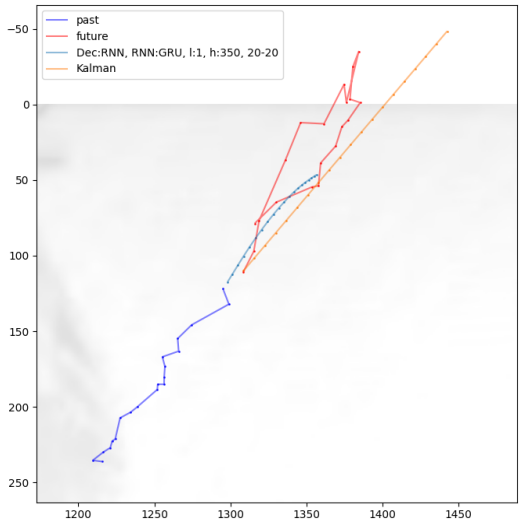

Figure 11: A hit sequence with several predictions. The Kalman filter will predict a hit when its prediction crosses the $y = 0$ line for example.

The trajectory prediction is used to produce a hit prediction by checking if the y coordinate of some prediction crosses a certain threshold (see figure 11 for an example). Following are raster plots describing the predictions alongside the actual screen touching events, recorded by the

screen, for a single trial. For each time step, a vertical bar (colored by the predicted x coordinate) is shown, if some prediction predicted a hit at this time step. The y-axis value of the bar indicates the number of time steps between the time the prediction was produced and the time step of the predicted hit. The dashed red line indicates screen touchings recorded by the screen. Ideally, the hit predictions and the ground truth red line should be exactly aligned.

The y-threshold is another parameter that can influence the accuracy of the predictions, as the spatial alignment between the screen and our coordinate system is not completely exact, and the temporal alignment is also affected by latencies in the system. Since the animal usually accelerates when attacking the screen, and the Kalman predictor uses a constant velocity model, it tends to predict that the hits will arrive later than they actually do, when the prediction time is early relative to the screen touching, and earlier than they actually do, when the prediction time is close to the screen touching (See figure 12, top left plot).

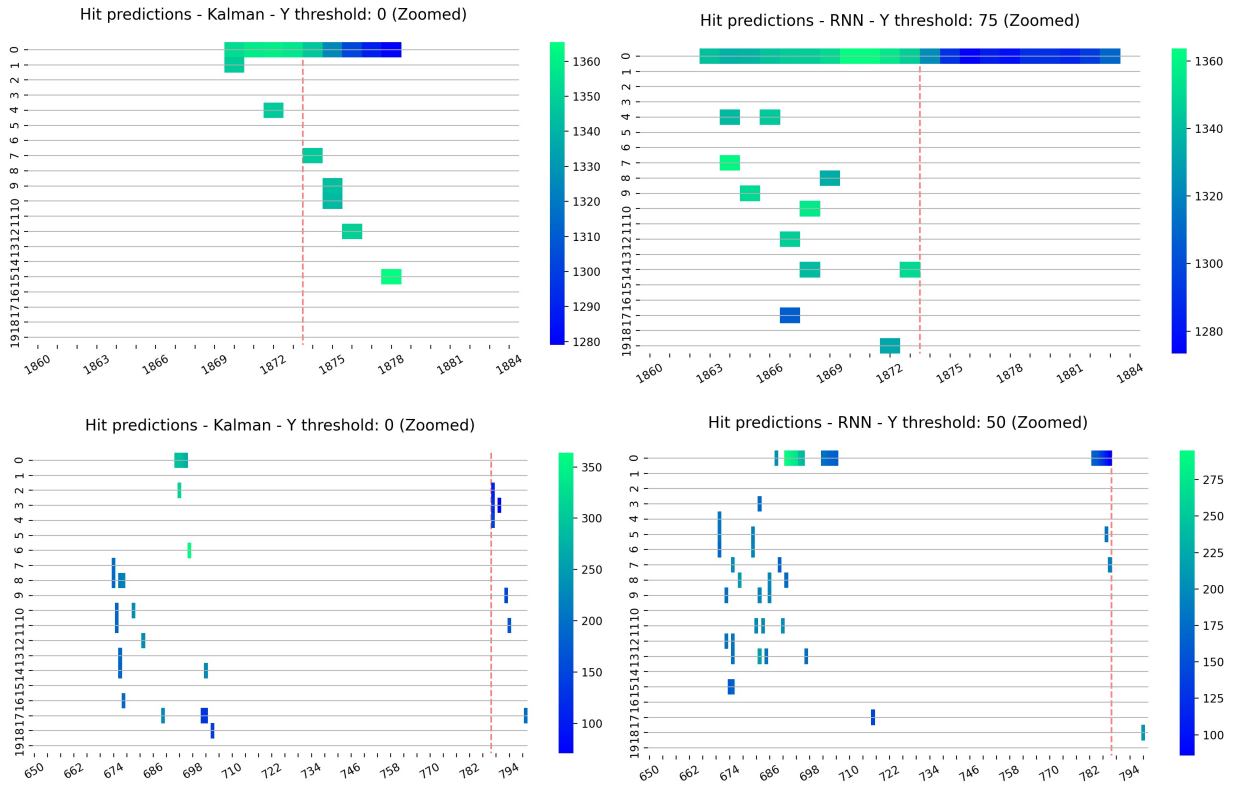

Figure 12: Each row in the figure shows a single screen touching event from different experiment trials. In the top plots the hit event was relatively successfully predicted, while in the bottom, both predictors first falsely predicted a hit and the actual screen touching event was poorly predicted. The duration of each time step is approximately 16ms.

Further work is necessary in order to evaluate the performance of the hit predictors. More exact metrics need to be devised according to the experimental requirements, such as more precise definitions of false positive and false negatives, and more complex metrics that include both spatial and temporal dimensions, across multiple screen touching events, and more.

Code Structure and Software Modules

General flow of the RT module

The video frame, received from the Arena module, is passed to a Detector object which returns a bounding box detection of the Pogona head in the frame. The bounding box coordinates are then transformed to a coordinate system relative to the screen using the Calibration module. The transformed detections are added to an array of past detections and this data is passed to a TrajectoryPredictor object that generates a trajectory forecast. From this trajectory forecast a hit prediction is generated, consisting of an approximate x coordinate of the hit (relative to the screen resolution), and the number of frames until the predicted hit. A hit is predicted when the bottom edge of the bounding box passes a certain y-coordinate threshold.

Sub-modules of the RT module

Following is a list of the modules and their high-level role in the system, the main objects and their relations, libraries and frameworks used, code design considerations and drawbacks, and optional solutions where appropriate. The code is written in Python 3.7, and requires installation of several other non-Python packages, such as Open-CV and the YOLO-v4 Darknet library.

calibration - Responsible for geometric transformations for correcting the distortion of the lens and transforming the data into a unified coordinate system, using existing functions from Open-CV. The functions are called by the HitPredictor object, and by functions from the Dataset module for offline processing.

This module, and especially the homography computation using the Aruco markers, was written to fit the experimental arena, but it can be refactored to fit other settings with relatively little work.

dataset - Responsible for the offline parsing and organization of the detections data, and for loading the data into the memory for analysis. The module's functions mostly operate with Pandas DataFrames. This module is written to analyze and organize 2D bounding box detections in files and load this data for analysis in a Jupyter notebook.

This has several design drawbacks:

- For the analysis part, the data is required to be 2D bounding box detections. Using different and more complex data (such as 2D multi-joint pose) would require a significant refactoring of the code.
- The code for loading the dataset into the notebook for analysis, is usable as long as the DataFrame of the detections does not exceed in size the available memory. In addition, the sequences that are produced from the detections dataframe require even more memory resources, depending on the input and output lengths. With the relatively small amount of data that was available to us, we had no problem loading and analyzing the data in its entirety. It's possible to consider the following solutions: splitting the data into parts that each fit in the memory, and processing them separately; organizing the data in a unified database, and conducting the analysis and training using SQL based queries; conducting the analysis per a single detections file, and loading into memory only the required data; or using a scalable analytics framework such as Dask.

detector - Responsible for implementing the detector of the animal's head. Contains an abstract class of a Detector, and currently a single implementation of such Detector using the YOLO-v4 algorithm. It's possible to use other detectors with the same interface of the Detector abstract class. Additionally, it contains various functions for transforming between different bounding box formats and computing the IoU metric.

The output of this module is the data that the rest of the modules operate with. It's currently restricted for using a single 2D bounding box, and refactoring the rest of the modules to use a more complex output (a multi joint 2D pose, for example) would probably require some work.

kalman_predict - This module is responsible for creating Kalman filter objects using the filterpy python library (Labbe, 2014/2020), and implements a trajectory predictor based on Kalman filters.

predictor - This module is responsible for real-time analysis and prediction of the Pogona behavior in the arena. It contains the HitPredictor class, which is the main entry-point of the prediction module, and is responsible for processing camera video frames into predictions of touch

events in real-time. The module also contains an abstract trajectory predictor class which defines the trajectory predictor interface used by the hit predictor.

seq2seq_predict - This module contains an implementation of a TrajectoryPredictor subclass based on encoder-decoder neural network models, as well as a customizable PyTorch encoder-decoder neural network module.

train_eval - This module is responsible for generating sequence data, sampling and masking it using various statistics, training models and evaluating them. It basically starts from the detections DataFrame, and generates Numpy/PyTorch 3D tensors (array of sequences) using the sliding window method described in the data section. Some design drawbacks of this module are: similarly to the Dataset module, the sequences arrays that are generated by this module are not scalable beyond the memory capacity of the machine. Similar solutions should be considered for generating the data and the training procedure. Another drawback is the separation between the dataset and the train_eval modules. It might have been helpful to place the code responsible for generating the sequences in the dataset module, and creating solutions for the larger than memory problem for both the detection data and the sequence data.

traj_models - This module contains useful functions for organizing various encoder-decoder models and storing them together with any parameters used to generate them for later use. It contains high-level functions for training models, maintaining a named list of trained models, and rebuilding previously trained models, so that they can be further evaluated and tested.

visualize - This module is responsible for creating visualizations for images, videos, statistical plots and notebook widgets.

The source code of the prediction module, including Jupyter notebooks with usage examples, can be found at:

https://github.com/neural-electrophysiology-tool-team/Pogona_Pursuit/tree/master/Arena/Prediction

Conclusion and Future Directions

This project's goal was to design and implement a module which provides an experimental system the option to react, in real-time, to an animal's behavior. Our module used computer vision and machine learning tools, as the sole input to the system is a stream of video frames. Our module is able to accurately detect the location of the animal's head, and produce a crude estimate of the

animal's future head movements, relative to the target screen. Machine learning based solutions to predict the animal's future trajectory were explored and, in our view, have the potential of providing accurate predictions, yet more data and further research are needed.

In addition to ideas that were already mentioned throughout the report, our project points to some directions that could be further explored.

If similar behavioral experiments with *Pogona vitticeps* are to be conducted frequently in the future with similar systems, producing a standardized dataset of videos and 3D poses, for developing and evaluating motion and general behavior models, could prove useful. For example, the Human 3.6M benchmark, that includes 3.6 million 3D labelled poses of humans, performing various tasks, recorded at 50fps (Ionescu et al., 2014), is used in the area of human motion prediction.

A substantial amount of data could allow training models that predict screen touches directly, without first producing trajectories. This might be an easier machine learning task. More generally, simpler solutions such as coupling the movement of the prey more directly with the animal's location in the arena and trying to maximize the distance between them, might prove useful.

Analyzing an animal's motion in high spatio-temporal resolution and in a live feedback experiment is a challenging and relatively new research topic, but has the capacity to further our understanding in a wide array of topics in behavioral and neuroscientific areas. We hope our tools and methods might also be useful for other experimental settings, with the appropriate modifications.

References

Aksan, E., Cao, P., Kaufmann, M., & Hilliges, O. (2020). A Spatio-temporal Transformer for 3D Human Motion Prediction. *ArXiv:2004.08692 [Cs]*. http://arxiv.org/abs/2004.08692

Alexey. (2020). *AlexeyAB/darknet* [C]. https://github.com/AlexeyAB/darknet (Original work published 2016)

Ax · Adaptive Experimentation Platform. (n.d.). Retrieved October 21, 2020, from https://ax.dev//index.html

Becker, S., Hug, R., Hübner, W., & Arens, M. (2019). RED: A Simple but Effective Baseline Predictor for the TrajNet Benchmark. In L. Leal-Taixé & S. Roth (Eds.), *Computer Vision – ECCV 2018 Workshops* (Vol. 11131, pp. 138–153). Springer International Publishing. https://doi.org/10.1007/978-3-030-11015-4_13

Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, *5*(2), 157–166. https://doi.org/10.1109/72.279181

Bochkovskiy, A., Wang, C.-Y., & Liao, H.-Y. M. (2020). YOLOv4: Optimal Speed and Accuracy of Object Detection. *ArXiv:2004.10934 [Cs, Eess]*. http://arxiv.org/abs/2004.10934

Chen, B. X., & Tsotsos, J. K. (2019). Fast Visual Object Tracking with Rotated Bounding Boxes. *ArXiv:1907.03892 [Cs]*. http://arxiv.org/abs/1907.03892

Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *ArXiv:1406.1078 [Cs, Stat]*. http://arxiv.org/abs/1406.1078

Deng, J., Dong, W., Socher, R., Li, L., Kai Li, & Li Fei-Fei. (2009). ImageNet: A large-scale hierarchical image database. *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 248–255. https://doi.org/10.1109/CVPR.2009.5206848

Elman, J. L. (1990). Finding Structure in Time. *Cognitive Science*, *14*(2), 179–211. https://doi.org/10.1207/s15516709cog1402_1

Gatti, F. (2020). *Ceccocats/tkDNN* [C++]. https://github.com/ceccocats/tkDNN (Original work published 2017)

Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, *9*(8), 1735–1780.

https://doi.org/10.1162/neco.1997.9.8.1735

Ionescu, C., Papava, D., Olaru, V., & Sminchisescu, C. (2014). Human3.6M: Large Scale Datasets and Predictive Methods for 3D Human Sensing in Natural Environments. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *36*(7), 1325–1339. https://doi.org/10.1109/TPAMI.2013.248

Kane, G., Lopes, G., Saunders, J. L., Mathis, A., & Mathis, M. W. (2020). Real-time, low-latency closed-loop feedback using markerless posture tracking. *BioRxiv*, 2020.08.04.236422. https://doi.org/10.1101/2020.08.04.236422

Labbe, R. (2020). *Rlabbe/filterpy* [Python]. https://github.com/rlabbe/filterpy (Original work published 2014)

Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., & Zitnick, C. L. (2014). Microsoft COCO: Common Objects in Context. In D. Fleet, T. Pajdla, B. Schiele, & T. Tuytelaars (Eds.), *Computer Vision – ECCV 2014* (pp. 740–755). Springer International Publishing. https://doi.org/10.1007/978-3-319-10602-1_48

Martinez, J., Black, M. J., & Romero, J. (2017). On human motion prediction using recurrent neural networks. *ArXiv:1705.02445 [Cs]*. http://arxiv.org/abs/1705.02445

Mathis, A., Mamidanna, P., Cury, K. M., Abe, T., Murthy, V. N., Mathis, M. W., & Bethge, M. (2018). DeepLabCut: Markerless pose estimation of user-defined body parts with deep learning. *Nature Neuroscience*, *21*(9), 1281–1289. https://doi.org/10.1038/s41593-018-0209-y

Nourizonoz, A., Zimmermann, R., Ho, C. L. A., Pellat, S., Ormen, Y., Prévost-Solié, C., Reymond, G., Pifferi, F., Aujard, F., Herrel, A., & Huber, D. (2020). EthoLoop: Automated closed-loop neuroethology in naturalistic environments. *Nature Methods*, *17*(10), 1052–1059. https://doi.org/10.1038/s41592-020-0961-2

Pan, S. J., & Yang, Q. (2010). A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering*, *22*(10), 1345–1359. https://doi.org/10.1109/TKDE.2009.191

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to Sequence Learning with Neural Networks. *ArXiv:1409.3215 [Cs]*. http://arxiv.org/abs/1409.3215

Tan, M., & Le, Q. V. (2020). EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. *ArXiv:1905.11946 [Cs, Stat]*. http://arxiv.org/abs/1905.11946

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention Is All You Need. *ArXiv:1706.03762 [Cs]*. http://arxiv.org/abs/1706.03762

Werbos, P. J. (1990). Backpropagation through time: What it does and how to do it. *Proceedings of the IEEE*, *78*(10), 1550–1560. https://doi.org/10.1109/5.58337

Wijeyakulasuriya, D. A., Eisenhauer, E. W., Shaby, B. A., & Hanks, E. M. (2020). Machine learning for modeling animal movement. *PLOS ONE*, *15*(7), e0235750. https://doi.org/10.1371/journal.pone.0235750

Yoo, S. B. M., Tu, J. C., Piantadosi, S. T., & Hayden, B. Y. (2020). The neural basis of predictive pursuit. *Nature Neuroscience*, *23*(2), 252–259. https://doi.org/10.1038/s41593-019-0561-6