ELEC 475 – Lab 02

Department of Electrical and Computer Engineering
Queen's University

Composed By
Nicholas Seegobin (20246787)

Date of Submission
2024 October 22

"I hereby attest that the work submitted is my own individual work and that no portion of this submission has been copied in whole or in part from another source, with the possible exception of properly referenced material."

Introduction

The objective of this lab is to implement SnoutNet, a custom convolutional neural network (CNN), to localize pet noses in images. The goal is to estimate the location of the nose by predicting UV (x, y) pixel coordinates within the image frame. To achieve this, the oxford-iiit-pet-noses dataset is utilized, which contains reannotated images from the original oxford-iiit-pets dataset, with each image labeled by the precise coordinates of the pet's nose. The task requires training the network to regress these coordinates, which can then be compared to the ground truth for performance evaluation.

The SnoutNet architecture, as shown in Figure 1, consists of several convolutional and fully connected layers. The convolutional layers focus on extracting high-level features from the input images, such as edges and textures, while the fully connected layers map these features to the predicted nose locations. The input to the model is a 227 × 227 RGB image, flattened before passing through the fully connected layers. The network is trained using regression loss, which minimizes the distance between predicted and actual nose locations.

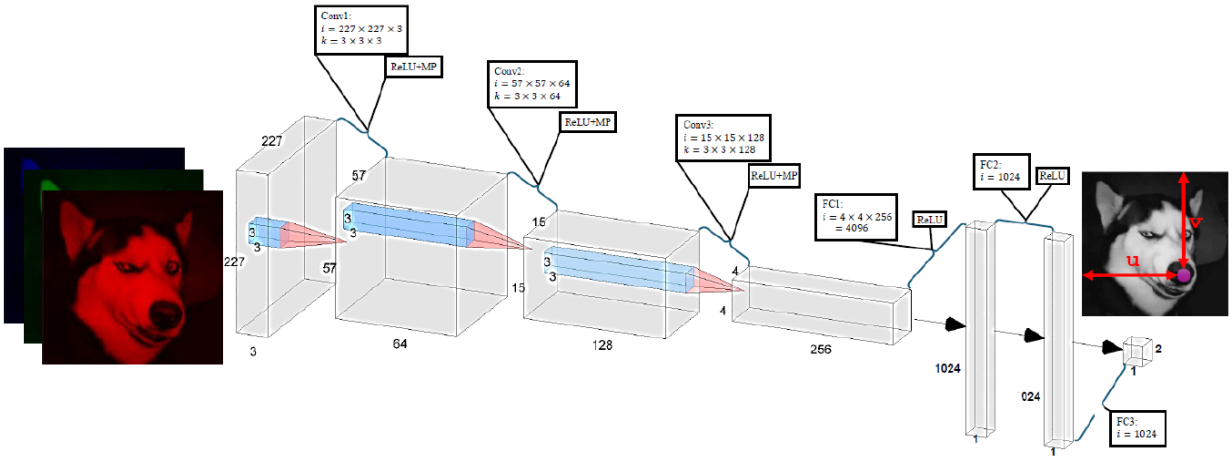

Figure 1: A diagram of the SnoutNet architecture used in the lab.

Throughout the lab, various data augmentation techniques are applied to improve the model's generalization ability, including horizontal flipping and color jitter. The impact of these augmentations is analyzed by comparing the localization error metrics, such as mean, minimum, maximum, and standard deviation of the Euclidean distance between predicted and actual nose locations, across different training configurations.

Part 1: Model Details

The SnoutNet architecture, implemented for the task of localizing pet noses in images, follows a convolutional neural network (CNN) structure. This architecture is specifically designed to predict the uv-coordinates of a pet's nose by extracting relevant spatial features from input images and processing them through a series of convolutional, max-pooling, and fully connected layers. The input to the network is an RGB image of size 3 × 227 × 227.

The convolutional layers are responsible for feature extraction. Each layer applies a set of filters that detect local patterns such as edges and textures in the image. These layers are followed by max-pooling layers, which progressively reduce the spatial resolution of the feature maps while retaining the most

important information. This combination allows the network to focus on high-level features while making the network more efficient by reducing the number of computations.

Once the feature maps have passed through the convolutional and pooling layers (see Figure 2), they are flattened into a one-dimensional vector. This flattening operation converts the two-dimensional spatial data into a format that can be processed by fully connected layers, which act as the decision-making part of the network. The fully connected layers, seen in Figure 3, combine the features extracted by the convolutional layers to predict the u and v coordinates of the pet's nose. These coordinates correspond to the predicted location of the nose within the image.

The first convolutional layer, Conv1, applies 64 filters of size 3 × 3 to the input image, with stride 2 and padding 2, producing a feature map of size 64 × 114 × 114. This layer is followed by a ReLU activation function to introduce non-linearity. MaxPool1 reduces the spatial dimensions to 64 × 57 × 57 by applying a max-pooling operation with a kernel size of 2 and a stride of 2.

The second convolutional layer, Conv2, takes the output from Conv1 and applies 128 filters of size 3 × 3, with stride 2 and padding 2, producing a feature map of 128 × 30 × 30. After ReLU activation, MaxPool2 reduces the size to 128 × 15 × 15 using a kernel size of 2 and a stride of 2.

The third convolutional layer, Conv3, applies 256 filters of size 3 × 3 with stride 2 and padding 2, generating a feature map of 256 × 9 × 9. MaxPool3 then reduces the size to 256 × 4 × 4 using a kernel size of 2 and a stride of 2. After this layer, the feature map is flattened into a one-dimensional vector to be processed by the fully connected layers.

```
# Conv1: 64 filters, 3x3 kernel, stride 2, padding 2 -> output: 64 channels, 114x114
self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=2, padding=2)
self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)  # MaxPool -> 57x57

# Conv2: 128 filters, 3x3 kernel, stride 2, padding 2 -> output: 128 channels, 30x30
self.conv2 = nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=2)
self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)  # MaxPool -> 15x15

# Conv3: 256 filters, 3x3 kernel, stride 2, padding 2 -> output: 256 channels, 9x9
self.conv3 = nn.Conv2d(128, 256, kernel_size=3, stride=2, padding=2)
self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)  # MaxPool -> 4x4
```

Figure 2: A snippet of model.py showing the convolutional and pooling layers.

The first fully connected layer, FC1, receives the flattened vector of size 256 × 4 × 4 = 4096 and maps it to 1024 neurons. A ReLU activation is applied to introduce non-linearity. The second fully connected layer, FC2, retains 1024 neurons, continuing the processing of features with another ReLU activation. Finally, the third fully connected layer, FC3, reduces the output to 2 neurons, representing the predicted u and v coordinates of the pet's nose.

```
# Fully connected layers
self.fc1 = nn.Linear(256 * 4 * 4, 1024)  # FC1: input 4096 (4x4x256), output 1024
self.fc2 = nn.Linear(1024, 1024)         # FC2: input 1024, output 1024
self.fc3 = nn.Linear(1024, 2)            # FC3: input 1024, output 2 (for u, v coordinates)
```

Figure 3: A snippet of model.py showing the fully connected layers.

The init() method initializes the SnoutNet model by defining all of these layers. It specifies the convolutional, pooling, and fully connected layers, setting the architecture of the network. The forward() method defines the forward pass, which outlines how the input data flows through the network. The input image is first passed through the convolutional layers, followed by pooling operations, then flattened and processed by the fully connected layers. The final output is the predicted coordinates of the pet's nose. An image of the init() and forward() methods can be seen in Figure 4 to visually reference the structure and flow of the model.

```
class SnoutNet(nn.Module):
    def __init__(self):
        super(SnoutNet, self).__init__()

        # Conv1: 64 filters, 3x3 kernel, stride 2, padding 2 -> output: 64 channels, 114x114
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=2, padding=2)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)  # MaxPool -> 57x57

        # Conv2: 128 filters, 3x3 kernel, stride 2, padding 2 -> output: 128 channels, 30x30
        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=2)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)  # MaxPool -> 15x15

        # Conv3: 256 filters, 3x3 kernel, stride 2, padding 2 -> output: 256 channels, 9x9
        self.conv3 = nn.Conv2d(128, 256, kernel_size=3, stride=2, padding=2)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)  # MaxPool -> 4x4

        # Fully connected layers
        self.fc1 = nn.Linear(256 * 4 * 4, 1024)  # FC1: input 4096 (4x4x256), output 1024
        self.fc2 = nn.Linear(1024, 1024)         # FC2: input 1024, output 1024
        self.fc3 = nn.Linear(1024, 2)            # FC3: input 1024, output 2 (for u, v coordinates)

    def forward(self, x):
        # Apply convolutions, ReLU, and pooling
        x = self.pool1(F.relu(self.conv1(x)))  # Conv1 -> Pool1, Output: [batch_size, 64, 57, 57]
        x = self.pool2(F.relu(self.conv2(x)))  # Conv2 -> Pool2, Output: [batch_size, 128, 15, 15]
        x = self.pool3(F.relu(self.conv3(x)))  # Conv3 -> Pool3, Output: [batch_size, 256, 4, 4]

        # Flatten for fully connected layers using reshape
        x = x.reshape(x.size(0), -1)  # Flatten to (batch_size, 4096)

        # Apply fully connected layers
        x = F.relu(self.fc1(x))  # FC1: Output: [batch_size, 1024]
        x = F.relu(self.fc2(x))  # FC2: Output: [batch_size, 1024]
        return self.fc3(x)  # FC3: Output: [batch_size, 2] (u, v coordinates)
```

Figure 4: A snippet showing the entire model.py file which includes the init and forward functions.

Part 2: Custom Dataset

The custom dataset was implemented in the custom_dataset.py file to handle the loading and processing of pet images and their associated nose coordinates for the SnoutNet model. The images were stored in the directory data\oxford-iiit-pet-noses\images-original\images, while the nose coordinates were contained in the annotations file data\oxford-iiit-pet-noses\train_noses.txt, which listed each image's filename along with its corresponding uv-coordinates.

The PetNoseDataset class handles reading images and labels, applying transformations, and returning processed data for the DataLoader. The class begins with the __init__ method, seen in Figure 5, which sets the paths to the images and annotations and loads the annotations into memory. These annotations are stored in a pandas DataFrame for easy access, with each image file being paired with its corresponding nose coordinates. The __init__ method also defines optional data augmentations, such as horizontal flip and color jitter, which can be applied to the images based on specific flags set by the user.

```
class PetNoseDataset(Dataset):
    def __init__(self, annotations_file, img_dir, apply_flip=False, apply_color_jitter=False):
        self.img_labels = pd.read_csv(annotations_file, names=["filename", "coordinates"])
        self.img_dir = img_dir
        self.apply_flip = apply_flip
        self.apply_color_jitter = apply_color_jitter
        self.resize_transform = transforms.Resize((227, 227))  # Resize to the correct size for SnoutNet

        # Define individual transforms (but they will only be applied based on flags)
        self.flip_transform = transforms.Compose([
            transforms.functional.hflip  # Full horizontal flip
        ])

        self.color_jitter_transform = transforms.Compose([
            transforms.transforms.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1, hue=0.1)
        ])
```

Figure 5: A snippet of code showing the __init__ function inside of the custom_dataset.py file.

A key part of the custom dataset implementation involves resizing the images to a consistent size of 227 x 227 pixels to ensure uniform input dimensions for training. This resizing is crucial since the images in the dataset have different original dimensions, and uniformity is needed for compatibility with the SnoutNet model. Along with resizing the images, the nose coordinates (uv-coordinates) are also scaled proportionally to match the new image dimensions. The scaling process adjusts the x and y coordinates of the nose to ensure that they remain accurate in relation to the resized image.

In the __getitem__ method (see Figure 6), images are loaded from the disk using the image file path, and their labels (nose coordinates) are retrieved from the annotations. The method first ensures that the image is in RGB format and then applies the necessary transformations, including resizing. If augmentation flags are enabled, the image may be flipped horizontally or have color adjustments applied. Additionally, the nose coordinates are adjusted accordingly if a flip is applied (e.g., the x-coordinate is inverted when a horizontal flip occurs). The final image is returned as a tensor, and the coordinates are returned as a tensor of floats.

```python
def __getitem__(self, idx):
    img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
    image = Image.open(img_path)

    # Ensure the image has 3 channels (convert to RGB)
    if image.mode != 'RGB':
        image = image.convert('RGB')

    image = transforms.ToTensor()(image)

    # Store the original width and height of the image
    orig_height, orig_width = image.shape[1], image.shape[2]

    # Resize the image to 227x227
    image = self.resize_transform(image)

    # Get label (coordinates), parse the string, and convert to a list
    label_str = self.img_labels.iloc[idx, 1]
    label_list = label_str.strip("()").split(", ")
    label_list = [int(coord) for coord in label_list]

    # Scale the label coordinates according to the new image size
    x_scale = 227 / orig_width
    y_scale = 227 / orig_height
    label_list[0] = label_list[0] * x_scale  # Scale x-coordinate
    label_list[1] = label_list[1] * y_scale  # Scale y-coordinate

    # Apply the transformations based on the flags set in the constructor
    if self.apply_flip:
        image = self.flip_transform(image)
        label_list[0] = 227 - label_list[0]  # Adjust x-coordinate for flip
    if self.apply_color_jitter:
        image = self.color_jitter_transform(image)

    # Convert label list to a tensor
    label = torch.tensor(label_list, dtype=torch.float32)

    return image, label
```

Figure 6: A snippet of code showing the __getitem__ function inside of the custom_dataset.py file.

To validate that the custom dataset and DataLoader were functioning correctly, a reality check was conducted by loading a few samples from the dataset (see Figure 7). This process involved displaying the images, printing their corresponding nose coordinates, and visually inspecting them. The coordinates were plotted as red "x" markers over the images to confirm that they aligned with the correct locations. This step ensured that the transformations were applied correctly, particularly the resizing of the images and the scaling of the coordinates. This quick fact check validated that the custom dataset was working

as expected and that the data was correctly prepared for input into the SnoutNet model for training and testing.

```python
# Visualization Code
if __name__ == "__main__":
    # File Paths
    annotations_file = 'data/oxford-iiit-pet-noses/train_noses.txt'
    img_dir = 'data/oxford-iiit-pet-noses/images-original/images'

    # Create the dataset
    dataset = PetNoseDataset(annotations_file=annotations_file, img_dir=img_dir)

    # Output first 5 images and their labels
    for i in range(5):
        image, label = dataset[i]
        print(f"Label {i+1}: (x, y) = ({label[0]:.1f}, {label[1]:.1f})")

        # Convert image tensor to a PIL image for visualization
        image_pil = transforms.ToPILImage()(image)  # Convert back to PIL image for display

        # Display the image
        plt.imshow(image_pil)

        # Plot the nose coordinates as an "x" marker
        plt.scatter(label[0], label[1], color='red', marker='x')  # x and y coordinates with a red 'x' marker
        plt.title(f"Label {i+1}: (x, y) = ({label[0]:.1f}, {label[1]:.1f})")

        # Show the plot
        plt.axis('on')
        plt.show()
```

Figure 7: A snippet of code showing the reality check test inside of the custom_dataset.py file.

To view images of the reality check, where sample images and their corresponding nose coordinates were displayed for verification, please refer to the 6.1. Reality Check Images section. The images in the appendix illustrate how the nose coordinates were correctly aligned with the pet noses in the dataset after applying the necessary transformations (image resize, horizontal flip, colour jitter), ensuring the accuracy of the custom dataset implementation.

Part 3: Training Details

The training process for the SnoutNet model involved training 20 different models. The first 16 models were automatically trained using the automate_train.py script (seen in Figure 8), which incorporated a variety of data augmentations such as horizontal flips and color jitter. These augmentations were controlled by flags within the script, and models were trained using different combinations of transformations to improve generalization. For the remaining 4 models, datasets containing only augmented data were used. This approach allowed a more focused evaluation of the impact of the transformations on the model's performance. The entire training process was automated, and hyperparameters were fine-tuned for performance using the Optuna library, which automatically identified the optimal settings.

```python
import subprocess

def run_training(e, b, t, v, d, flip, color_jitter):
    # Base command
    command = f"python train.py --e {e} --b {b} --af {t} --vf {v} --img_dir {d}"

    # Add flags for transformations if applicable
    if flip:
        command += " --hflip"
    if color_jitter:
        command += " --color"

    # Use subprocess to execute the command
    print(f"Executing command: {command}")
    process = subprocess.Popen(command, shell=True)
    process.communicate()  # Wait for the process to complete

if __name__ == '__main__':
    t = r'.\data\oxford-iiit-pet-noses\train_noses.txt'
    v = r'.\data\oxford-iiit-pet-noses\test_noses.txt'
    d = r'.\data\oxford-iiit-pet-noses\images-original\images'

    run_training(e=30, b=128, t=t, v=v, d=d, flip=True, color_jitter=True)
    run_training(e=30, b=128, t=t, v=v, d=d, flip=False, color_jitter=False)
    run_training(e=30, b=128, t=t, v=v, d=d, flip=True, color_jitter=False)
    run_training(e=30, b=128, t=t, v=v, d=d, flip=False, color_jitter=True)

    run_training(e=30, b=64, t=t, v=v, d=d, flip=False, color_jitter=False)
    run_training(e=30, b=64, t=t, v=v, d=d, flip=True, color_jitter=False)
    run_training(e=30, b=64, t=t, v=v, d=d, flip=False, color_jitter=True)
    run_training(e=30, b=64, t=t, v=v, d=d, flip=True, color_jitter=True)

    run_training(e=30, b=32, t=t, v=v, d=d, flip=False, color_jitter=False)
    run_training(e=30, b=32, t=t, v=v, d=d, flip=True, color_jitter=False)
    run_training(e=30, b=32, t=t, v=v, d=d, flip=False, color_jitter=True)
    run_training(e=30, b=32, t=t, v=v, d=d, flip=True, color_jitter=True)

    run_training(e=30, b=16, t=t, v=v, d=d, flip=False, color_jitter=False)
    run_training(e=30, b=16, t=t, v=v, d=d, flip=True, color_jitter=False)
    run_training(e=30, b=16, t=t, v=v, d=d, flip=False, color_jitter=True)
    run_training(e=30, b=16, t=t, v=v, d=d, flip=True, color_jitter=True)
```

Figure 8: A script called automate_train.py written to automatically train 16 models with different parameters.

3.1. Hyperparameters

For training, the default hyperparameters were used as defined in the train.py file. These were optimized using an Optuna-based hyperparameter tuning process, which ran for 8 hours to find the

best-performing parameters. The main hyperparameters identified from the tuning process can be seen in Figure 9.

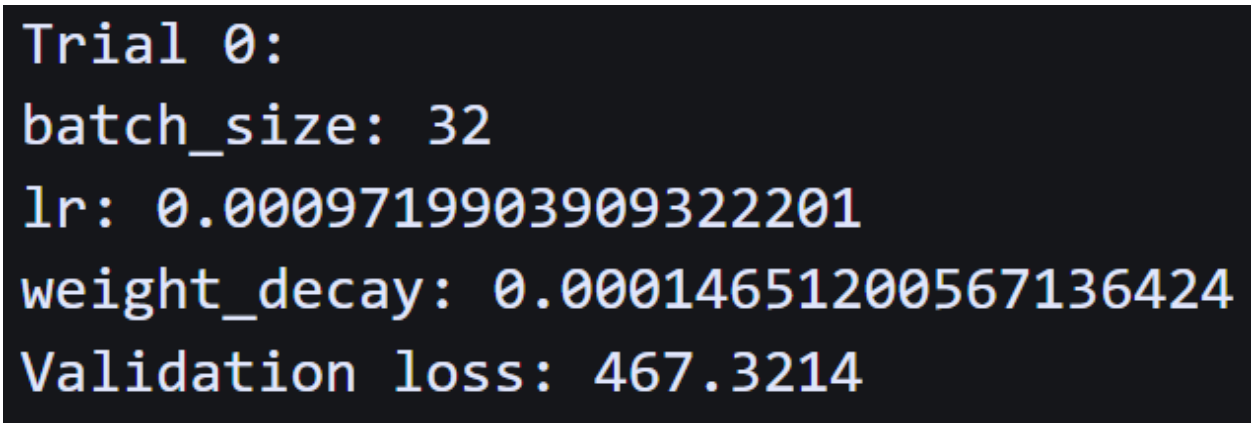

Figure 9: The most optimal parameters found by the Optuna library.

The Optuna trials explored various learning rates, weight decay, and batch sizes to minimize validation loss. The optimal values were selected based on the results of these trials, as recorded in the Optuna output logs.

3.2. Hardware Used

The training process was executed on an RTX 2060 6GB GPU, coupled with an Intel i7 9700 processor and 16 GB DDR4 RAM. This setup allowed efficient parallel processing on the GPU, speeding up the training process. The total training time across all 20 models was approximately 10 hours, with the process automated to run overnight.

3.3. Loss Plots and Training Duration

The training process for the SnoutNet model involved multiple experiments with 20 different models. Throughout the training, loss plots were generated for each model, which provide a visual representation of the training and validation losses over the 30 epochs.

In general, the training loss for all models steadily decreased as the number of epochs increased, reflecting effective learning from the dataset. Meanwhile, the validation loss plateaued after an initial decrease, indicating that the model reached convergence after approximately 10 epochs. This pattern was consistent across different training configurations, which included variations in batch size, data augmentation techniques, and combinations of transformations such as horizontal flipping and color jitter.

Each experiment's plot reveals specific behaviors influenced by the training settings. For instance, some models using larger batch sizes or more extensive augmentations tended to show slower initial improvements in validation loss, but overall reached lower final losses. The models trained with a batch size of 32 generally displayed smoother curves and better generalization compared to smaller batch sizes.

The final average training loss for some models, such as the one highlighted in Figure 10, was as low as 17.5, with a mean localization error of around 18.2 pixels. The total runtime of training per model varied significantly based on batch size and augmentation settings, with runtimes ranging from around 13

minutes for models with smaller batch sizes, to upwards of 46 minutes for models using larger batch sizes or more complex augmentation strategies.

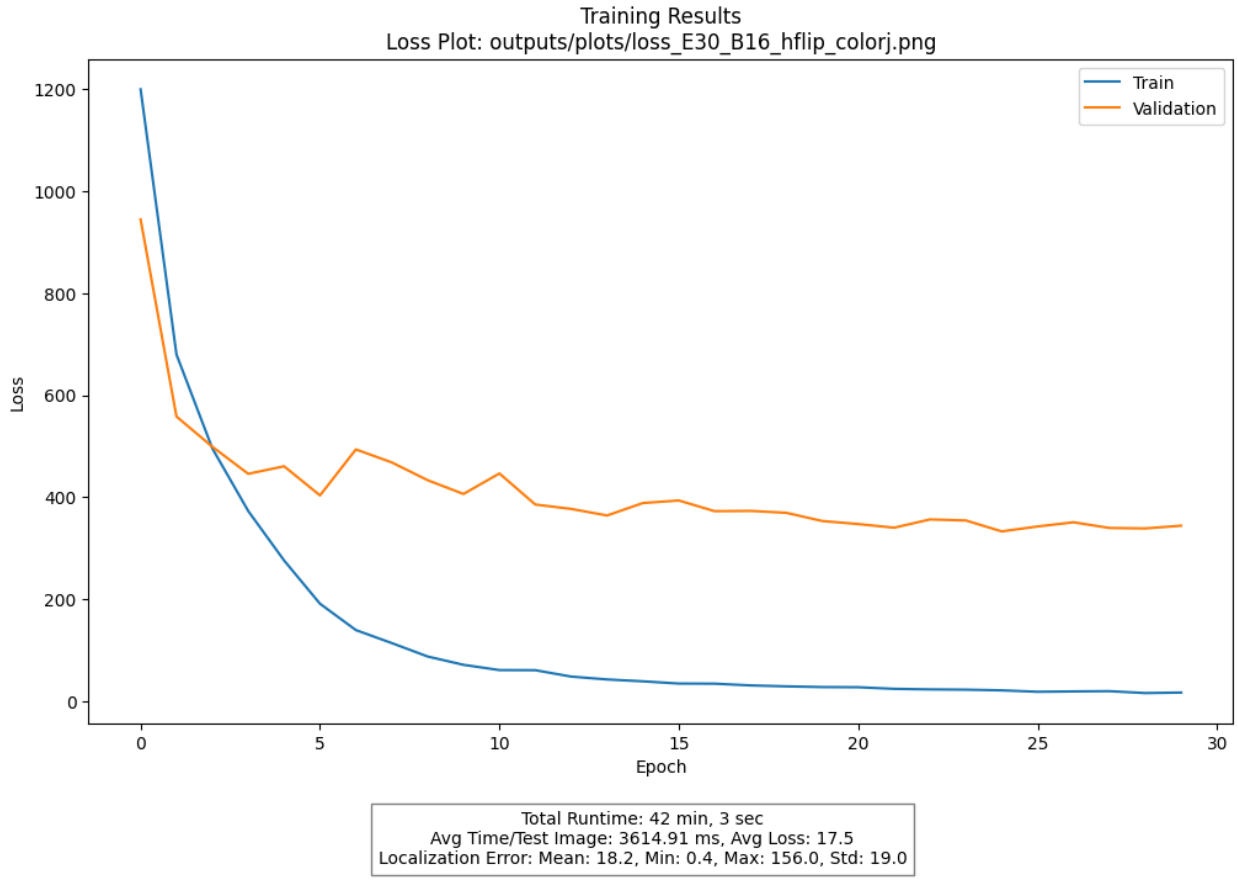

Figure 10: A plot showing the training results of a model that ran for 30 epochs with a batch size of 16, all transforms applied.

For a complete visual reference of these loss plots, including detailed comparisons between models trained with and without augmentations, as well as different batch sizes, refer to section 6.3. Loss Plots in the appendix, where all 20 plots are displayed.

Part 4: Data Augmentation

The data augmentation methods implemented for training the SnoutNet model include horizontal flipping and color jitter. Horizontal flipping mirrors the image across the vertical axis, effectively creating a flipped version of the original image. This augmentation helps the model become more robust by exposing it to different orientations of the same data. In real-world scenarios, objects can appear in different orientations, and flipping helps the model generalize better to such variations, especially when identifying pet noses, where orientation can vary.

Color jitter, on the other hand, introduces random variations in the image's brightness, contrast, saturation, and hue. This transformation simulates different lighting conditions, making the model less sensitive to such changes. By exposing the model to various visual environments, color jitter ensures that the model doesn't overfit to specific lighting conditions in the training data, enhancing its adaptability to diverse lighting situations in real-world applications. Both of these augmentation techniques expand the dataset artificially, introducing variations that improve the model's ability to generalize to unseen data.

Part 5: Experiment Results and Model Performance

5.1. Experiment Results

The experiments conducted in this lab were designed to evaluate the performance of the SnoutNet model under various data augmentation configurations. The experiments were carried out using an NVIDIA RTX 2060 6GB GPU and an Intel i7 9700 processor, with 16GB DDR4 RAM. This hardware setup allowed for efficient model training and testing across multiple configurations. Each experiment involved different combinations of horizontal flip and color jitter augmentations to assess their impact on model accuracy and runtime performance.

The data provided in Table 1 illustrates an ablation study, where different augmentation techniques were applied to assess their effect on localization accuracy. The key metric used for evaluation was the localization error, which measures the Euclidean distance between the predicted nose coordinates and the actual ground truth. The results of each experiment are summarized in terms of minimum, maximum, mean, and standard deviation of the localization error, as well as the runtime for training each model.

Table 1: Results after 30 epochs with batch size 16, showing the effect of H. Flip and Colour Jitter on localization accuracy.

Augmentation		Localization Error				Runtime	
Horizontal Flip	Colour Jitter	min	max	mean	stdev	Time per Img	Total Time
NO	NO	0.3 px	139.6 px	22.4 px	18.3 px	1258.23 msec	14 min, 38 sec
YES	NO	0.5 px	148.8 px	19.6 px	18.4 px	2408.62 msec	28 min, 01 sec
NO	YES	0.3 px	112.3 px	21.2 px	18.2 px	2437.69 msec	28 min, 21 sec
YES	YES	0.4 px	156.0 px	18.2 px	19.0 px	2437.69 msec	28 min, 21 sec

When no augmentations were applied, the mean localization error was 22.4 pixels, with a minimum of 0.3 and a maximum of 139.6 pixels. The total training time for this configuration was relatively short, with an average of 1258.23 milliseconds per image, resulting in a total runtime of 14 minutes and 38 seconds.

Applying horizontal flipping alone slightly improved the model's performance, reducing the mean localization error to 19.6 pixels, although the maximum error increased slightly to 148.8 pixels. However, this configuration more than doubled the training time, with an average of 2408.62 milliseconds per image and a total runtime of 28 minutes and 1 second.

Introducing color jitter augmentation alone resulted in a mean localization error of 21.2 pixels and improved the minimum and maximum errors (0.3 and 112.3 pixels, respectively). The training time was comparable to the horizontal flip configuration, taking 28 minutes and 21 seconds to complete.

The final configuration, which combined both horizontal flipping and color jitter, yielded the best performance in terms of the lowest mean localization error (18.2 pixels) but came with a slight trade-off in terms of a higher maximum error of 156.0 pixels. The training time was also similar, averaging 2437.69 milliseconds per image and taking 28 minutes and 21 seconds in total.

The results of the model's performance on localizing pet noses are presented in the qualitative examples shown in Figure 11 through Figure 15. Figure 11 presents a histogram of the localization errors, illustrating the overall distribution of the model's predictions. The majority of errors fall within the 25-

to-75-pixel range, with a tail extending to higher errors above 100 pixels. This shows that while many predictions are accurate, a significant portion still contains higher errors, especially in more challenging images.

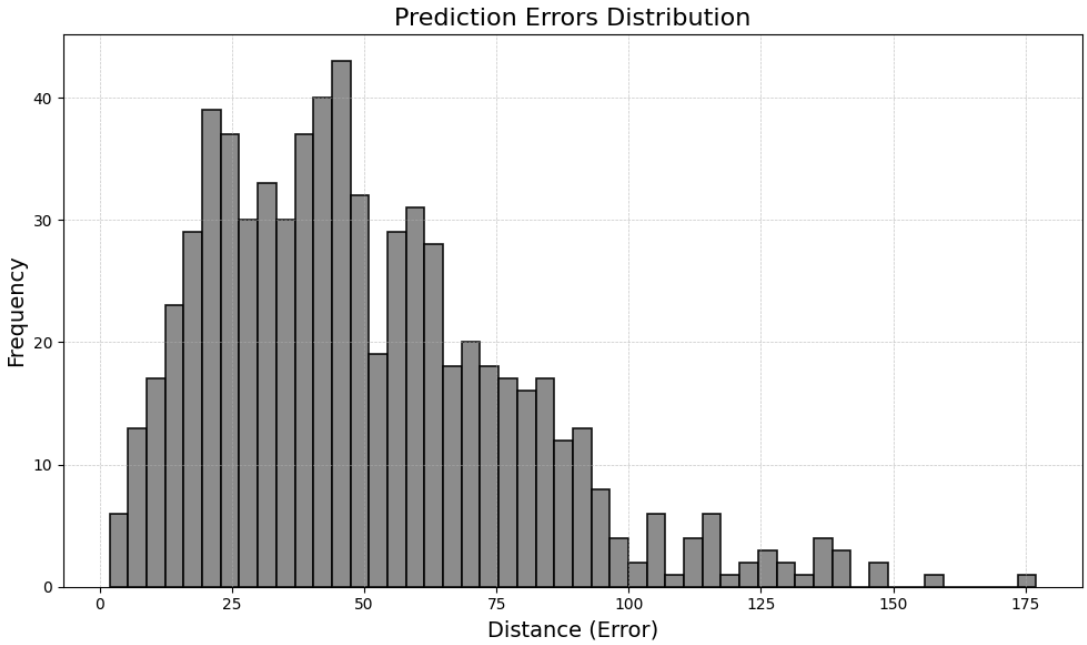

Figure 11: Localization error distribution. Most errors fall between 25-75 pixels, with outliers beyond 100 pixels.

Figure 12 through Figure 15 highlight specific examples of predictions. Figure 12 shows an almost perfect prediction with an extremely low localization error of just 1.06 pixels, indicating excellent alignment between the predicted and ground truth coordinates. Figure 13 is another successful case, with a low error of only 4.32 pixels, showcasing the model's capability to handle clear and centered images of the pets.

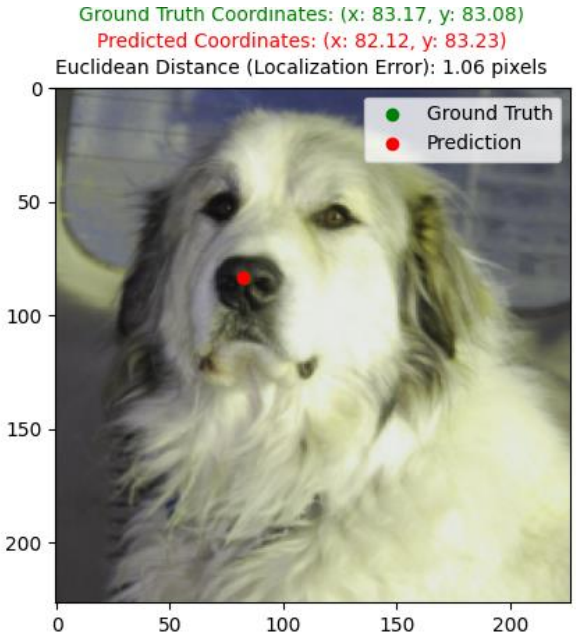

Figure 12: Example of successful localization with minimal error.

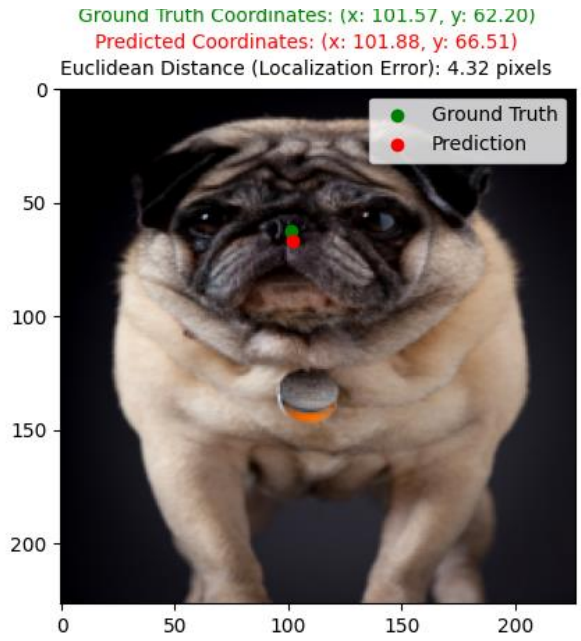

Figure 13: Another successful prediction with low localization error.

However, the performance declines in more complex cases. Figure 14 shows a prediction with a substantial localization error of 153.45 pixels, where the model struggled with the background complexity and movement of the subject. Similarly, Figure 15 demonstrates a significant error of 145.12 pixels, where the model's prediction was considerably off from the ground truth, likely due to the challenging image conditions, such as lighting or object occlusion.

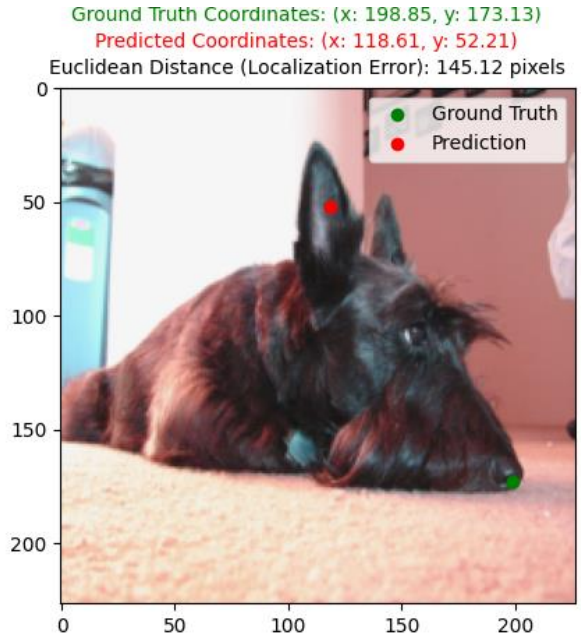

Figure 14: Example of high localization error.

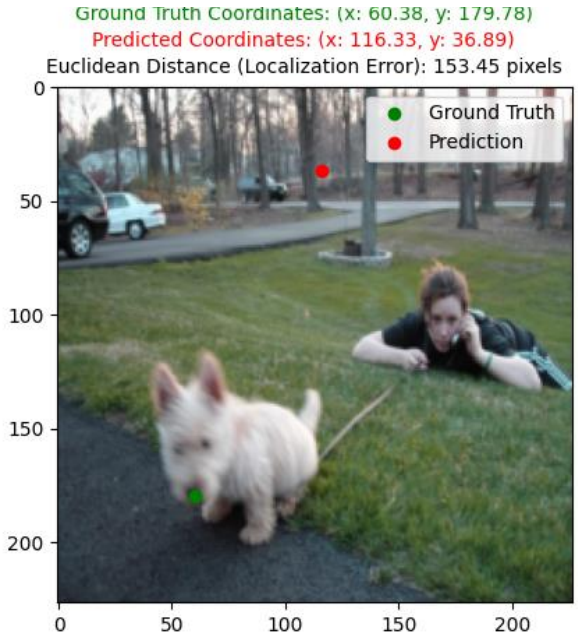

Figure 15: Another example of high localization error.

5.2. Model Performance

The model performed decently well overall, though there is room for improvement. While I am satisfied with its performance, I did expect slightly better accuracy, especially in the more challenging cases where localization errors exceeded 100 pixels, as seen in some of the higher error examples. The model handled simpler cases quite well, achieving errors as low as 1.06 pixels in some instances.

One of the key benefits came from using data augmentation, particularly the combination of horizontal flips and color jitter, which helped generalize the model to different image variations. This resulted in improved performance, though the gains were incremental. Augmentations such as color jitter seemed especially helpful in cases where lighting conditions varied, as it forced the model to adapt to these changes.

A significant challenge I encountered was during early training phases, where I noticed the model took about 4 minutes per epoch. After inspecting the model using torch summary, I realized that it had around 200 million parameters, making it excessively large and inefficient. By optimizing the architecture and reducing the number of parameters, the model became much faster and more efficient, bringing the training time down substantially without compromising performance. Overcoming this challenge helped ensure that training could proceed smoothly, especially when experimenting with multiple models and augmentations.

Part 6: Appendix

6.1. Reality Check Images

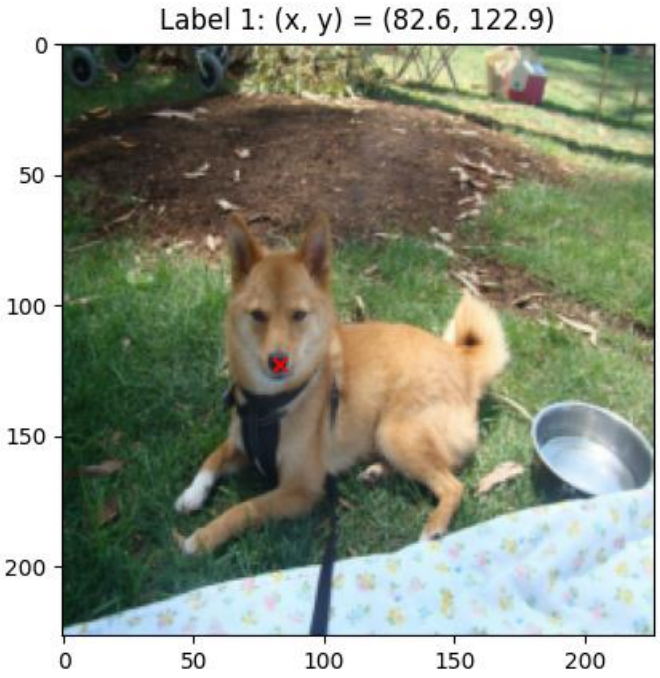

Figure 16: The first image in the train_noses.txt file with the resize transformation applied.

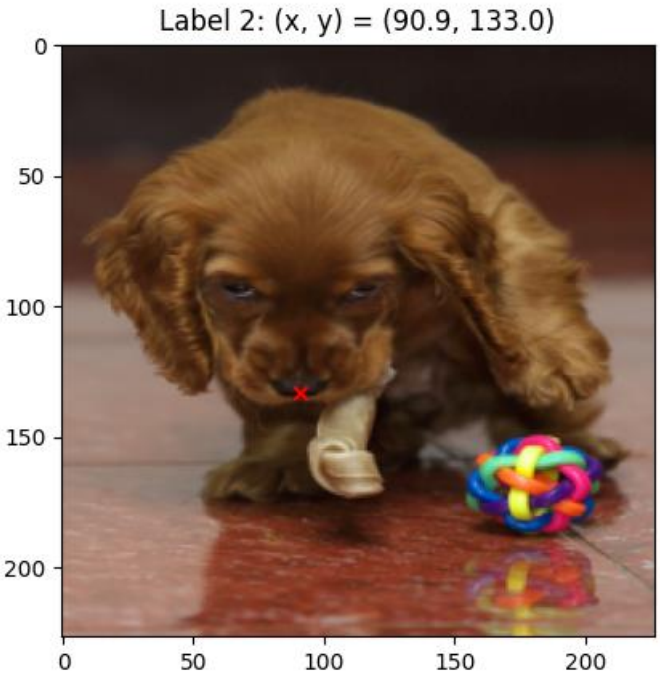

Figure 17: The second image in the train_noses.txt file with the resize transformation applied.

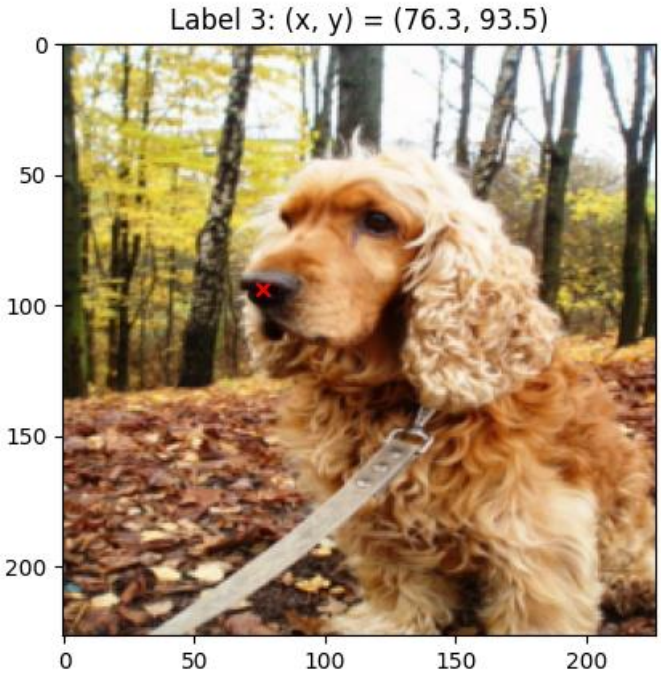

Figure 18: The third image in the train_noses.txt file with the resize transformation applied.

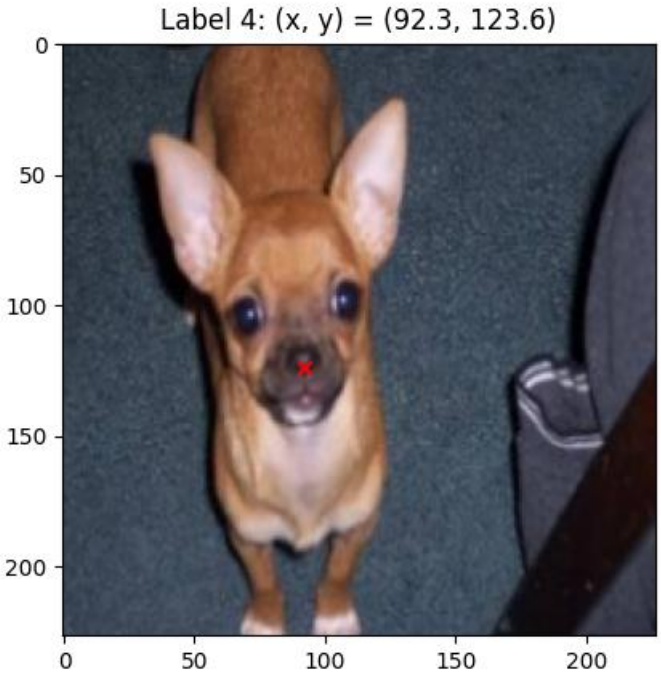

Figure 19: The fourth image in the train_noses.txt file with the resize transformation applied.

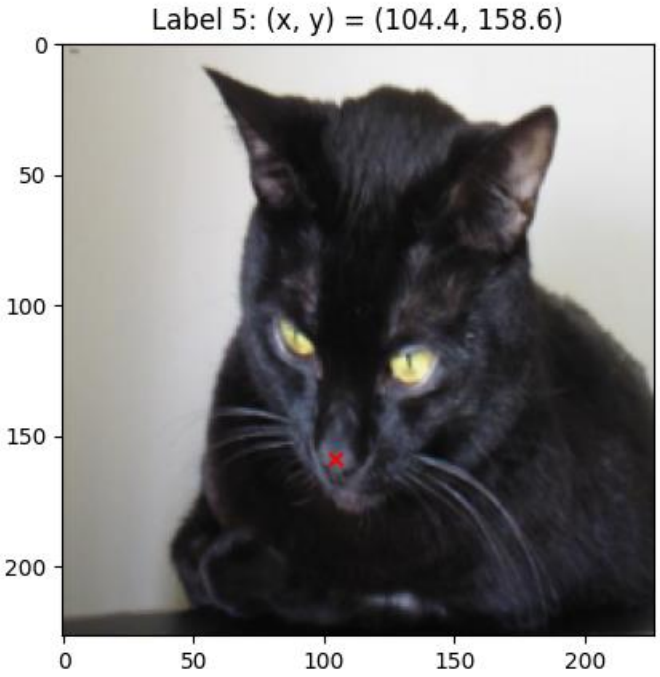

Figure 20: The fifth image in the train_noses.txt file with the resize transformation applied.

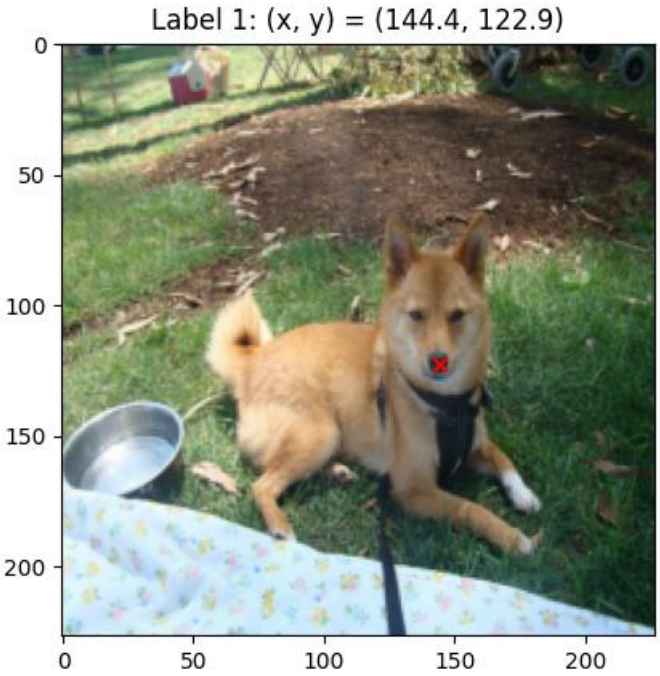

Figure 21: The first image in the train_noses.txt file with the resize and horizontal flip transformation applied.

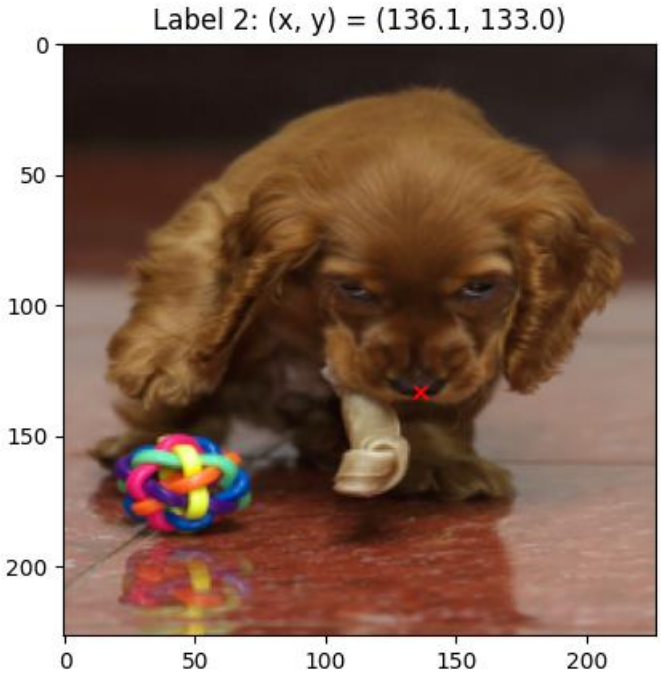

Figure 22: The second image in the train_noses.txt file with the resize and horizontal flip transformation applied.

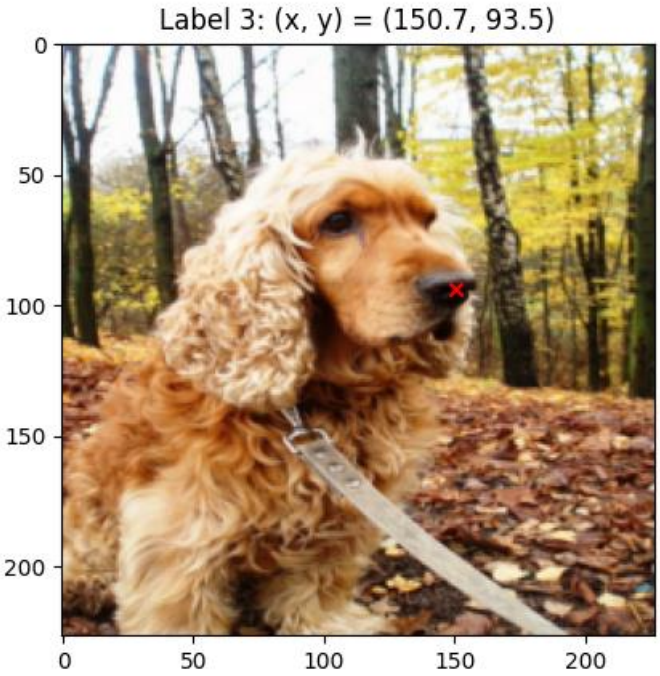

Figure 23: The third image in the train_noses.txt file with the resize and horizontal flip transformation applied.

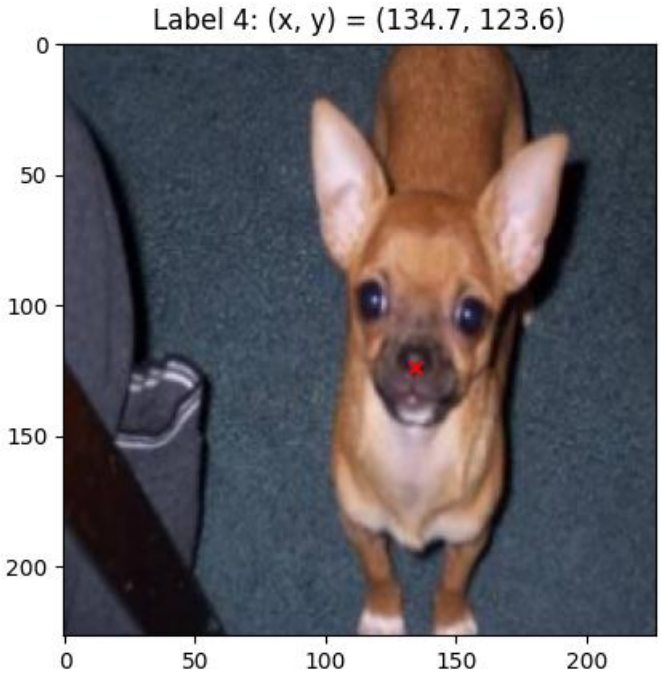

Figure 24: The fourth image in the train_noses.txt file with the resize and horizontal flip transformation applied.

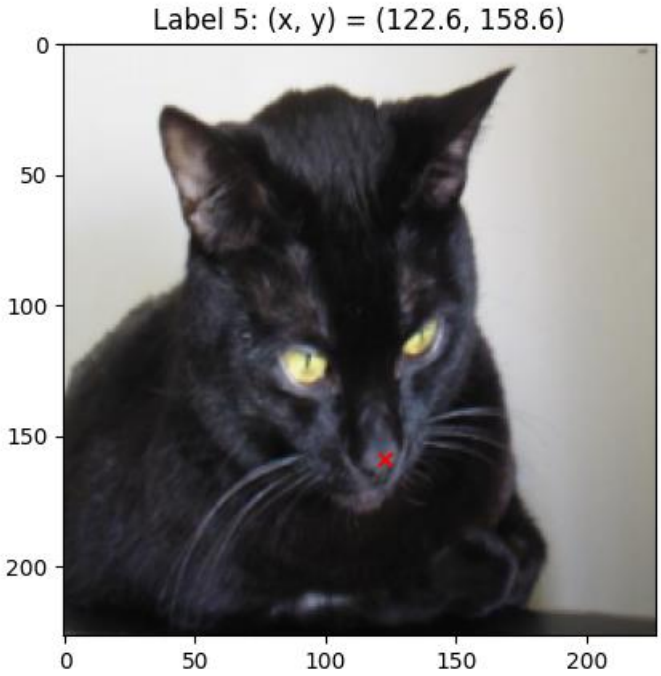

Figure 25: The fifth image in the train_noses.txt file with the resize and horizontal flip transformation applied.

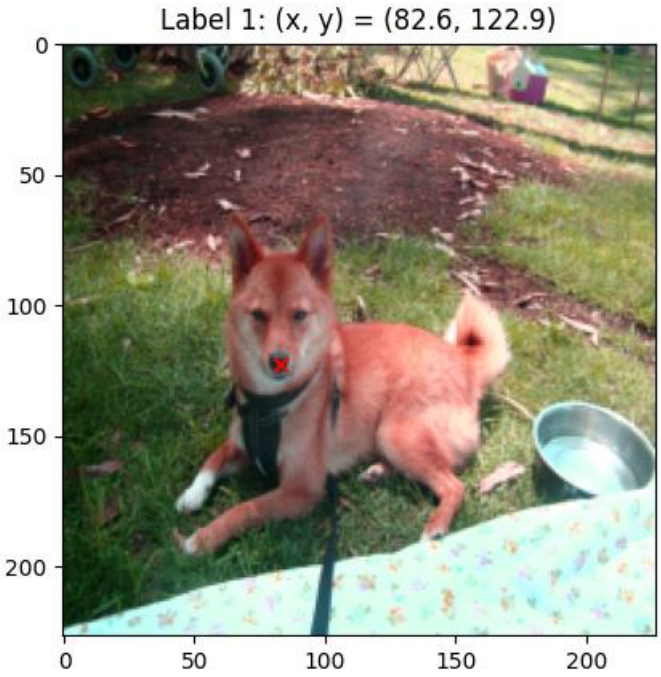

Figure 26: The first image in the train_noses.txt file with the resize and colour jitter transformation applied.

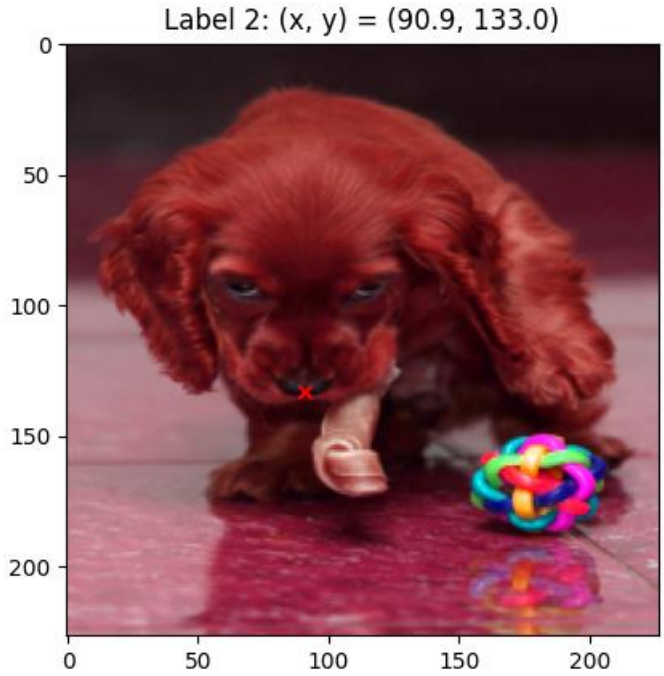

Figure 27: The second image in the train_noses.txt file with the resize and colour jitter transformation applied.

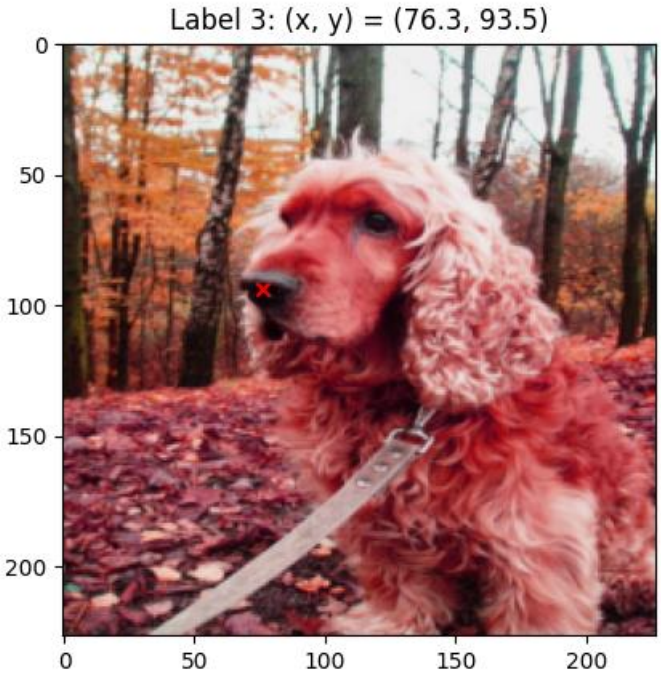

Figure 28: The third image in the train_noses.txt file with the resize and colour jitter transformation applied.

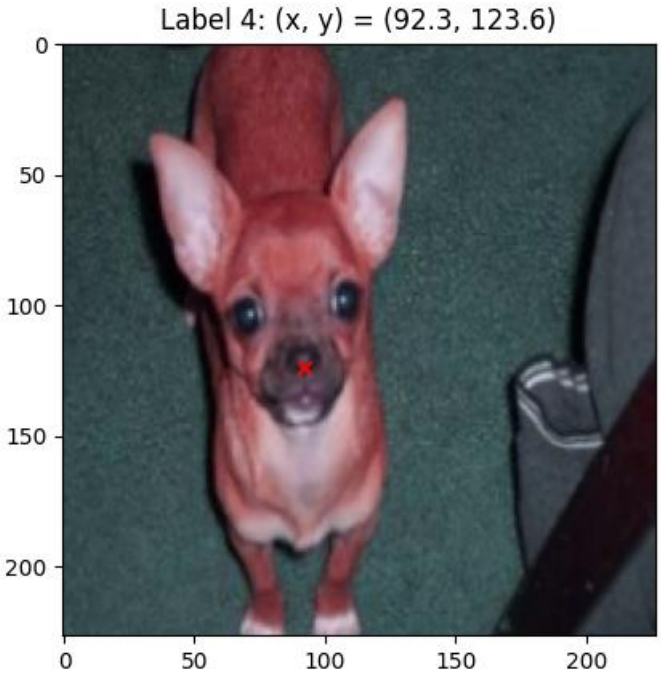

Figure 29: The fourth image in the train_noses.txt file with the resize and colour jitter transformation applied.

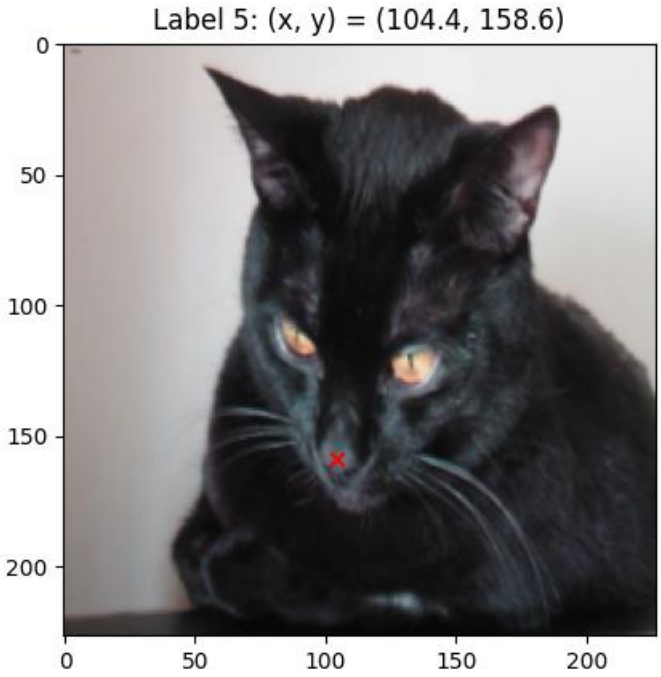

Figure 30: The fifth image in the train_noses.txt file with the resize and colour jitter transformation applied.

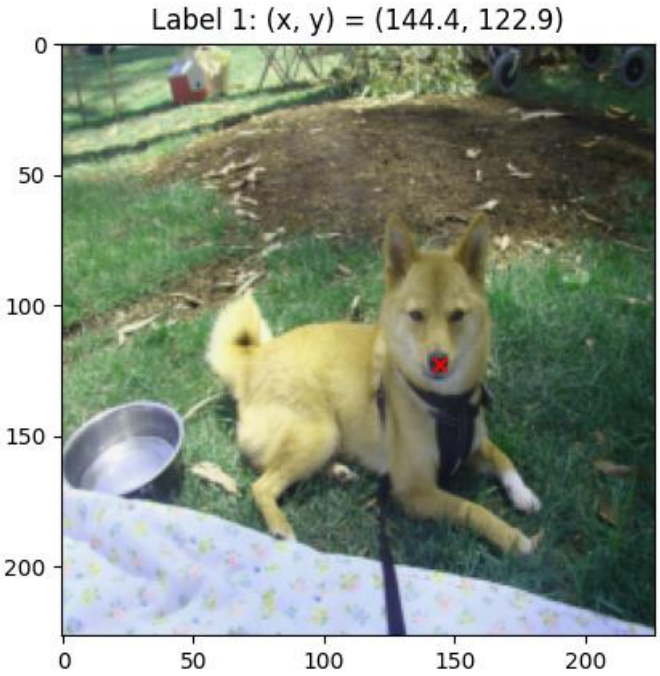

Figure 31: The first image in the train_noses.txt file with the resize, horizontal flip, and colour jitter transformation applied.

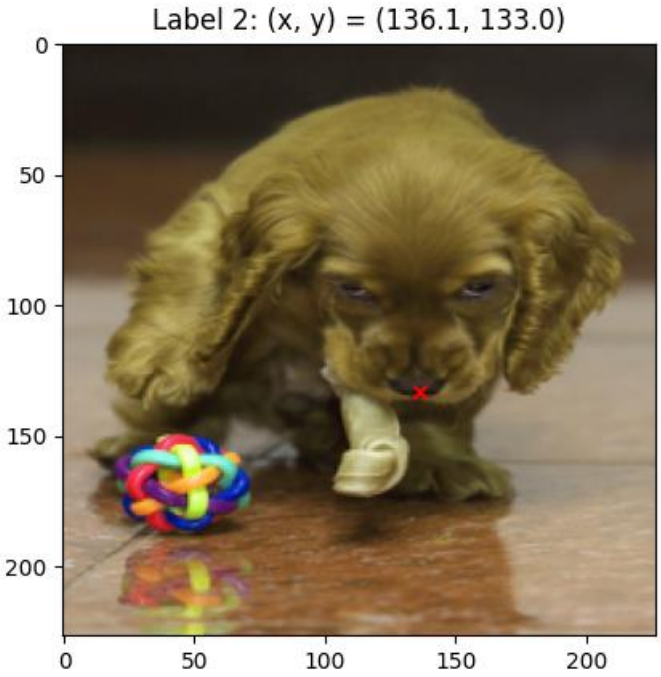

Figure 32: The second image in the train_noses.txt file with the resize, horizontal flip, and colour jitter transformation applied.

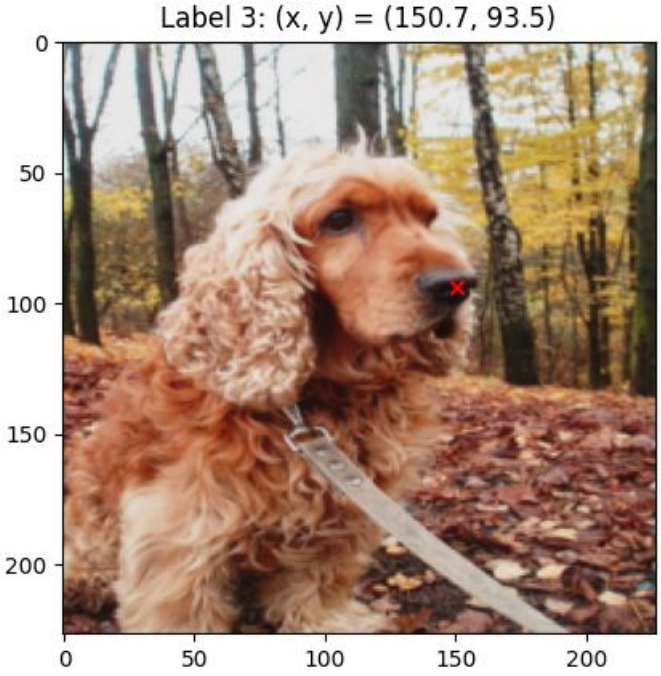

Figure 33: The third image in the train_noses.txt file with the resize, horizontal flip, and colour jitter transformation applied.

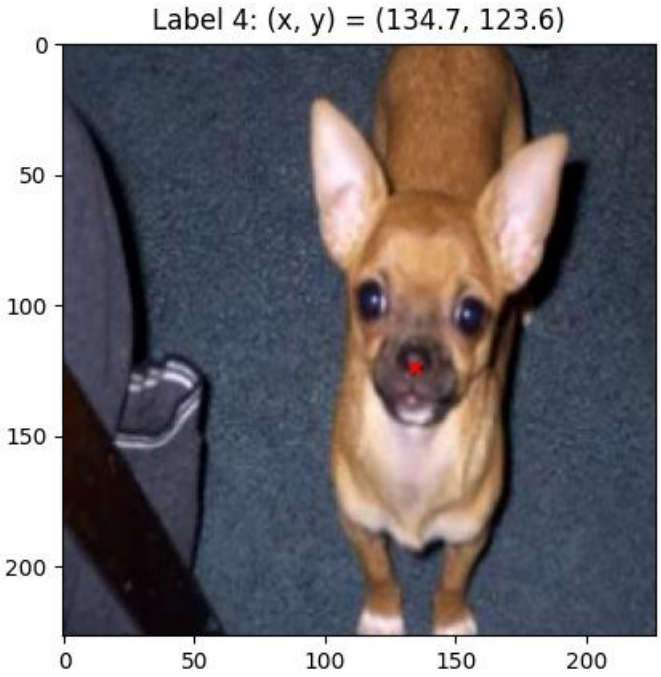

Figure 34: The fourth image in the train_noses.txt file with the resize, horizontal flip, and colour jitter transformation applied.

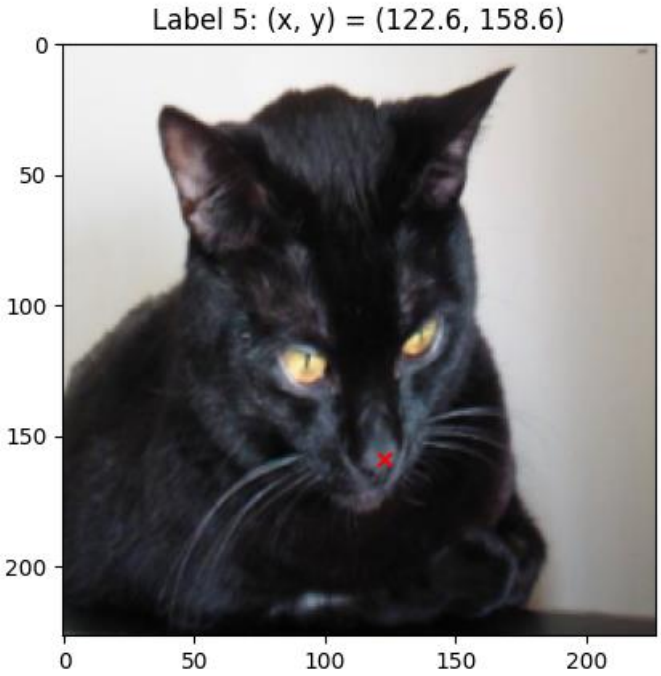

Figure 35: The fifth image in the train_noses.txt file with the resize, horizontal flip, and colour jitter transformation applied.

6.2. Optuna Output Logs

```
Trial 0:
batch_size: 32
lr: 0.0009719903909322201
weight_decay: 0.00014651200567136424
Validation loss: 467.3214

Trial 1:
batch_size: 128
lr: 0.0001426307760200956
weight_decay: 0.0001511237344782219
Validation loss: 764.4873

Trial 2:
batch_size: 64
lr: 0.00021915825425877932
weight_decay: 6.487082166980291e-05
Validation loss: 538.1883
```

Figure 36: Results for trial 0, 1, and 2 of the Optuna hyperparameter tuning.

```
Trial 3:
batch_size: 64
lr: 2.290779693786147e-05
weight_decay: 1.2285146438030237e-05
Validation loss: 1020.1188

Trial 4:
batch_size: 128
lr: 0.0003728045922444661
weight_decay: 0.000879318071603385
Validation loss: 628.1597

Trial 5:
batch_size: 32
lr: 3.269600944252984e-05
weight_decay: 0.00015280887110917632
Validation loss: 834.8051
```

Figure 37: Results for trial 3, 5, and 5 of the Optuna hyperparameter tuning.

```
Trial 6:
batch_size: 128
lr: 2.288639851807511e-05
weight_decay: 2.635555965573213e-05
Validation loss: 1126.4418

Trial 7:
batch_size: 32
lr: 5.581835091803989e-05
weight_decay: 9.505203314489739e-05
Validation loss: 761.4074
```

Figure 38: Results for trial 6 and 7 of the Optuna hyperparameter tuning.

6.3. Loss Plots

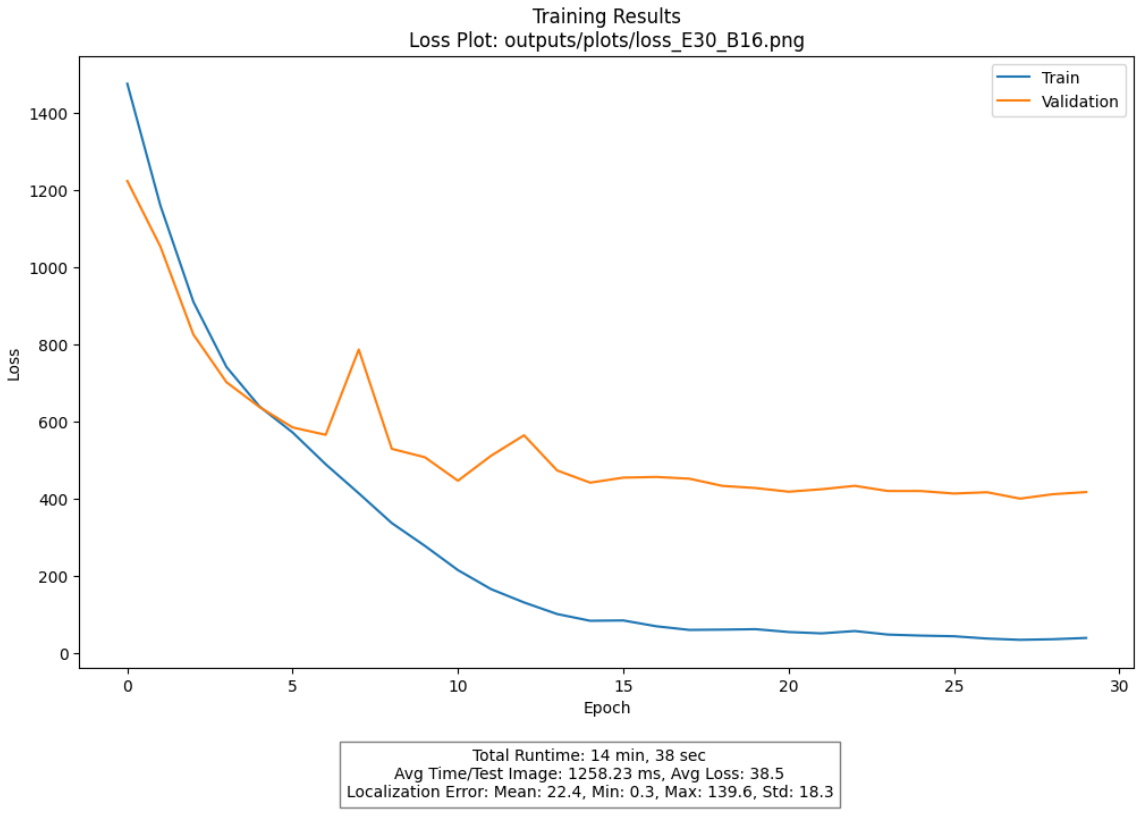

Figure 39: A plot showing the training results of a model that ran for 30 epochs with a batch size of 16, no transforms applied.

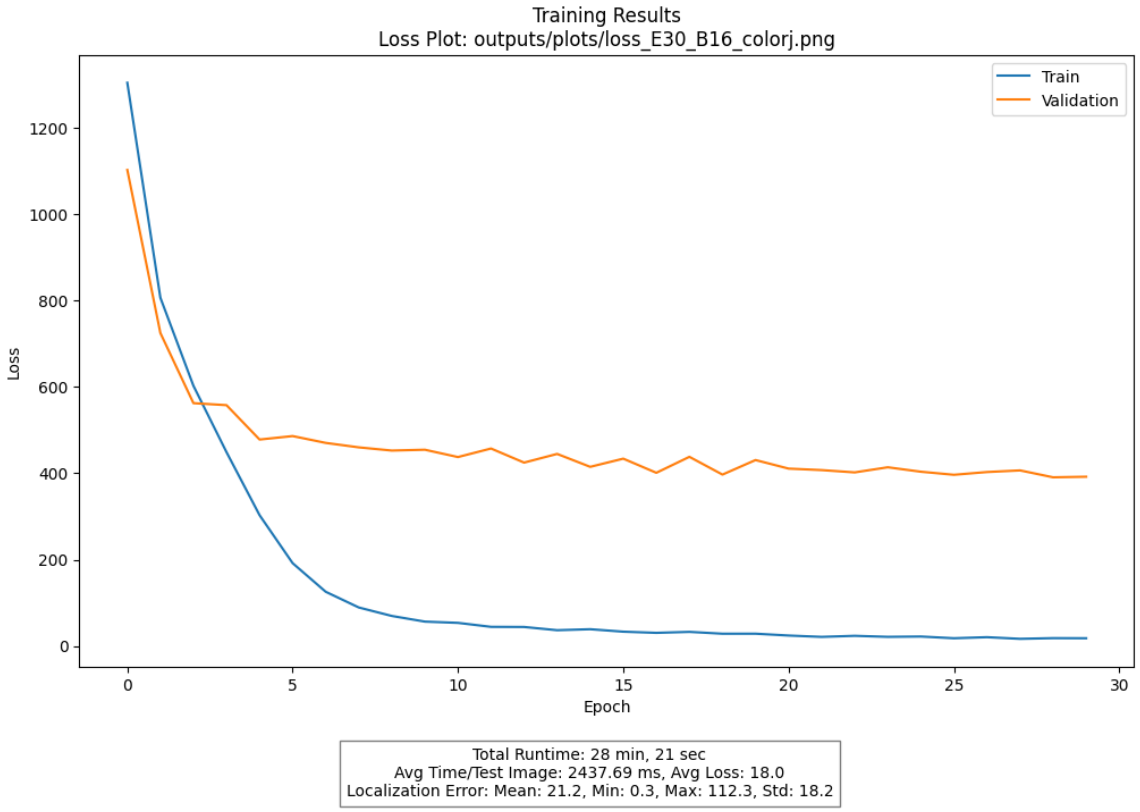

Figure 40: A plot showing the training results of a model that ran for 30 epochs with a batch size of 16, color jitter applied.

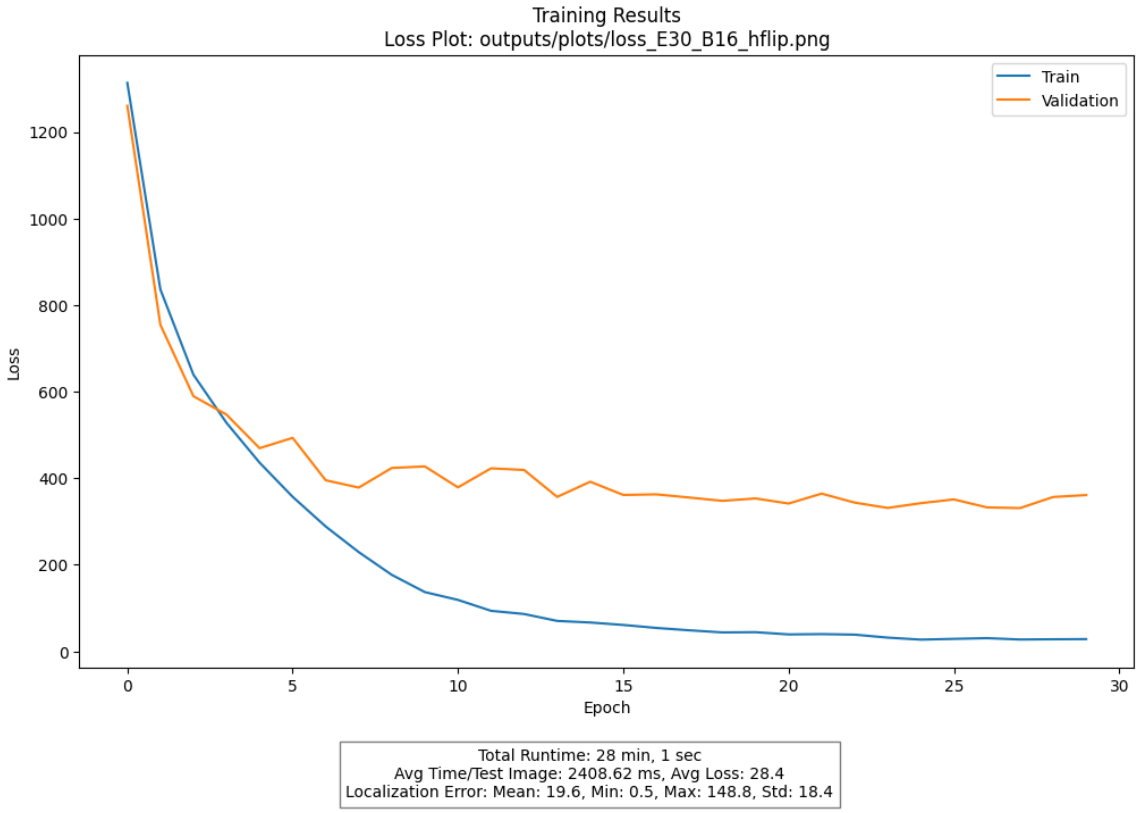

Figure 41: A plot showing the training results of a model that ran for 30 epochs with a batch size of 16, horizontal flip applied.

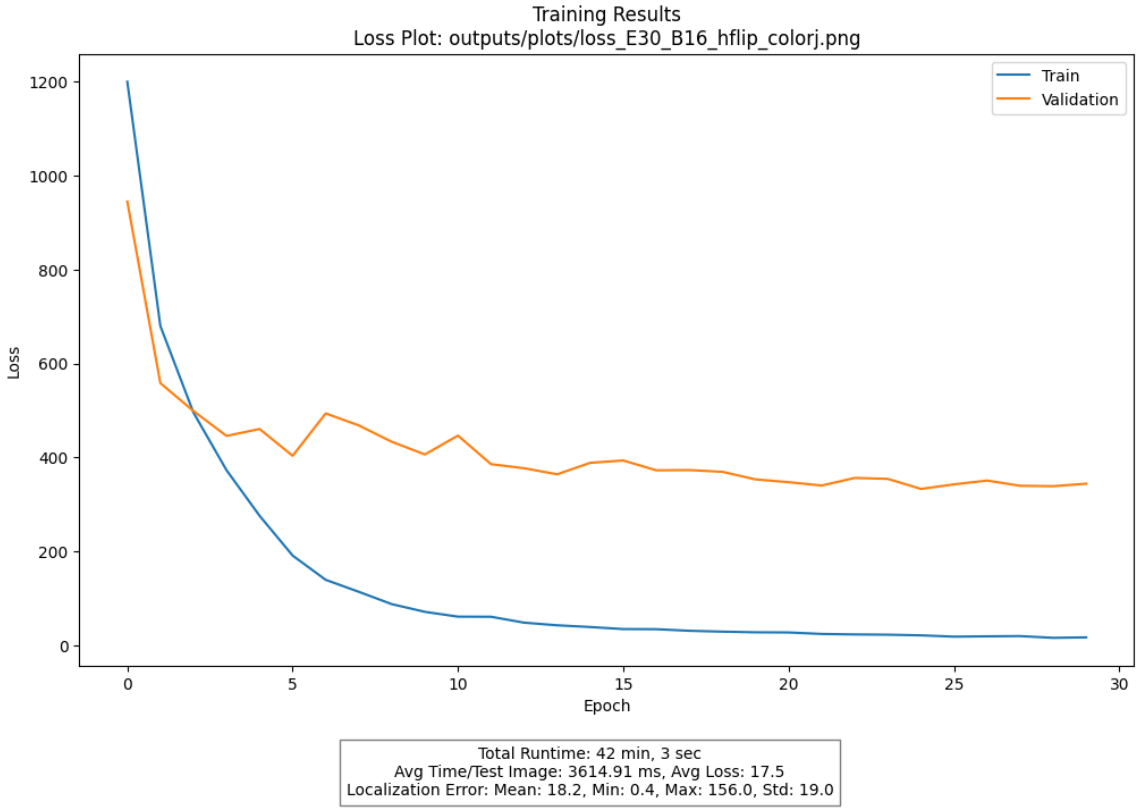

Figure 42: A plot showing the training results of a model that ran for 30 epochs with a batch size of 16, both transforms applied.

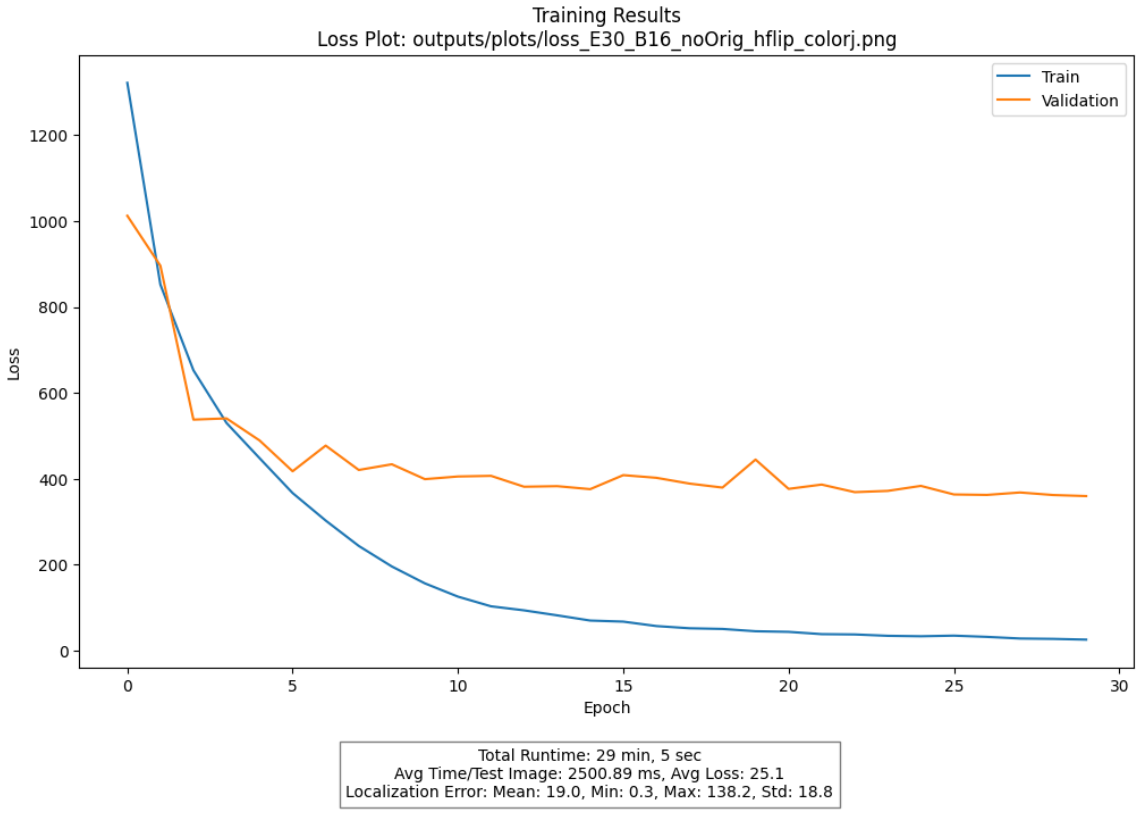

Figure 43: A plot showing the results of a model that ran for 30 epochs with a batch size of 16, only augmented data applied.

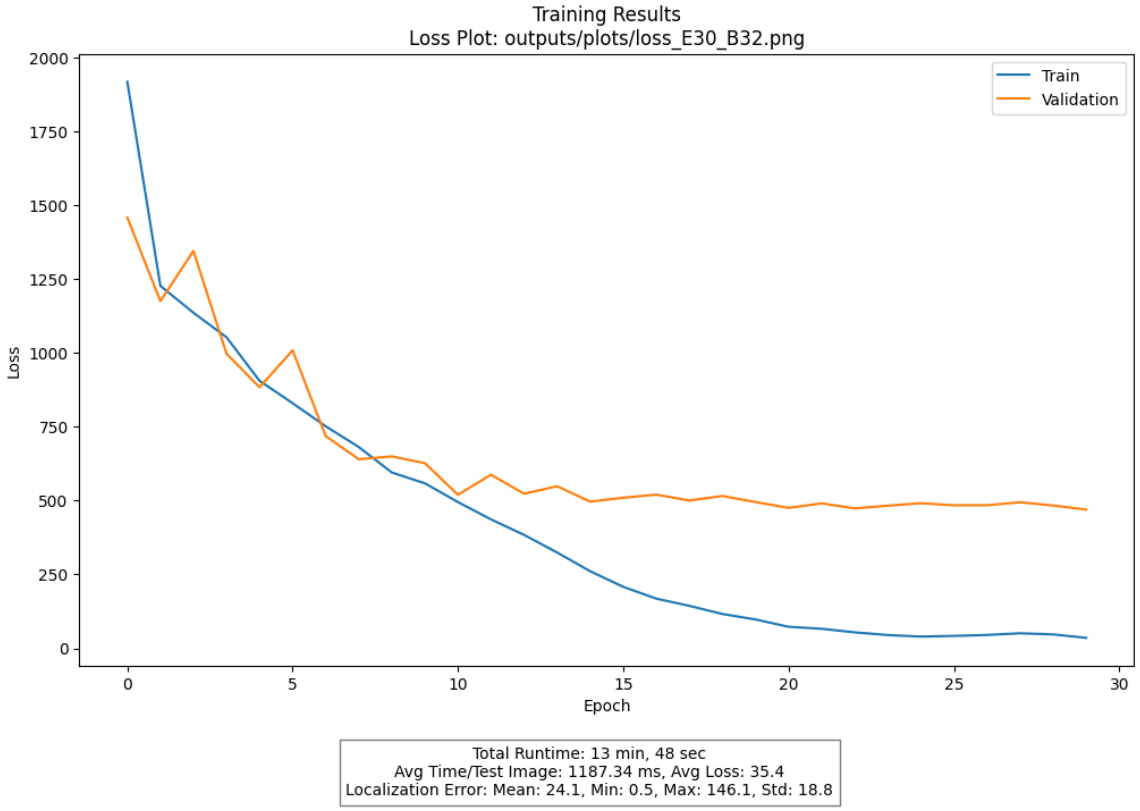

Figure 44: A plot showing the training results of a model that ran for 30 epochs with a batch size of 32, no transforms applied.

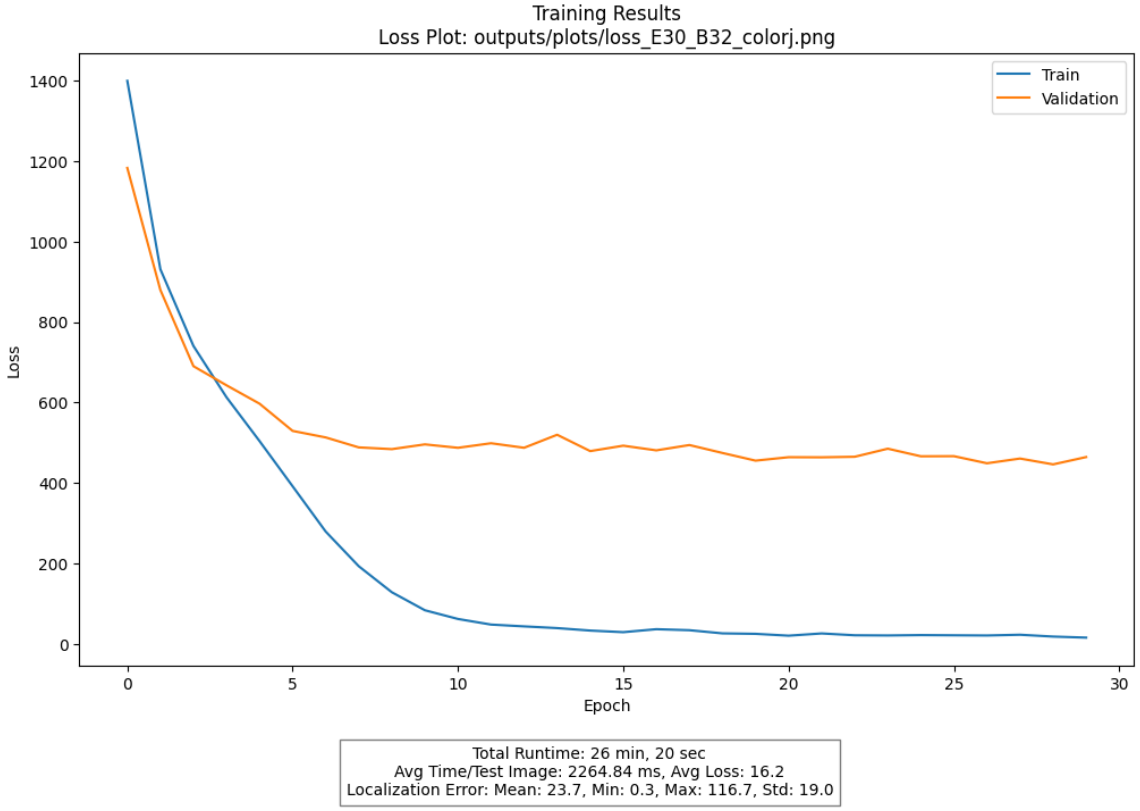

Figure 45: A plot showing the training results of a model that ran for 30 epochs with a batch size of 32, color jitter applied.

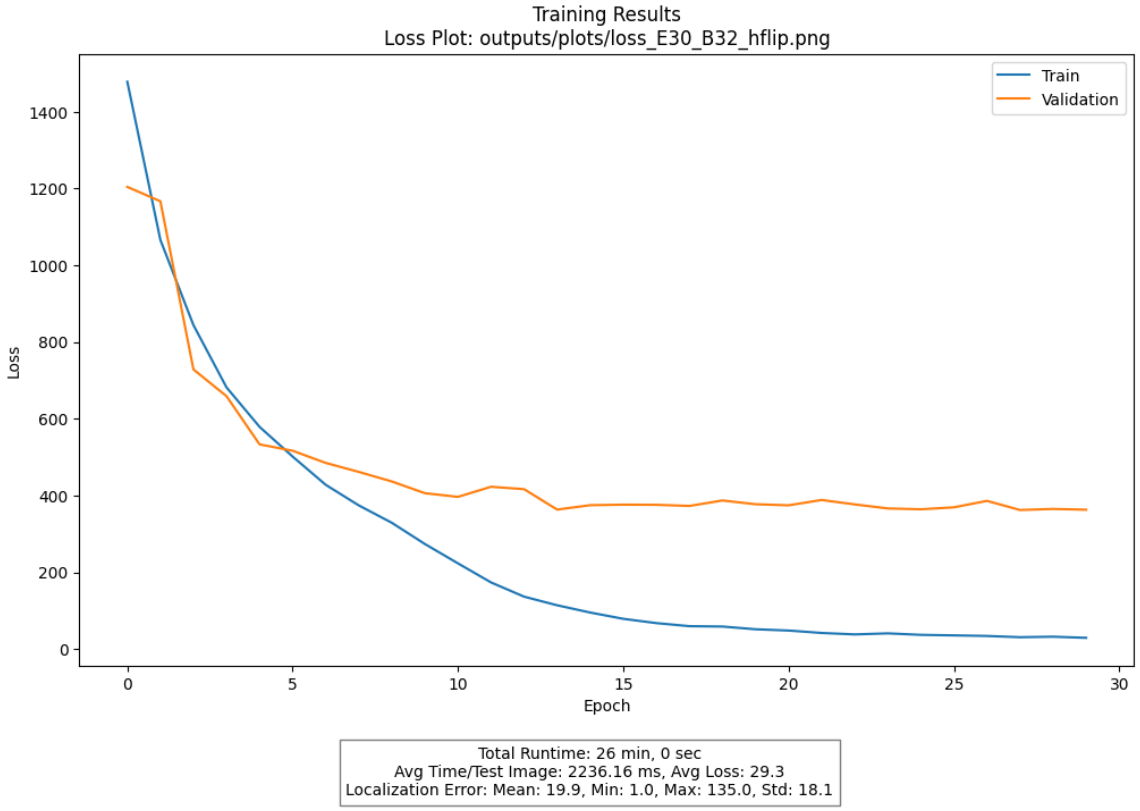

Figure 46: A plot showing the training results of a model that ran for 30 epochs with a batch size of 32, horizontal flip applied.

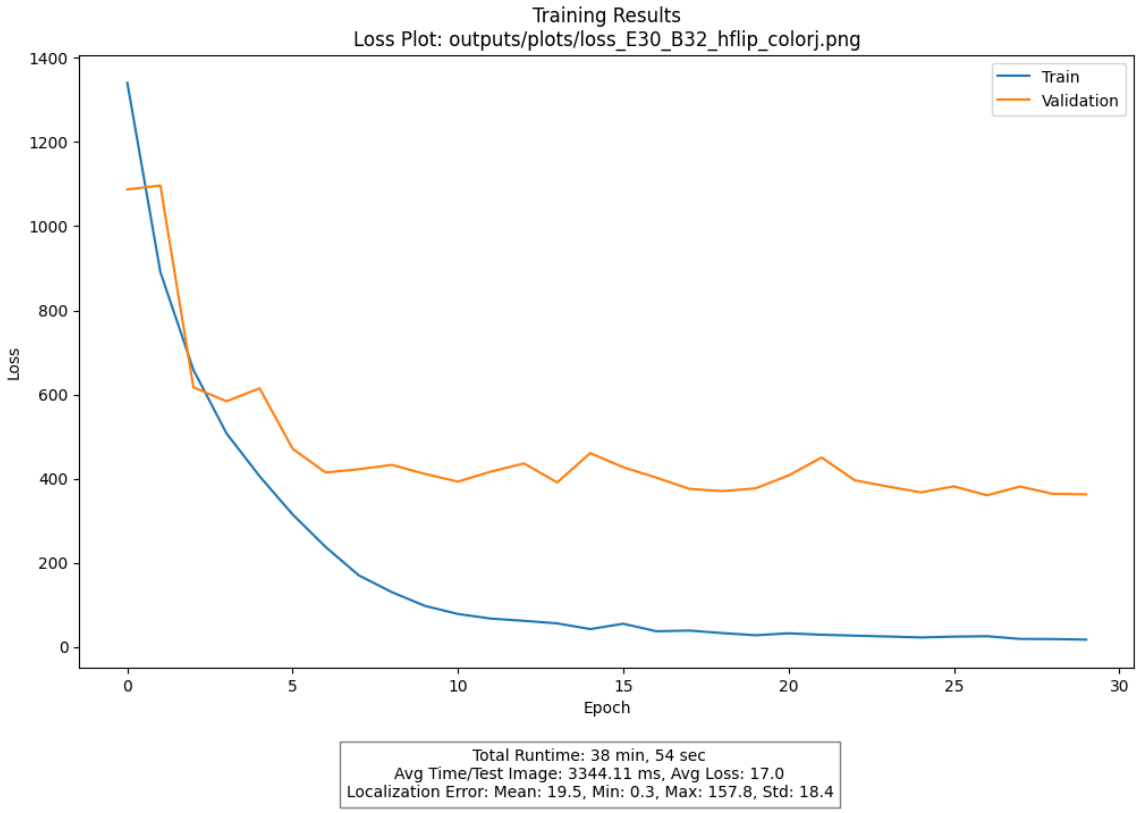

Figure 47: A plot showing the training results of a model that ran for 30 epochs with a batch size of 32, both transforms applied.

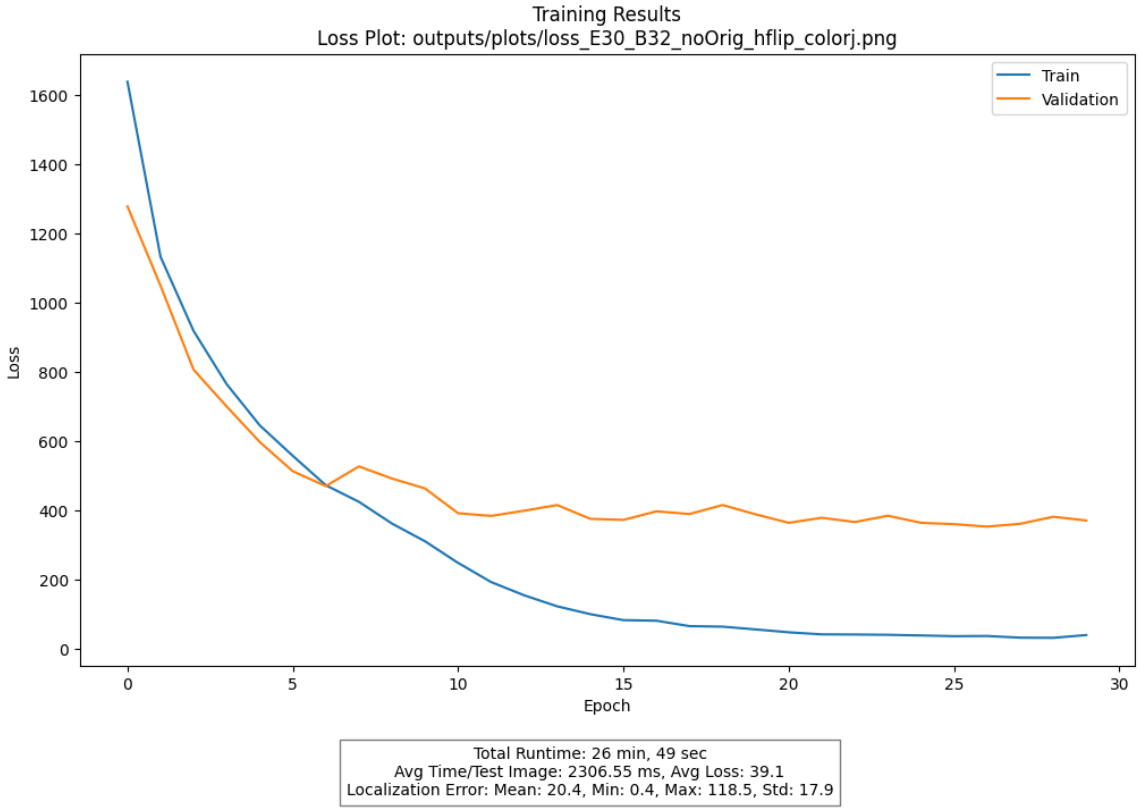

Figure 48: A plot showing the results of a model that ran for 30 epochs with a batch size of 32, only augmented data applied.

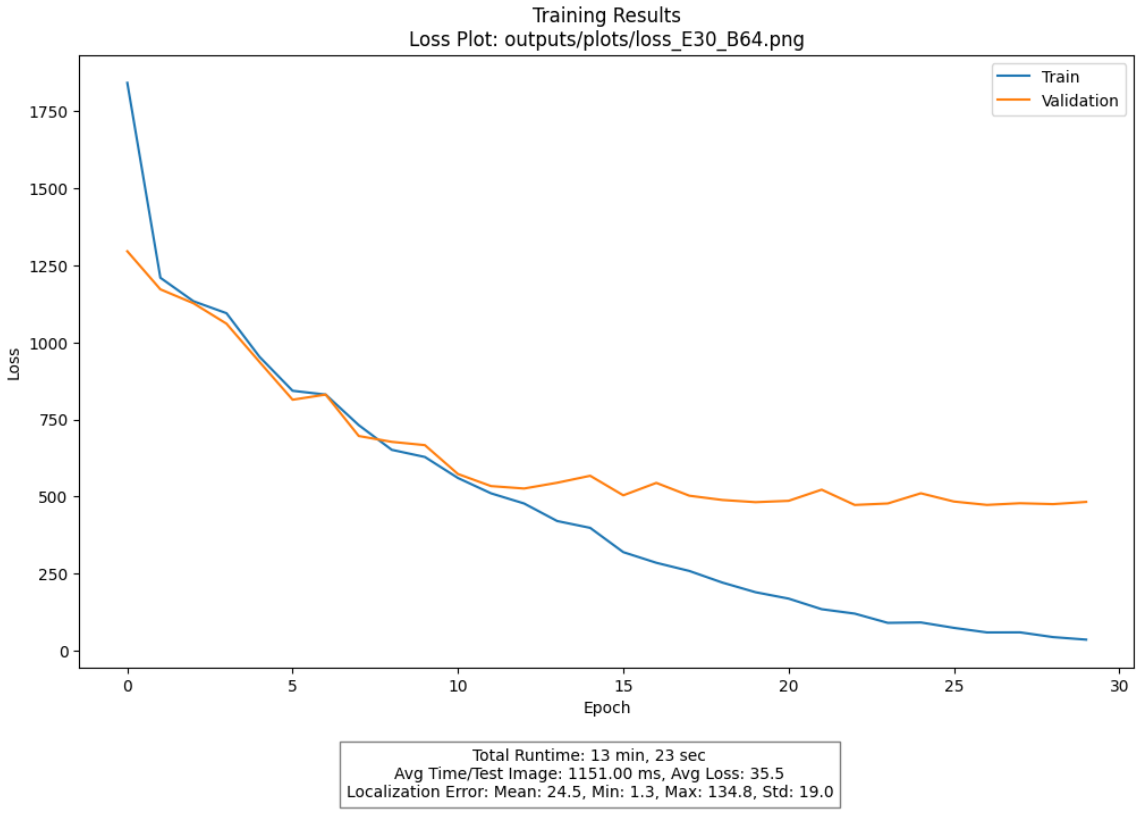

Figure 49: A plot showing the training results of a model that ran for 30 epochs with a batch size of 64, no transforms applied.

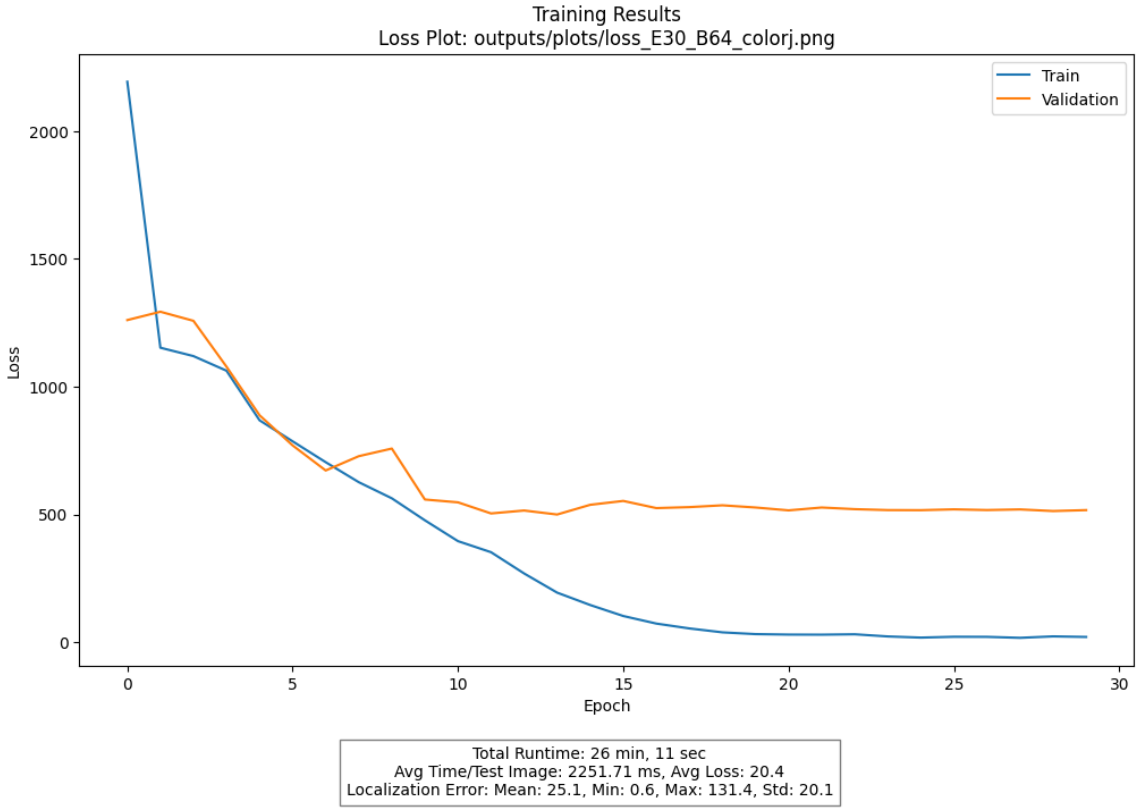

Figure 50: A plot showing the training results of a model that ran for 30 epochs with a batch size of 64, color jitter applied.

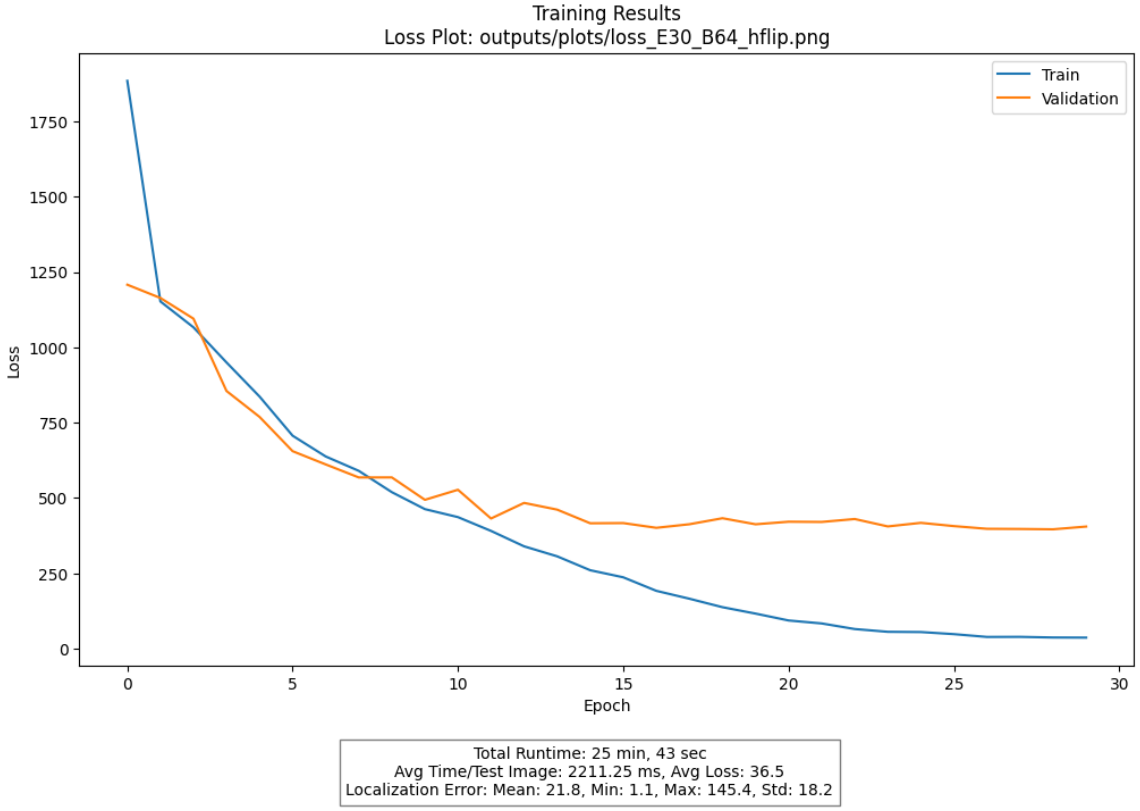

Figure 51: A plot showing the training results of a model that ran for 30 epochs with a batch size of 64, horizontal flip applied.

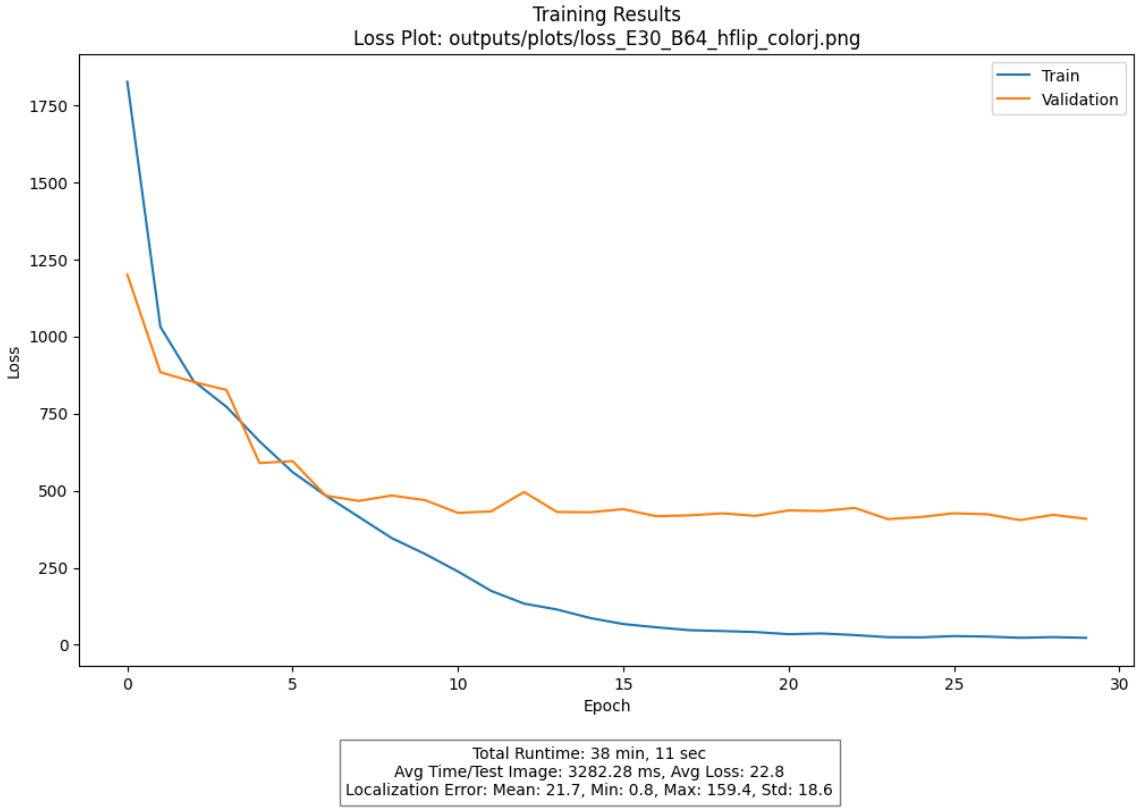

Figure 52: A plot showing the training results of a model that ran for 30 epochs with a batch size of 64, both transforms applied.

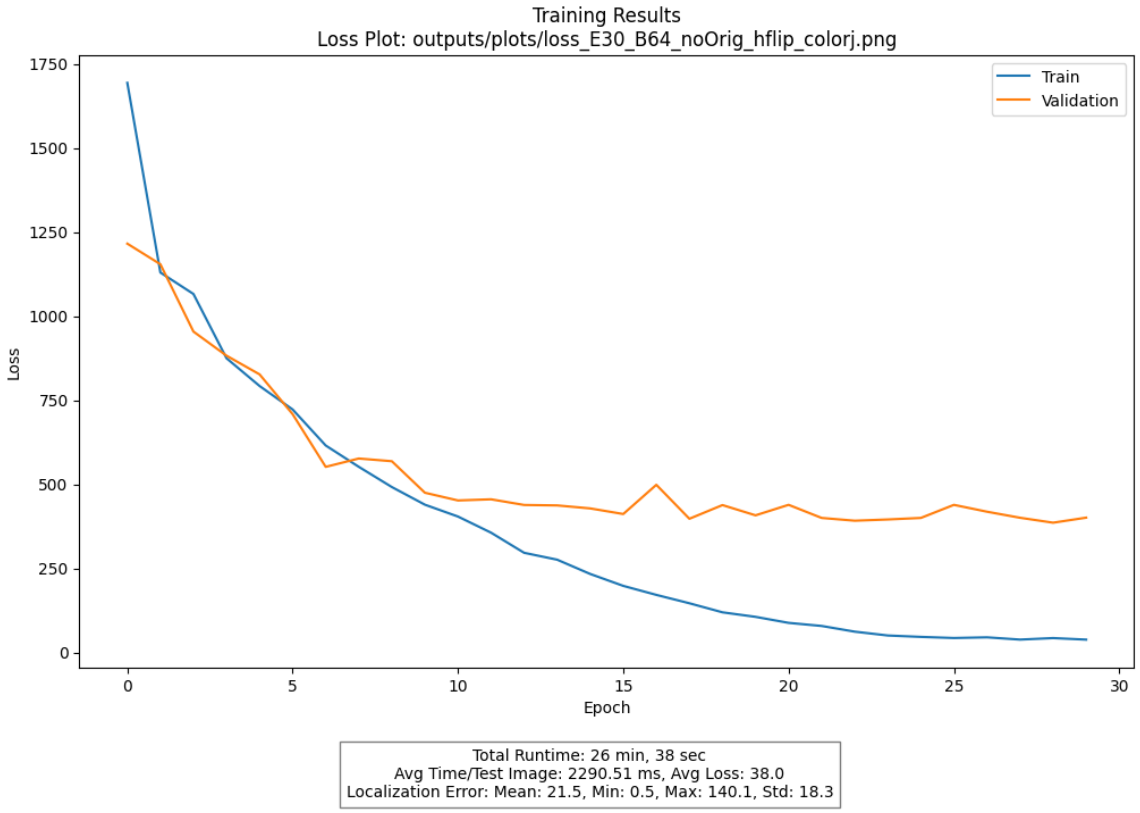

Figure 53: A plot showing the results of a model that ran for 30 epochs with a batch size of 64, only augmented data applied.

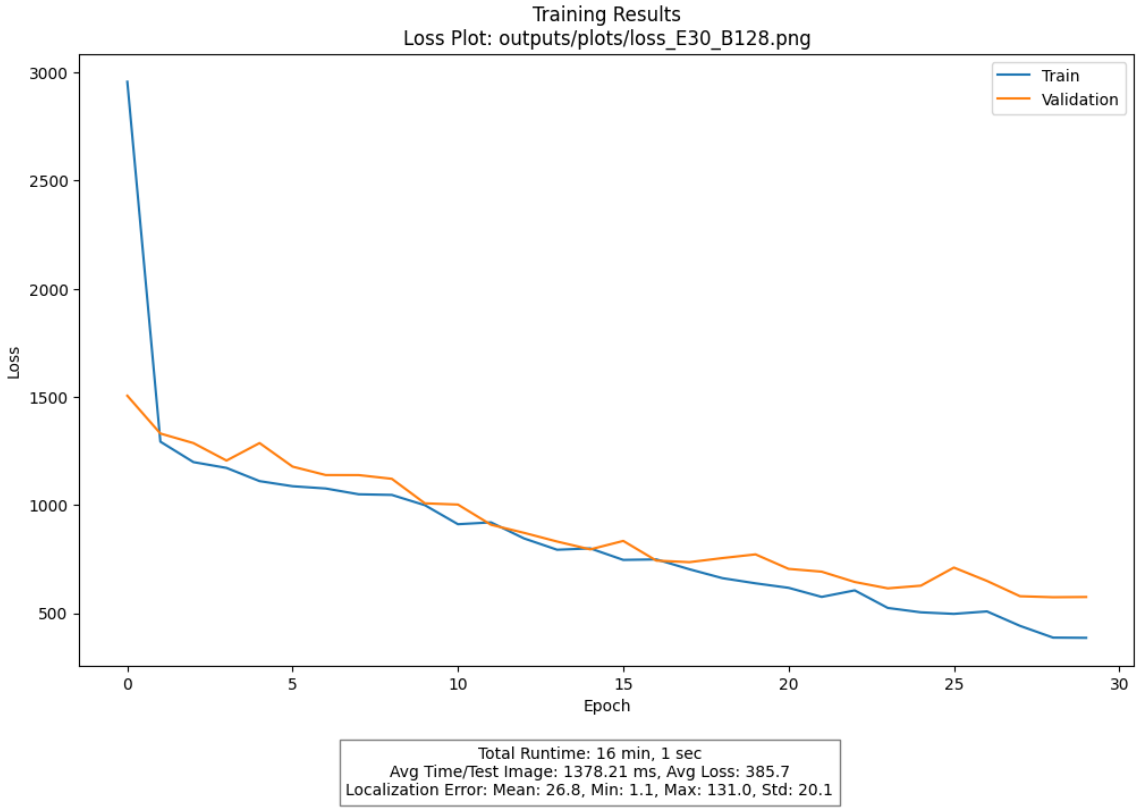

Figure 54: A plot showing the training results of a model that ran for 30 epochs with a batch size of 128, no transforms applied.

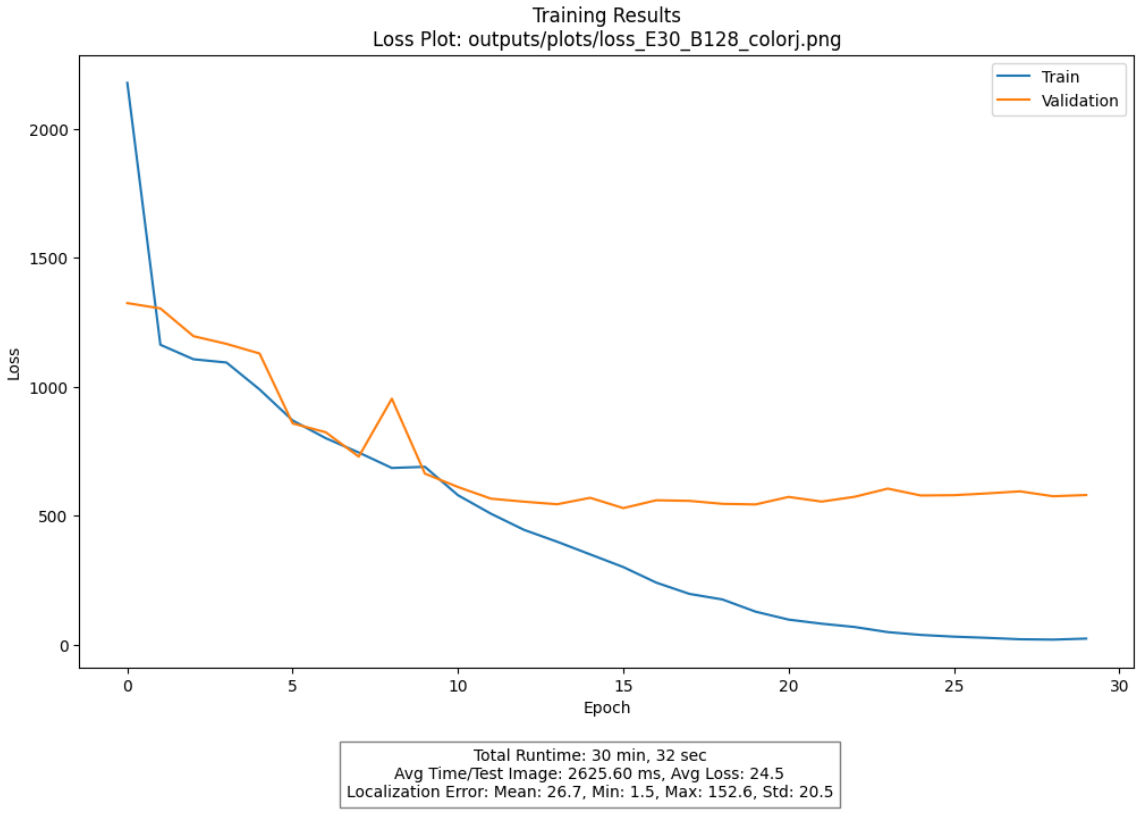

Figure 55: A plot showing the training results of a model that ran for 30 epochs with a batch size of 128, color jitter applied.

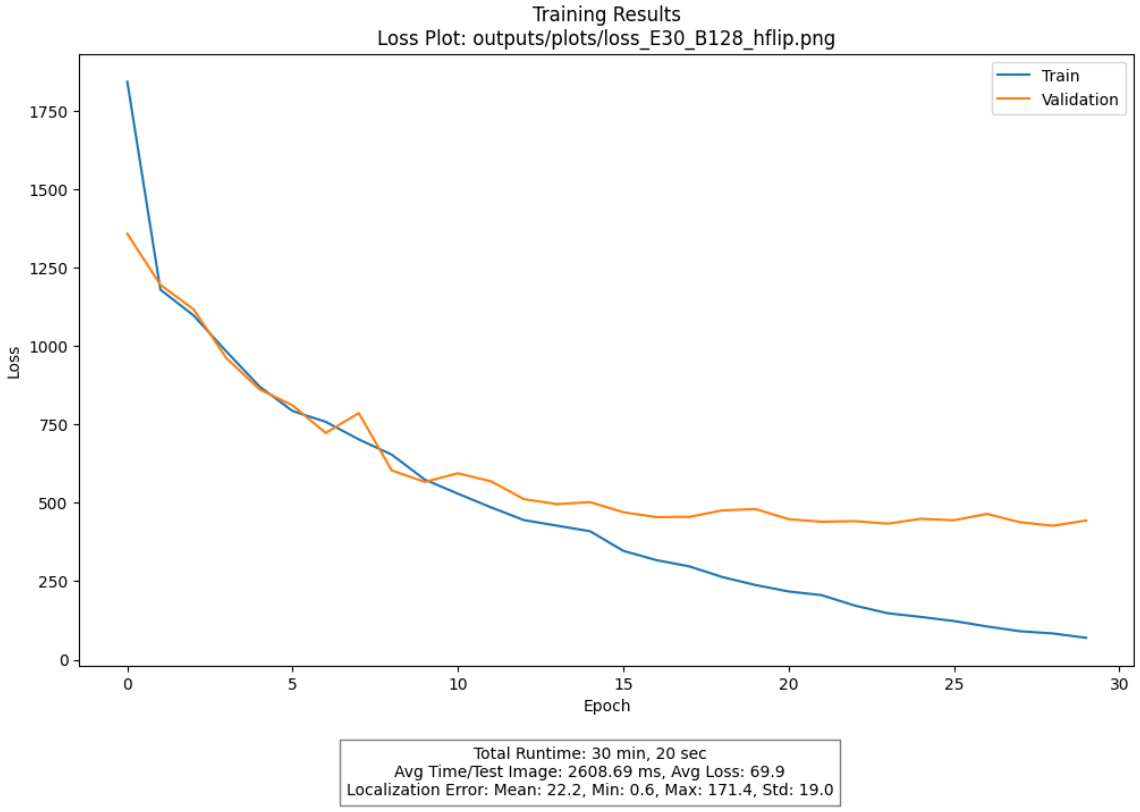

Figure 56: A plot showing the training results of a model that ran for 30 epochs with a batch size of 128, horizontal flip applied.

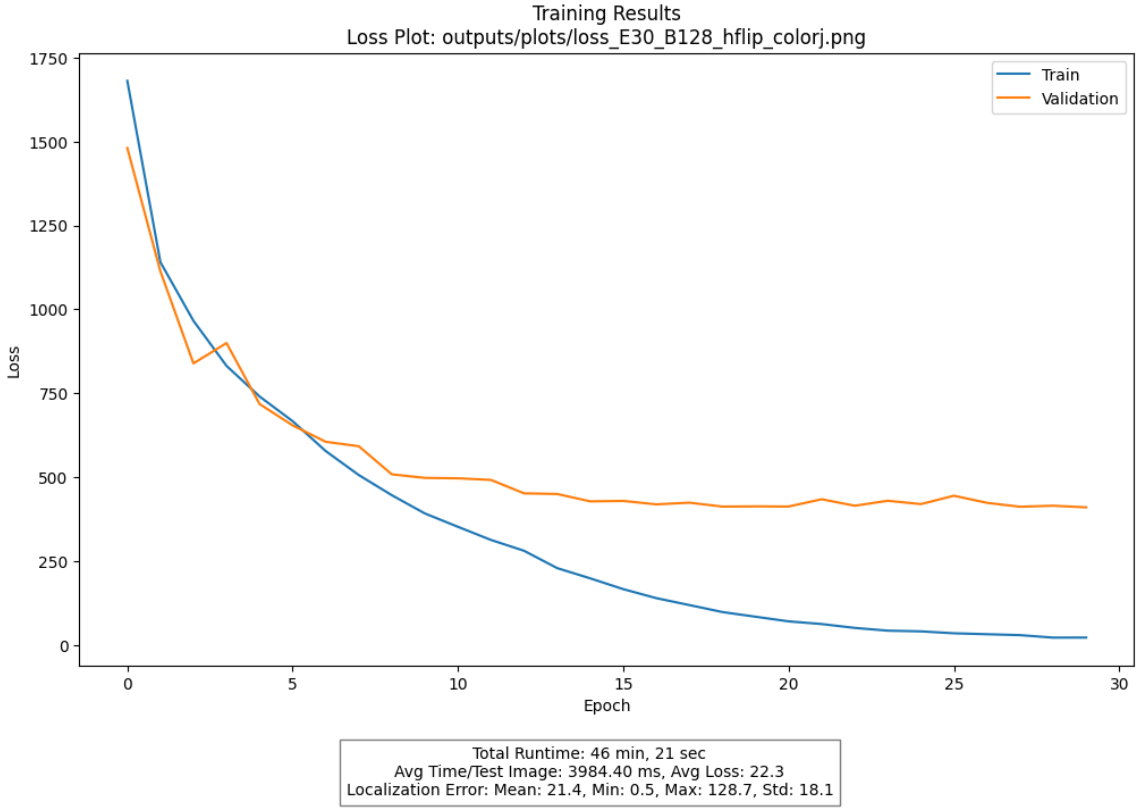

Figure 57: A plot showing the results of a model that ran for 30 epochs with a batch size of 128, both transforms applied.

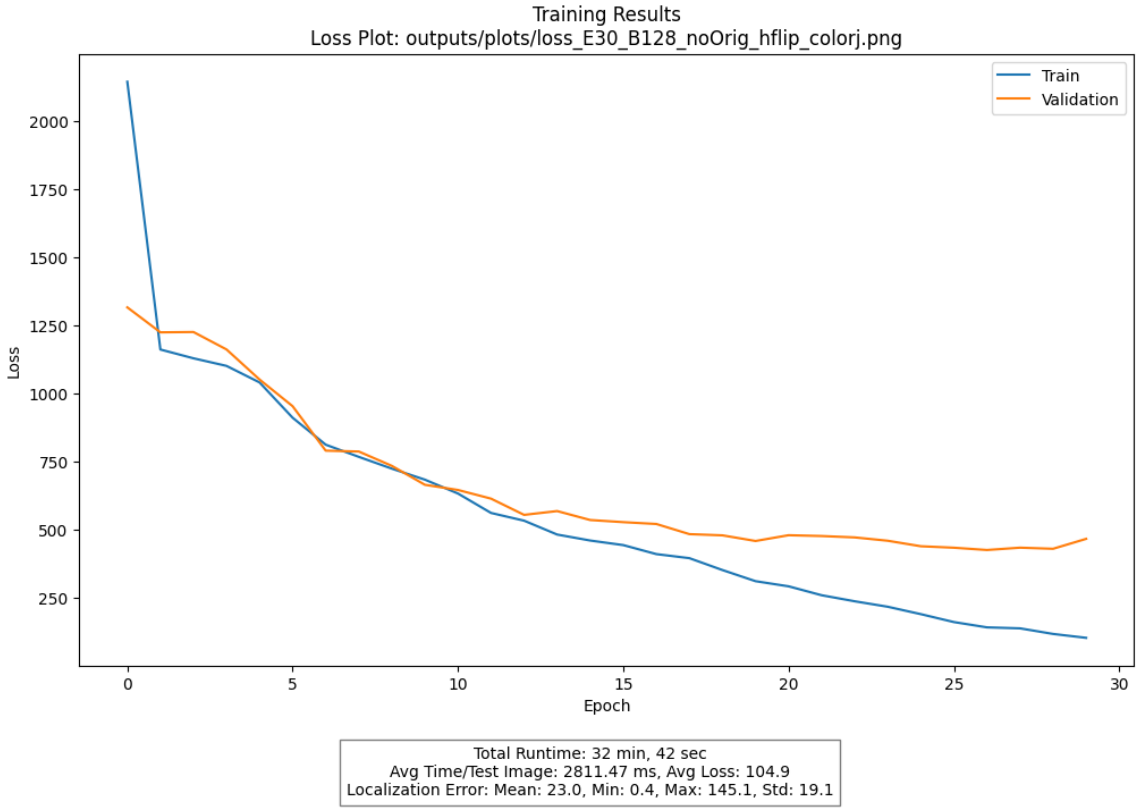

Figure 58: A plot showing the results of a model that ran for 30 epochs with a batch size of 128, only augmented data applied.