

Pet Nose Localization– ELEC 475 Lab 5 Report

Noah Serhan

Step 2 – Model

The model that I used was ResNet 18. I selected this model because it's widely applicable and versatile, and seemed like it would fit the solution to this project well. In the paper (2) by Wu et al., they made some alterations to the model. Because of the way they used ResNet 18, I was inspired to use it in a similar way for our project. In the paper, the model was used as a solution for 3D, so I took this and translated it to a 2D solution.

In terms of modifications, I added a few layers to the existing model to properly fit the scope of our project in terms of functionality.

```
class NoseNet(nn.Module):
    def __init__(self):
        super(NoseNet, self).__init__()
        self.resnet = models.resnet18(weights=ResNet18_Weights.DEFAULT)
        self.resnet = nn.Sequential(*list(self.resnet.children())[:-1])
        self.adaptive_pool = nn.AdaptiveAvgPool2d((1, 1)) # Add adaptive pooling
        self.voting_layer = nn.Linear(512, 2) # Output 2 values for x and y coordinates
```

Figure 1: Model Modifications

The first layer that is highlighted in *figure 1* creates a ResNet model and removes its last layer by constructing a new `nn.Sequential` module, containing all but the last layer of the original ResNet. The second highlighted layer, adaptive pool, initializes an adaptive average pooling layer that dynamically reduces the spatial dimensions of the input, serving as a global average pooling operation. Lastly, the voting layer defines a linear layer with input size 512 and output size 2, representing two output coordinates: x and y.

Step 3

Hyper-parameters:

- The optimization method used for this training is “Adam”, which is an adaptive learning rate optimization algorithm with efficient convergence properties that combines the advantages of both the AdaGrad and RMSProp gradient descent methods.
- The optimizer is initialized with a learning rate of $1e-4$ and has a weight decay of 0.0002.
- The batch size can be specified by the user, however, a batch size defaulted to 16 was used for both training and testing of the model. Additionally, I added another case where I changed it to 20 (and used 40 epochs).

- Like the batch size, the number of epochs can also be changed based on the user's inputs, however, there was no need to go past 20 as the loss plots would seem to plateau around or before that point.
- The scheduler used was the StepLR, which adjusts the learning rate by decaying it by a factor of gamma every set number of epochs. In this case, it was reduced every 15 epochs by a factor of 0.8.

Hardware

The hardware used for training was a RTX 3090, 24 GB of VRAM, which made use of NVIDIA's CUDA capabilities to allow for faster training times. If CUDA is not available however, a CPU can be used to train and test the models, with the consequence of it taking more time to do so.

Time Taken

The training of our model took just over 20 minutes, equating to around 60 seconds for each individual epoch.

Loss plot

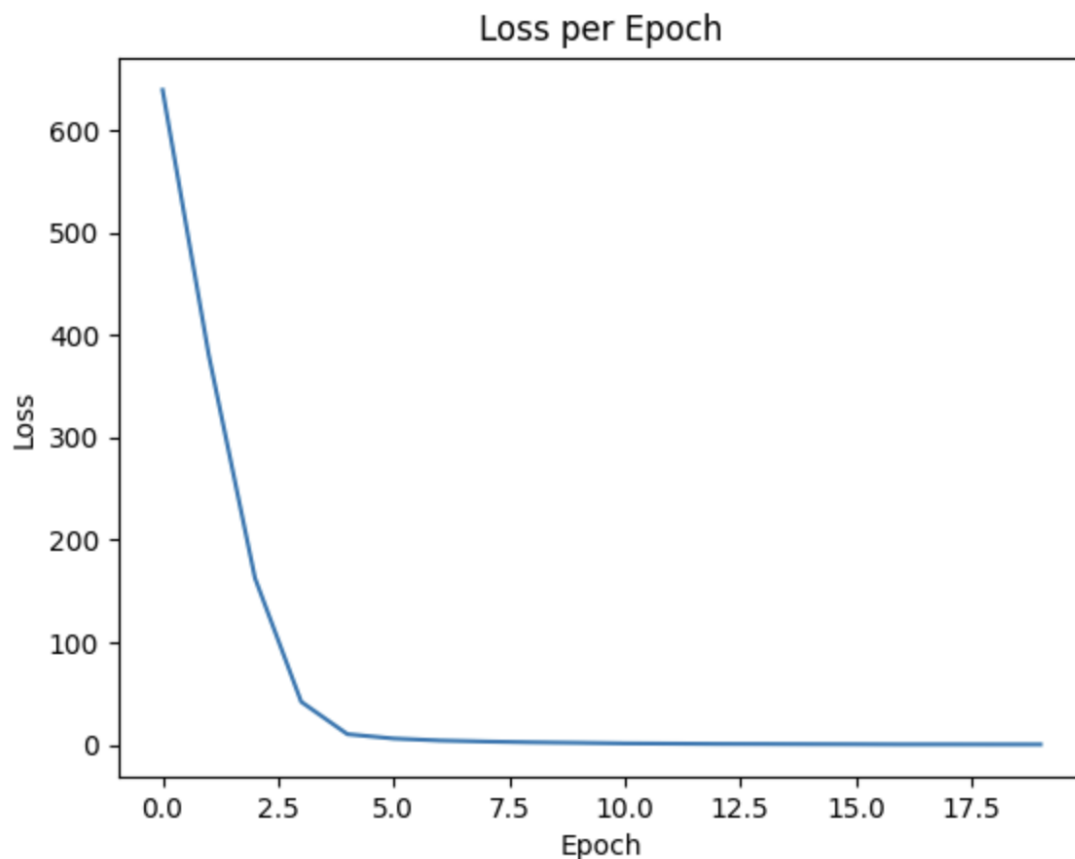
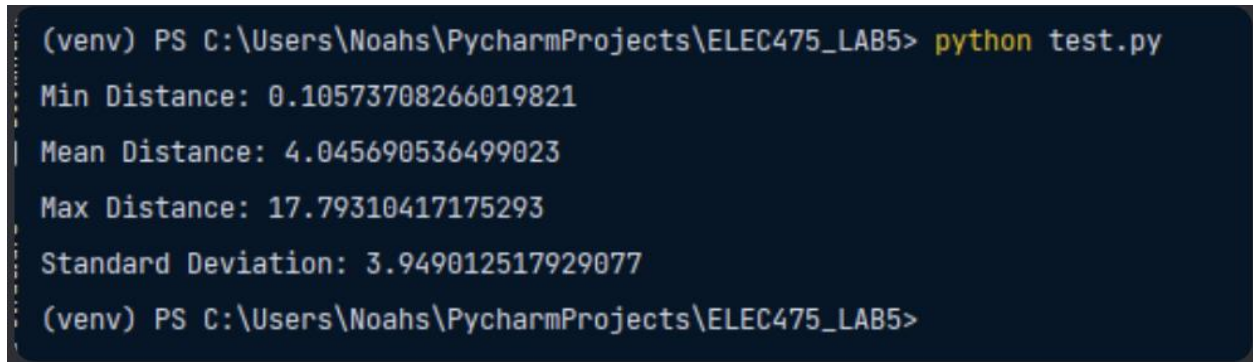


Figure 2: training loss

The loss rapidly decreases until around 6 epochs, and then a more gradual decrease of loss from 6 and beyond. It consistently decreases throughout the process and also stabilizes as it approaches around the 15 epoch mark, indicating proper model training assuming no overfitting. As we can see, anything more than 20 epochs would not decrease the loss substantially.

Step 4

Overall results:

A terminal window with a dark blue background and white text. The prompt is '(venv) PS C:\Users\Noahs\PycharmProjects\ELEC475_LAB5>'. The command 'python test.py' has been executed, resulting in four lines of output: 'Min Distance: 0.10573708266019821', 'Mean Distance: 4.045690536499023', 'Max Distance: 17.79310417175293', and 'Standard Deviation: 3.949012517929077'. The prompt is repeated at the bottom.

```
(venv) PS C:\Users\Noahs\PycharmProjects\ELEC475_LAB5> python test.py
Min Distance: 0.10573708266019821
Mean Distance: 4.045690536499023
Max Distance: 17.79310417175293
Standard Deviation: 3.949012517929077
(venv) PS C:\Users\Noahs\PycharmProjects\ELEC475_LAB5>
```

When tested, our model showed on average to be sufficiently accurate to meet our satisfaction; with a mean distance of 4.045690536499023 between the prediction point and the ground truth point. A good example of the typical (mean distance) output from our model can be seen in the image below.

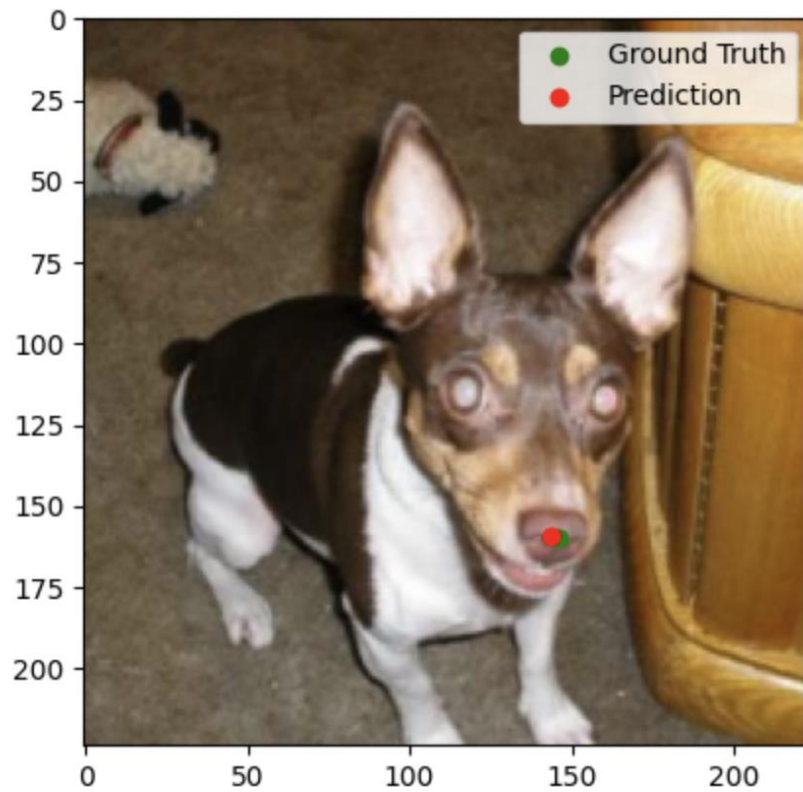


Figure 2: Output Image - Mean

Our model's tests had a minimum Distance of 0.10573708266019821 from the ground truth point, which shows that our model is very accurate, however since it is not microscopic, this means there is of course some room for improvement, which could be done by altering our model or making training more intense. Three example outputs from our model can be seen below. As you can see, in the first image you can't see any part of the green point whatsoever!

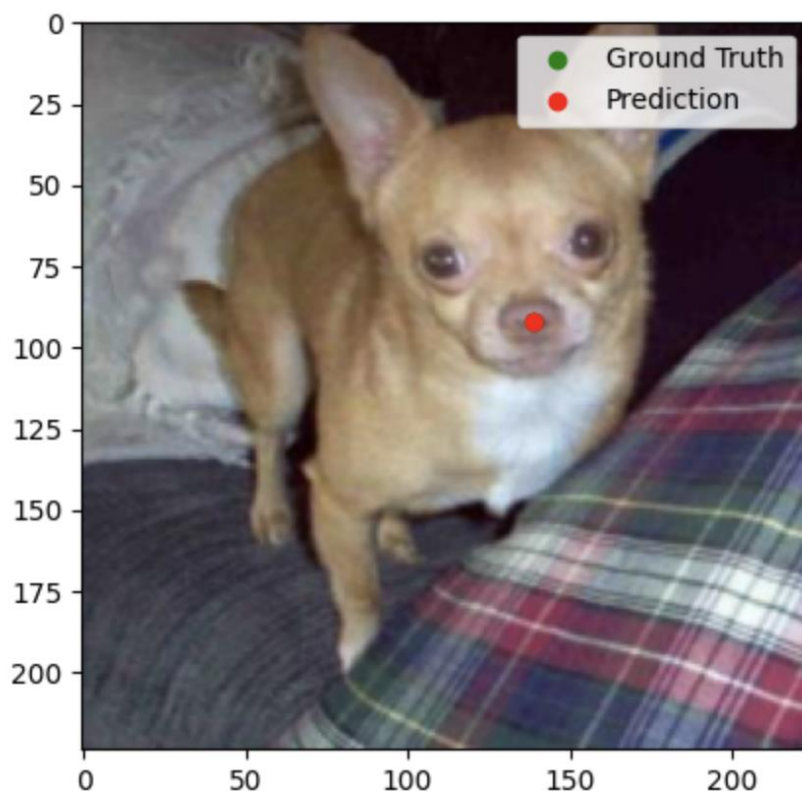


Figure 3: Output Image - Min

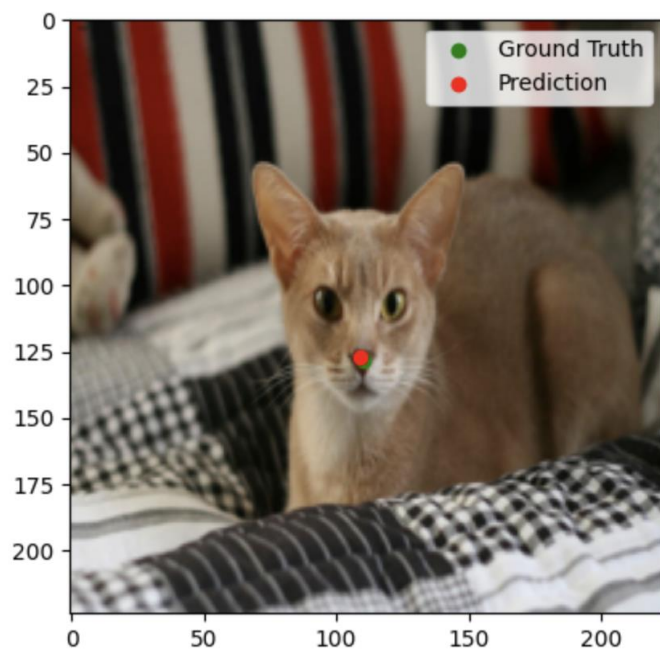


Figure 4: Output Image - Min

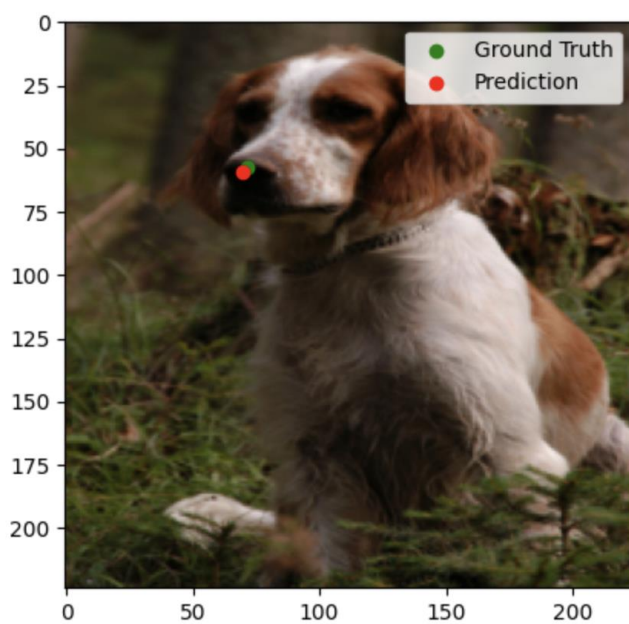


Figure 4: Output Image - Min

Maximum Distance

Our model tests showed a max distance of 17.79310417175293 from the ground truth point, which is slightly higher than I was hoping for, but there are almost always going to be some outliers when testing on data. The example output from our model can be seen below in *figure 5*.

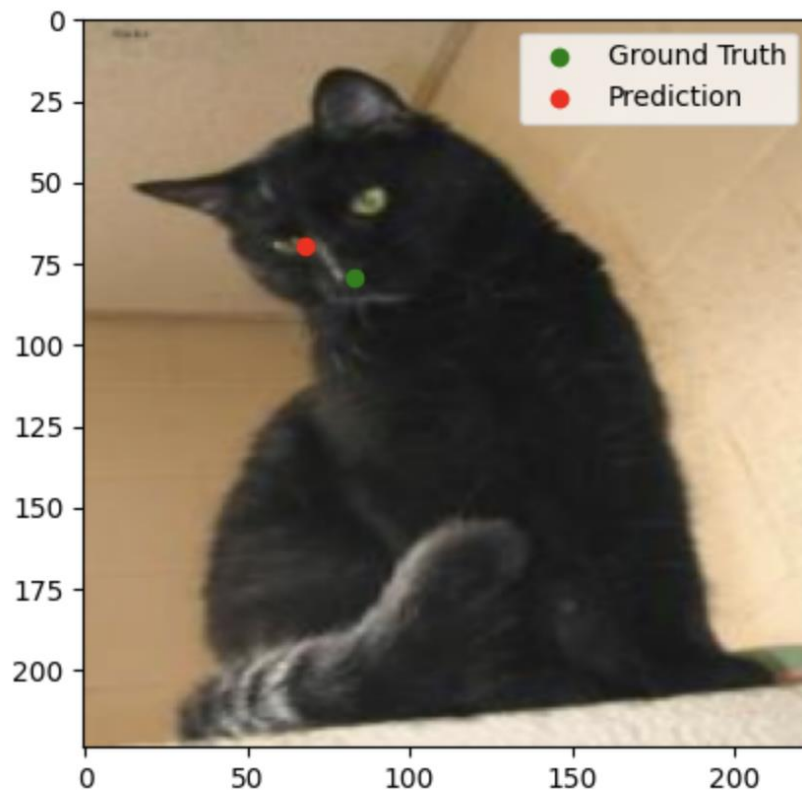


Figure 5: Output Image - Max

Similar to the mean distance discussed earlier; our model outputs had a standard deviation of the Euclidean distance of 3.949012517929077. The standard deviation of the Euclidean distance is a measure of the standard deviation of the straight-line distance using real values vectors.

Additionally, The hardware used to execute the inference tests was the same as what was used for training; an RTX 3090, 24 GB of VRAM and the time performance was around 8 msec per image.

Discussion

Expectations:

The performance of the Model was as expected. Incorporating ResNet proved to be successful architecturally for ensuring a high performance in nose localization. It allowed the model to train effectively, even when deep architectures are involved.

Difficulties

One of the major setbacks I had was involving our images coming out improper. Initially, the images were coming out in odd colours during testing; I had to manually denormalize them to return them to normal in our test.py code. Additionally, the temp.txt file was full of images that were corrupted, so I had to individually find out which ones were and manually remove them, which halted our process significantly. *Figure 6*, below, is an example of what our images with the modifications done for the training looked like.

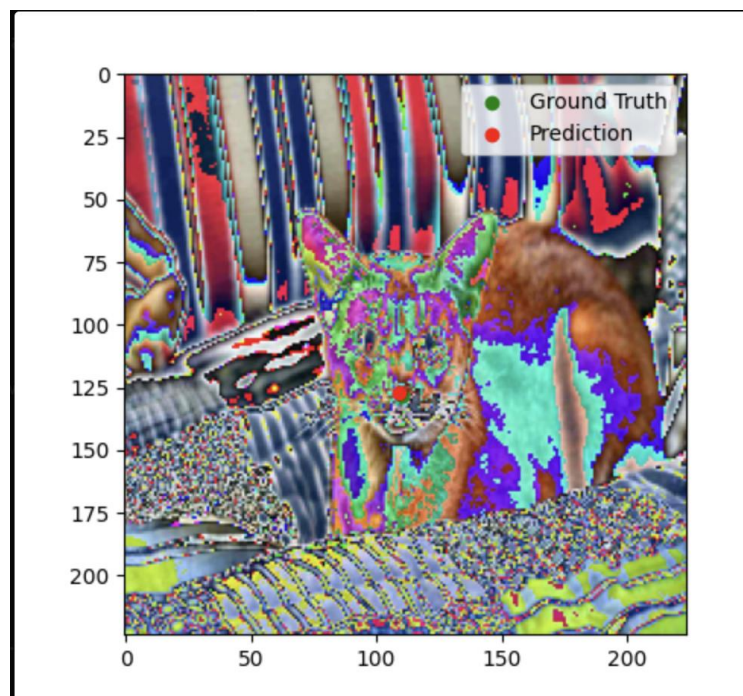


Figure 6: Dataset Image

Additionally, a major difficulty I went through was changing our model. In *figure 7* below, we can see the commented-out code, which is the original model which I had to discard. In the code below that we can see the changes that were made, such as the adaptive pool layer and the `pooled_features` section.

```
# class NoseNet(nn.Module):
#     def __init__(self):
#         super(NoseNet, self).__init__()
#         self.resnet = models.resnet18(weights=ResNet18_Weights.DEFAULT)
#         self.resnet = nn.Sequential(*(list(self.resnet.children())[:-1]))
#         self.voting_layer = nn.Linear(512, 2)
#
#     def forward(self, x):
#         feature_maps = self.resnet(x)
#         keypoint_location = self.voting_layer(feature_maps)
#         # vote_maps = torch.sigmoid(vote_maps) # Apply sigmoid for normalization
#         # keypoint_location = aggregate_votes(vote_maps)
#         return keypoint_location
class NoseNet(nn.Module):
    def __init__(self):
        super(NoseNet, self).__init__()
        self.resnet = models.resnet18(weights=ResNet18_Weights.DEFAULT)
        self.resnet = nn.Sequential(*(list(self.resnet.children())[:-1]))
        self.adaptive_pool = nn.AdaptiveAvgPool2d((1, 1)) # Add adaptive pooling
        self.voting_layer = nn.Linear(512, 2) # Output 2 values for x and y coordinates
    def forward(self, x):
        feature_maps = self.resnet(x)
        pooled_features = self.adaptive_pool(feature_maps)
        pooled_features = pooled_features.view(pooled_features.size(0), -1) # Flatten the features
        keypoints = self.voting_layer(pooled_features)
        return keypoints
```

Figure 7: Model change