Pet Nose Localization with SnoutNet

ELEC 475

Lab 2

October 22nd, 2024

Ryan Zietlow 20347719

Simon John 20348233

Network Architecture

The SnoutNet architecture in this model is a Convolutional Neural Network that processes images with the dimensions 3x227x227, where the height and width are both 227 pixels and there are 3 channels for RGB, and outputs a two-dimensional vector representing the (x, y) coordinates of the pet's nose.

The input layer of the CNN takes images with pixel dimensions 227x227 and 3 channels for RGB format. These images are then fed into the network's first convolutional layer as tensors of the shape (B, 3, 227, 227), where B is the batch size.

The first three layers of the neural network are made up of a convolution operation, application of the ReLu function, and max pooling. After the convolutional layers, the tensor is flattened from (B, 256, 4, 4) to (B, 1, 4096). Once flattened the tensor is passed through three fully connected layers, ReLu is applied after each FC layer.

The first convolutional layer takes the output from the input layer which is a batch of images with the shape (B, 3, 227, 227). The output of the first layer must be of shape (B, 64, 57, 57) and the size of the applied kernel must be 3x3x3. The second number of the output shape indicates the number of distinct kernels to apply to the input. By applying the kerel of size 3x3x3 on the 3x227x227 input with 1 pixel added as padding and a stride length of 2, the output of the convolution is 1x114x114. As ReLu will not change the size of the tensor, the max pooling operation must take the size of the tensor from 1x114x114 to 1x57x57. As 114 is exactly double 57, max pooling with a 2x2 window, a stride length of 2, and no padding will half the height and width of the tensor to a size of 57x57. Combining the convolution and max pooling as specified with 64 distinct kernels and the given batch size will result in the desired output shape of (B, 64, 57, 57).

The second layer is also convolutional and should take the tensor from (B, 64, 57, 57) to (B, 128, 15, 15) using a kernel of size 64x3x3, ReLu, and max pooling. With the same logic that was used to create the first convolutional layer it can be found that the convolution operation in the second layer will have 128 kernels, operating with a stride of 2 and 2 pixels of padding. A convolution as specified will take an input tensor from (64, 57, 57) to the shape (128, 30, 30). Once again ReLu will have no effect on the shape. Just like in the first layer the width and height of the tensor are exactly double the desired magnitude. To half the width and height of the tensor max

pooling is implemented using a window of 2x2, a stride of 2, and zero padding. When implemented as described this layer will change the size of a tensor from (B, 64, 57, 57) to (B, 128, 15, 15).

The third convolutional layer will take the tensor from (B, 128, 15, 15) to (B, 256, 4, 4) with a kernel size of 128x3x3. The convolution in the third layer uses 256 kernels with a stride length of 2 and 1 pixel of padding. With the settings as specified the convolution will produce an output tensor of shape (1x8x8) or (256x8x8) when considering the number of kernels. Just like the max pooling of the third layer uses a 2x2 window with a stride of 2 and no padding to change the size of the tensor to 256x4x4. If created as instructed then this layer will take in tensor of shape (B, 128, 15, 15) and will output a tensor of shape (B, 256, 4, 4).

As outlined earlier, the tensor is flattened to the shape (B, 1, 4096).

The tensor is then passed through three fully connected layers with ReLu applied after each layer.

The first layer takes in a tensor of shape (1, 4096), and outputs a tensor of shape (1, 1024). The second layer keeps the shape at (1, 1024). The third layer takes the shape from (1, 1024) to (1, 2). Keep in mind that multiple batches are working in parallel so the shape of the tensor will have an additional dimension of a size matching the chosen batch size.

If the neural network is implemented as described above then it will take in an input of size (B, 3, 227, 227) where B is the batch size, and will output a tensor of size (B, 1, 2) containing x and y coordinates of the pictured animals.

Dataset Implementation

To implement the custom dataset, a PetNoseDataset class was designed to load the images and their corresponding labels. We defined an initialization function to store the pets images, the path to the label file and any transforms that will be applied to the images. In this function *img_dir* points to where the images are stored, and *label_file* contains the ground truth labels of the nose coordinates for each image. The *_read_labels* file function reads the label and extracts the corresponding images and nose coordinates and stores them as a tuple in the form *(img_name, nose_coords).* This function reads the label file line by line, if the line is valid it extracts the image name and nose coordinates and converts the coordinates from string form to actual

cooridnate values. If any issues arise during the label reading it outputs the error, skips the line and moves to the next line. When the function is complete, it outputs a list of tuples where each tuple contains the image name and its noses coordinates. The *__len__* function outputs the number of images and labels in the dataset by printing the length of the samples. The *__getitem__* function loads an image from the disk using the PIL library, and rescales the noses coordinates to match the 227x227 image dimensions since all images are resized before entering the network. It then applies any transformations if provided and returns a tuple containing a transformed image as a tensor and the scaled nose coordinates as a tensor. The *custom_collate_fn* is present to handle any images that fail to load. If any samples output None it gets filtered out and stacks the remaining images into batches using *torch.stack()*.

To load the data for the training dataset, the *get_train_loader()* function initializes the PetNoseDataset using training images and their corresponding labels and applies the necessary transformations. It then returns the the images and labels in batches of the requested size. To load the data for the testing dataset, the *get_test_loader()* function performs the same steps as the training loader except it does not perform any augmentations on the data.

To confirm the dataset and data loaders are functioning as required a *reality_check()* function is defined to quickly inspect samples of data. It iterates through batches of images and labels in the data loader and prints out the shape of the image and label batches to ensure they are correctly shaped. For each image in the batch, it overlays a red dot on the pet's nose using the ground truth coordinates to see if the labels align with the images content. If any images fail to load due to missing files or corruption they are skipped, and the error is reported in the terminal.

Hyperparameters and Training

The learning rate determines the size of the steps taken towards the minimum of the loss function during optimization which dictates how fast or slow the model will learn from its mistakes. The Adam optimizer was used for this, which adapts the learning rate for each parameter based on the first and second moments of the gradients. The first moment (mean) is the running average of the gradients and the second moment (variance) is the running average of the squared gradients. The learning rate of this optimizer was 1e-3 which is a standard balanced choice that provides fast convergence and stability.

The learning rate schedule is a strategy that adjusts the learning rate during the training of the model by controlling how much the models' weights change with respect to the loss gradient during each iteration of training. This model uses a ReduceLROnPlateau scheduler which reduces the learning rate when a performance metric stops improving significantly and hits a plateau, meaning the model is struggling to improve its performance with the current learning rate. When the scheduler detects this plateau, it reduces the learning rate by 0.1 to let it fine tune its performance.

Weight decay is a regularization technique that attempts to prevent overfitting by adding a penalty term to the loss function to discourage large weights. The weight decay in this model is set to 1e-5.

The batch size of the model refers to the number of training examples in one iteration of the training process. In this model the batch size is set to 600. The number of epochs is the number of full complete passes of the entire training dataset through the model. This model uses 50 epochs.

The loss function is a computation of the error between the prediction and target values of the model. The goal is to try and minimize this loss. In this model, the loss function is Mean Squared Error (MSE): $MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y} - y)^2$, where $\hat{y}$ is the predicted value for the i-th data point and y is the ground truth value for the i-th data point. The MSE loss function is useful for the PetNoseDataset because the model predicts continuous values, and it is trying to minimize the distance between predicted and true continuous values.

This model was run on a NVIDIA 4070 Super GPU. Training the model for 50 epochs with a batch size of 600 took approximately 18 seconds per epoch, and 15 minutes to complete the entire training dataset.

The loss plots seen below are for each augmentation case respectively. As you can see the model learns a lot in the first two epochs and then slowly learns throughout the other 48 as it plateaus.

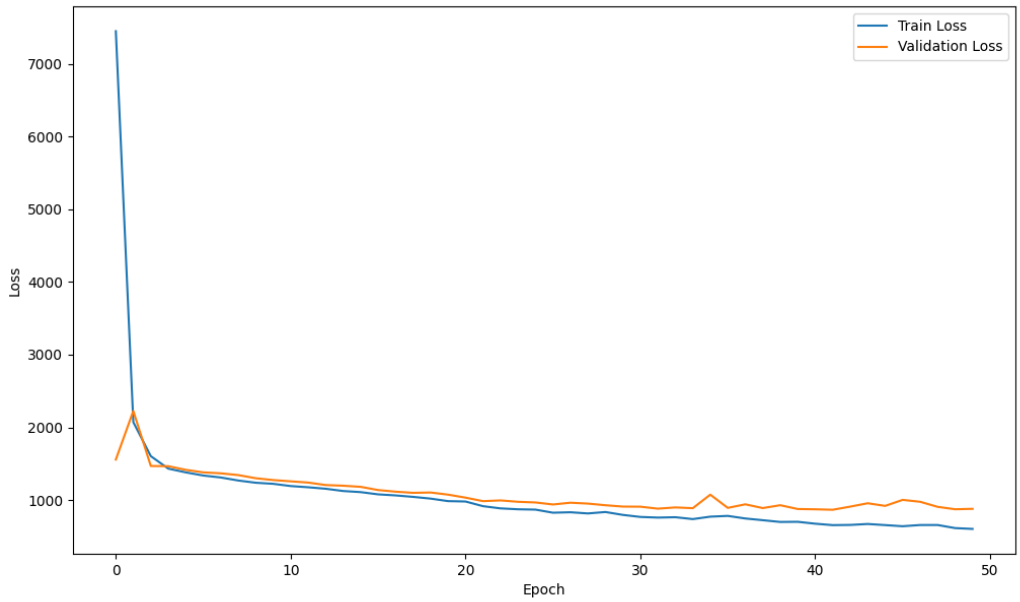

Figure 1: Loss plot for training of No Augment model

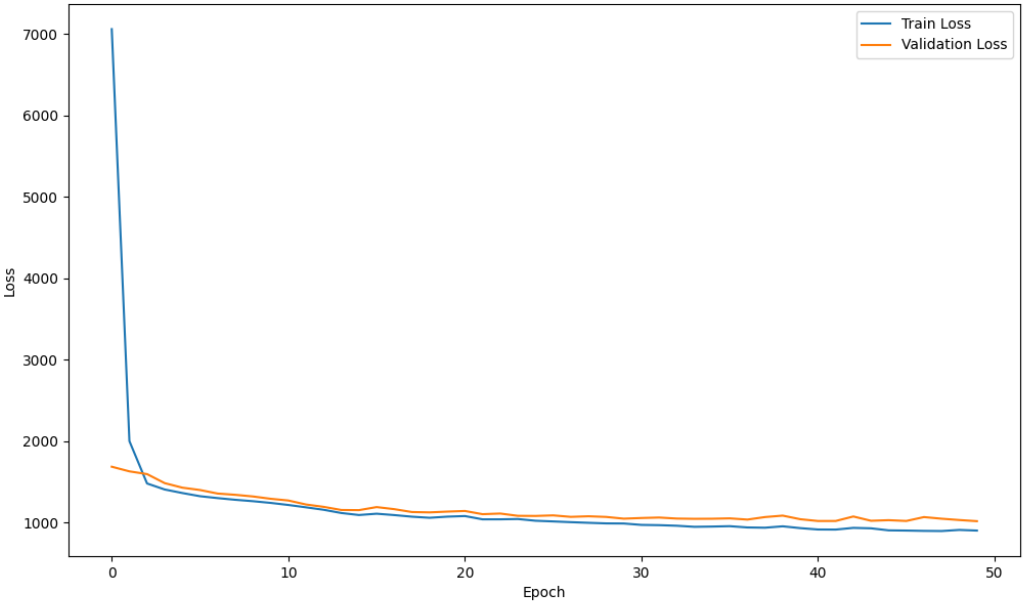

Figure 2: Loss plot for training of Flip Augment model

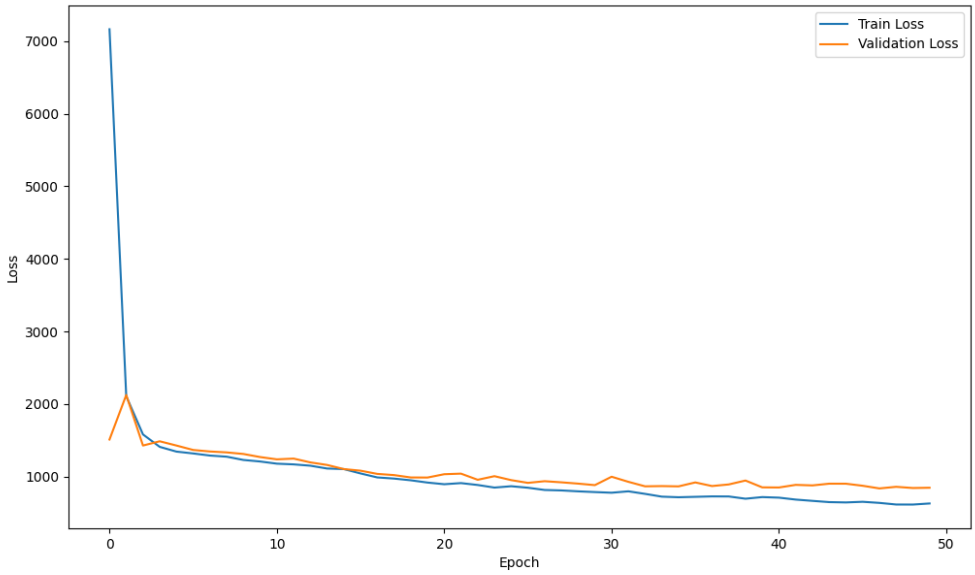

Figure 3: Loss plot for training of Saturated Augment model

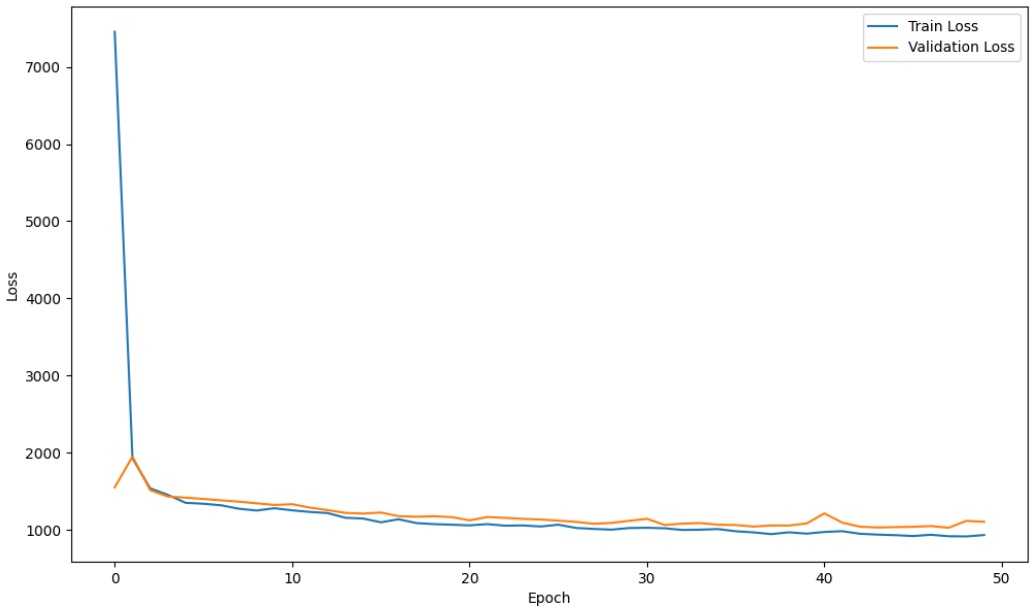

Figure 4: Loss Plot for training of All Augments model

Data Augmentation

Two augmentations methods were incorporated into the model to improve its generalization to different variations of input images. The first augmentation is a random horizontal flip that has a probability of occurrence set to 50%. This assists the model with learning that a pets nose can be on either side of the image regardless of its orientation. This was implemented in the code through an if statement. It checks if the transform type is a horizontal flip, and if it is it returns a randomly flipped image and then applies the base resizing transform. The second augmentation is a color jitter (saturation adjustment). This augmentation randomly adjusts the saturation of the image's colors with a factor of 0.5. It allows the model to learn under various lighting conditions and color intensities since they can vary due to different picture environments or camera settings. This was implemented in the code through an if statement that checks if transform type is set to saturation, and if it is it applies a saturation jitter intensity between 1 +/- 0.5, so between 0.5 and 1.5, and then a base transform to resize the image is applied at the end. After none, one or both augmentations are applied to the image, there is a standard transform applied to the image. This includes the image being resized to the required 227x227 to maintain data consistency, and then it is converted to a PyTorch tensor and its finally normalized based on the ImageNet mean and standard deviation to speed up training and stabilize the learning process.

Experimentation

As mentioned previously the model was trained on an NVIDIA 4070 Super GPU in a system with 32GB of DDR5 RAM at a speed of 6000MHz, and an AMD 7800x3D CPU with 8 cores and 16 threads. The GPU contains 7168 CUDA (Compute Unified Device Architecture) cores which are the basic processing units in NVIDIA GPUs that are optimized for parallel processing which enables the GPU to execute multiple threads at the same time. Since CNNs involve many matrix and tensor operations, these CUDA cores allow them to operate in parallel which speeds up the model training process. The GPU also contains 568 4th generation AI TOPS Tensor Cores which are specialized hardware units that are designed specifically for deep learning tasks. These cores allow models to compute calculations with reduced precision without sacrificing accuracy leading to faster computations and lower memory usage. This can be done because the tensor cores support mixed precision training. Furthermore, tensor cores are designed to perform matrix multiplications more efficiently than CUDA cores, leading to faster training times. The GPU

uses both types of cores in parallel, CUDA for general purpose computations and tensor for matrix multiplications.

In table 1 below, the training time for each augmentation is displayed. As seen in the table, no augmentation takes the second shortest amount of time which makes sense as no augmentations are performed besides the standard resizing. The saturated training time does take the least amount of time which is a bit odd but the conclusion we can draw is that it's not a computationally complex augment and it's just a rare occurrence in the training times where this one happened to be longer. Performing all the augmentations took the longest, which again makes sense since it's more computationally intense to saturate and flip the image and then resize it.

Table 1: Training times of the models

Augmentation	**Training Time**
No Augment	13 minutes and 15 seconds
Flip	13 minutes and 37 seconds
Saturate	13 minutes and 5 seconds
Flip and Saturate	13 minutes and 52 seconds

In table 2 below, the average time taken to process each image during testing is displayed. No augmentation is the fastest time which makes sense since no augments are being performed. It takes slightly longer to perform the flip and saturate augments meaning there's little overhead in the image processing. Performing all augmentations is comparable to using either augmentation alone indicating that the increase in complexity does not affect the processing time per image on average.

Table 2: Average time per image during testing

Augmentation	**Average Time per Image (msec)**
No Augment	0.327
Flip	0.333
Saturate	0.337
Flip and Saturate	0.334

In table 3 below, the minimum distance, maximum distance and mean distance of the predicted values to the actual nose values are displayed for each augment case. The minimum distance is the smallest distance between the predicted nose location and actual nose location across all

predictions. The maximum distance is the largest distance between the predicted nose location and actual nose location across all predictions. The mean distance is the average distance between the predicted nose location and actual nose location across all predictions. When no augments are performed on the model it performs worse in every category across the table which should be the case since augments allow the model to improve generalization. Flip and saturate both improve all the categories across the table, but as you can see flip performs better across the board. Performing both augments on the image drastically increases performance from all other cases which makes sense with the reasoning explained earlier regarding generalization.

Table 3: Distance statistics of the testing of the models

Augmentation	Minimum Distance	Maximum Distance	Mean Distance
No Augment	12.23	272.34	118.93
Flip	3.66	189.89	68.80
Saturate	6.18	208.21	89.93
Flip and Saturate	0.86	157.20	50.84

Extending the conclusions from table 3 above, table 4 below displays more accuracy and error statistics on the augmentation cases. Standard deviation is a measure of how spread out the distances are from the mean distance; a lower standard deviation indicates that the distances are closer to the mean meaning better performance. Mean absolute error is the average of the absolute differences between the predicted nose locations and the actual nose locations. It does not consider the direction of the error (high or low, left or right) just the absolute value of the distance. A lower MAE means better performance. The root mean squared error is the square root average of the squared differences between the predicted values and actual values. RMSE gives more weight to larger errors since they are squaring the difference of the values. A lower RMSE indicates better performance. As seen in the results below, no augment performs the worst across the board. Flip and saturate both improve results, but flip ends up performing slightly better which aligns with the conclusions pulled from table 3 above. Performing both augmentations on the images drastically increased the error statistics across the board which aligns with conclusions pulled from the above tables.

Table 4: accuracy and error statistics of the testing of the models

Augmentation	Standard Deviation	Mean Absolute Error	Root Mean Square Error

No Augment	46.64	77.31	90.33
Flip	34.04	44.25	54.28
Saturate	36.57	55.21	68.58
Flip and Saturate	27.51	32.71	40.87

Displayed below are images for the best- and worst-case testing results from each augmentation model. As seen below, the best and worst case performs better as you augment the image more, and flip is still outperforming the saturate augmentation. The conclusion we can draw from this is that the model would get better outputs if we performed multiple flip operations such as one horizontal, one vertical and one 45-degree turn, instead of using saturation.

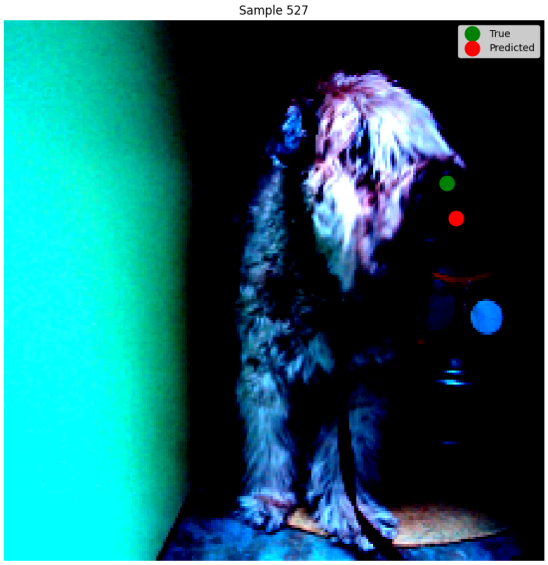

Figure 5: Best performance of No Augment model

Figure 6: Worst performance of No Augment model

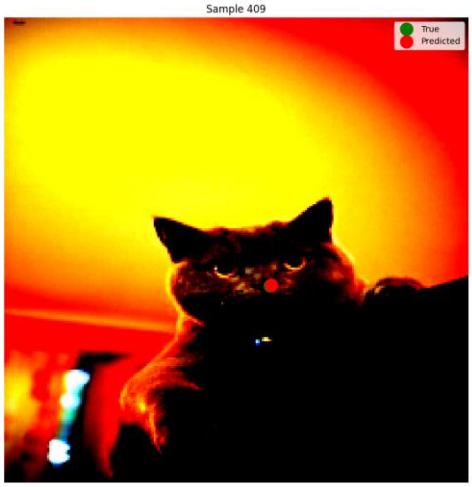

Figure 7: Best performance of Flip Augment model

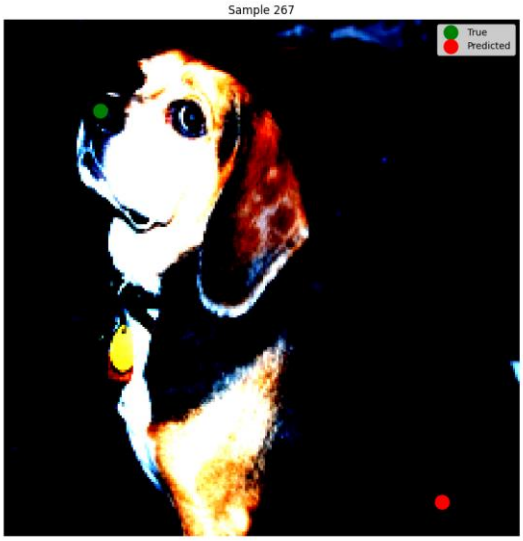

Figure 8: Worst performance of Flip Augment model

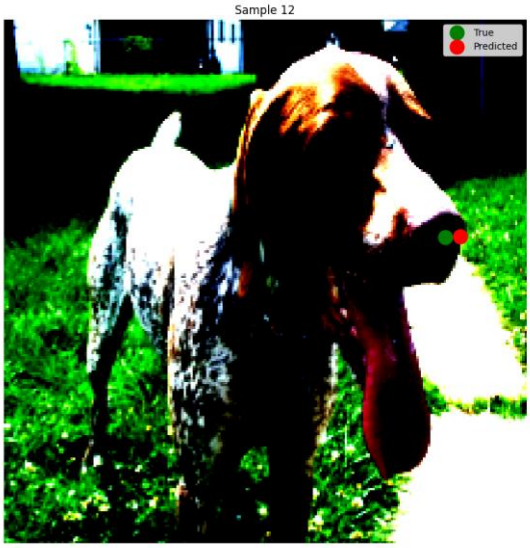

Figure 9: Best performance of Saturation Augment model

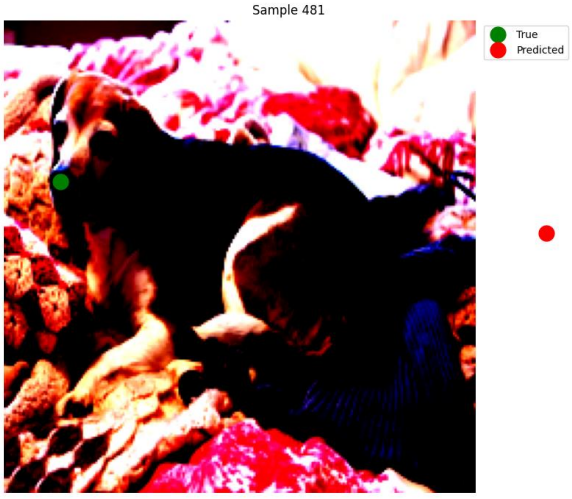

Figure 10: Worst performance of Saturation Augment model

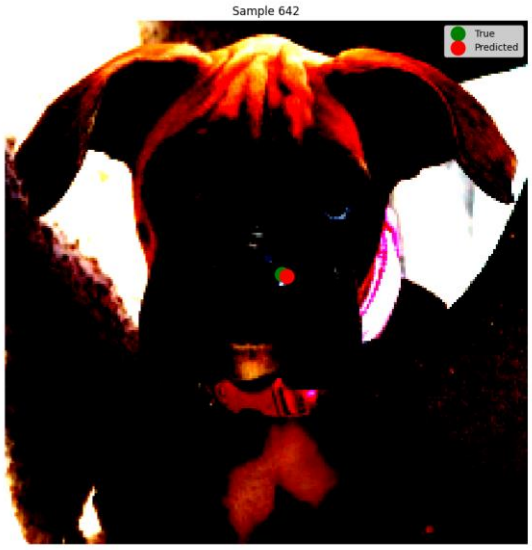

Figure 11: Best performance of All Augment model

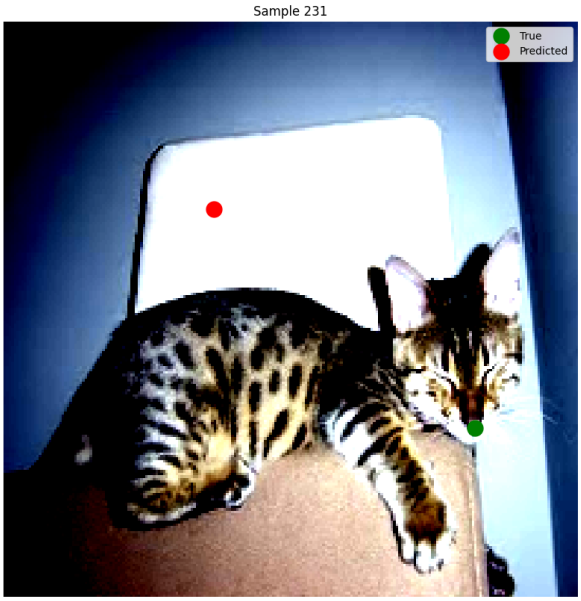

Figure 12: Worst performance of All Augment model

Performance of the System

The system seems to be able to find the nose of the animal in some cases and usually has a guess better than a random choice. As seen in Table 3 the model with both augmentations was within 50 pixels of the nose on average. Just flipping the image or just increasing the saturation seemed to positively influence the performance of the system as the results of those models were better than the model with no augmentations. Applying both augmentations at the same time further improved the results from either individual augmentation.

Flipping the image horizontally seemed to have a greater effect on the results compared to increasing saturation, this is represented in the results shown in Tables 3 and 4.

When creating the dataloader, if the image underwent a transformation that changed either its size or orientation it was difficult to ensure that the label for the correct nose position also underwent the same transformations.