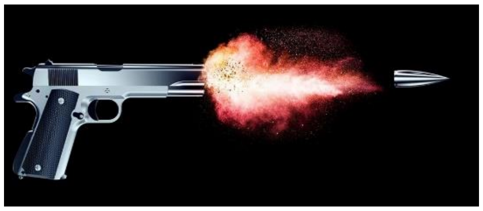
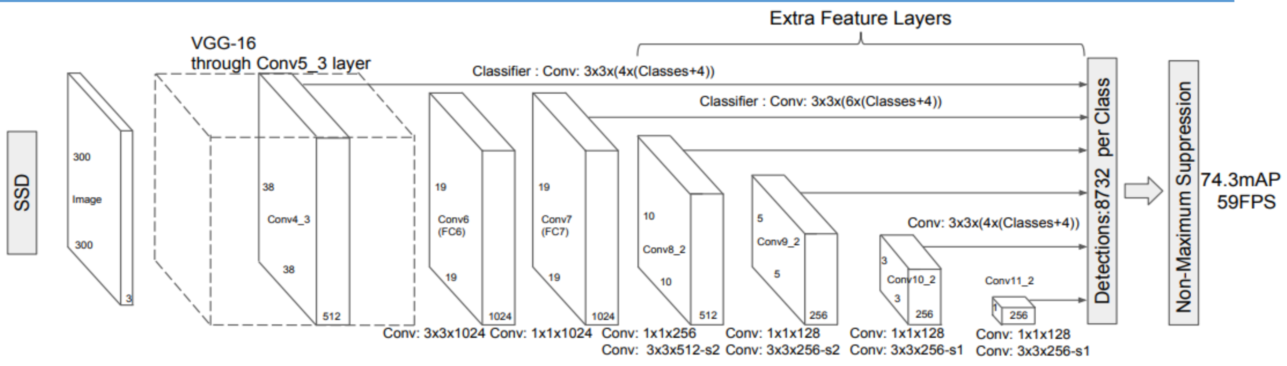
We choose SSD Mobilenet -> 315007120 + 318974557 = 633981677 -> 7
SSD with Mobilenet backbone

Single Shot Detector:

When we use SSD we only need one **single shot to detect** multiple objects within the image.



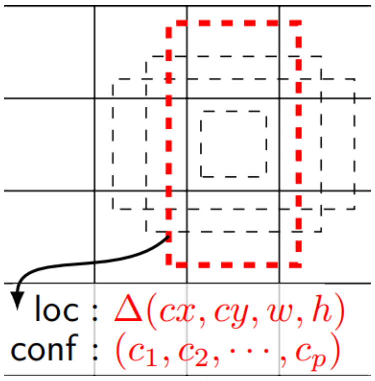
Our model is SSD with Mobile-Net backbone, which means that it's an improvement of the SSD architecture or in other words the backbone of the network is for feature extraction that are to be used for prediction in the later stages of the network. So as a result we will explain on the SSD network and then on the Mobile-Net network and will see if it is an improvement over the default.



SSD architecture:

the network is constructed by using the base layer (which the default is vgg-16 and ours will be Mobile-Net) to extract features and moving to a large array of smaller and smaller kernels of convoluted layers to detect objects.

Each layer will construct 4 bounding boxes, so if the layer k is m x n there will be (classes + 4) kmn outputs.
e.g. let's take layer conv4_3, k = 4, m, n = 38 and let's assume there are 10 classes.
Output: 38 x 38 x 4 x (10 + 4) = 80864
the amount of bounding boxes = 38 x 38 x 4 = 5776


loc : $\Delta(cx, cy, w, h)$
conf : $(c_1, c_2, \cdots, c_p)$

SSD Loss Function:

$$L(x,c,l,g) = \frac{1}{N}(L_{conf}(x,c) + \alpha L_{loc}(x,l,g))$$

The localization loss L_{loc} is the mismatch between the ground truth box and the predicted boundary box. SSD only penalizes predictions from positive matches. We want the predictions from the positive matches to get closer to the ground truth. Negative matches can be ignored.

$$L_{loc}(x,l,g) = \sum_{i\in Pos}^{N} \sum_{m\in\{cx,cy,w,h\}} x_{ij}^{k} \mathrm{smooth}_{\mathrm{L1}}(l_i^m - \hat{g}_j^m)$$

$$\hat{g}_j^{cx} = (g_j^{cx} - d_i^{cx})/d_i^w \qquad \hat{g}_j^{cy} = (g_j^{cy} - d_i^{cy})/d_i^h$$

$$\hat{g}_j^{w} = \log\left(\frac{g_j^w}{d_i^w}\right) \qquad \hat{g}_j^{h} = \log\left(\frac{g_j^h}{d_i^h}\right)$$

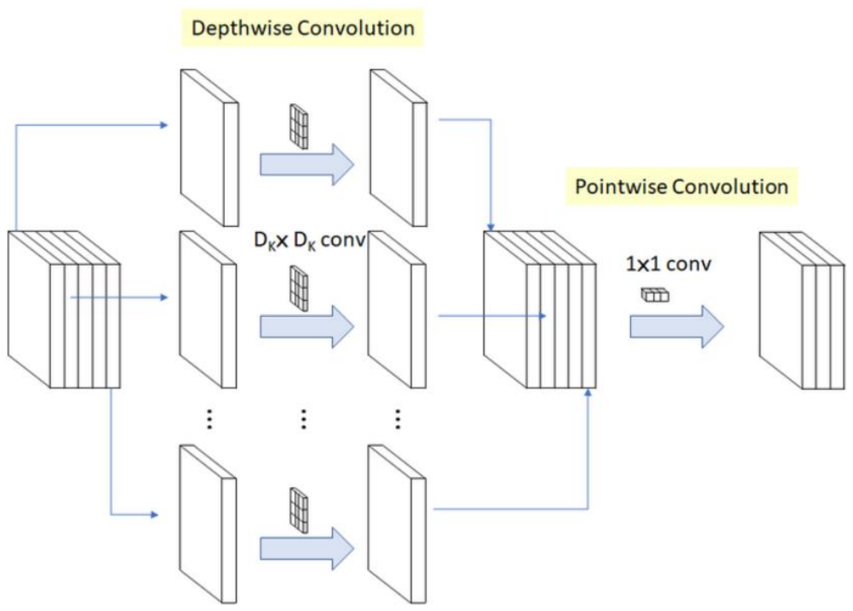
The confidence loss L_{conf} is the loss of making a class prediction. For every positive match prediction, we penalize the loss according to the confidence score of the corresponding class.

$$L_{conf}(x,c) = -\sum_{i\in Pos}^{N} x_{ij}^{p} log(\hat{c}_i^p) - \sum_{i\in Neg} log(\hat{c}_i^0) \quad \text{where} \quad \hat{c}_i^p = \frac{\exp(c_i^p)}{\sum_p \exp(c_i^p)}$$

In this network the base layer can be change according to the need of more accurate results or less computationally heavy.

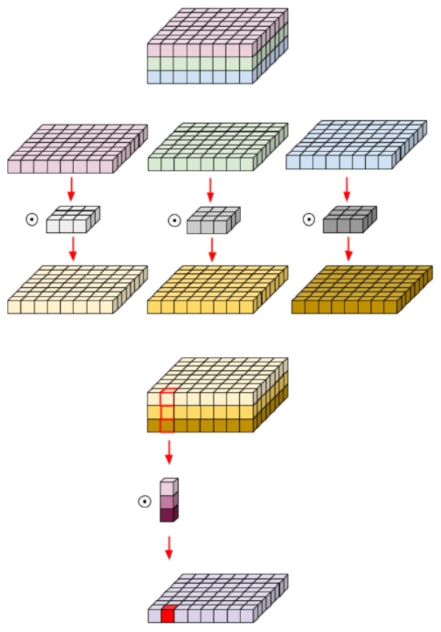
Mobile-Net:

As the name suggests this neural network is meant to be used for mobile devices and by so it is less computationally heavy but accurate none the less.



Mobile-Net architecture:

the mobile net network uses Depth-wise separable convolution. Depthwise Separable Convolution splits the computation into two steps: depthwise convolution applies a single convolutional filter per each input channel and pointwise convolution is used to create a linear combination of the output of the depthwise convolution. The comparison of standard convolution and depthwise separable convolution is shown to the right. As we can see we first divide the plain to his layers and conduct a conv layer on each point and then combine each resulting plain and perform a 1 x 1 conv layer on each resulting plain point.



Mobile-Net also uses global hyperparameters to effectively reduce the computational cost further.

- Width Multiplier: Thinner Models:

  For each layer, the width multiplier α will be multiplied with the input and the output channels (N and M) in order to narrow a network.

  $$D_K \cdot D_K \cdot \alpha M \cdot D_F \cdot D_F + \alpha M \cdot \alpha N \cdot D_F \cdot D_F$$

  Here α will vary from 0 to 1, with typical values of [1, 0.75, 0.5 and 0.25]

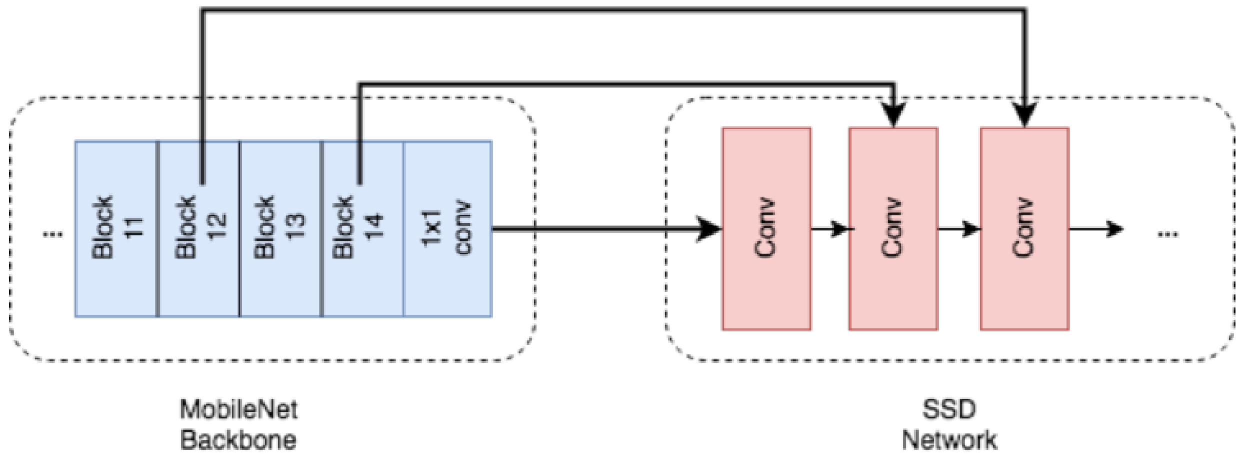
- Resolution Multiplier: Reduced Representation:

  For a given layer, the resolution multiplier ρ will be multiplied with the input feature map. we can express the computational cost by applying width multiplier and resolution multiplier as:

  $$D_K \cdot D_K \cdot \alpha M \cdot \rho D_F \cdot \rho D_F + \alpha M \cdot \alpha N \cdot \rho D_F \cdot \rho D_F$$

Mobile-Net SSD:

the network will be as follows:
first the Mobile-Net as the base layer of the network and then the resulting inputs will be given to the later layers in the SSD network.



Part 2: Implementation

Dataset

We used the Dogs dataset from the open images dataset, containing more than 30,000 dogs images separated into train, validation, and test sets.

Training Pipeline

We followed the implementation of https://github.com/qfgaohao/pytorch-ssd. We used the MobilenetV1 SSD architecture.
For the transfer learning, we downloaded the pre trained classification Mobilenet weights and used it to initialize the weights of the SSD base-net.
The augmentation used in training were random mirroring of the image and the bounding boxes, random cropping of the image together with the filtering of cropped out bounding boxes, resizing the image and bounding boxes to smaller size and filling the border with zeros, and photometric distort of the image (contrast, saturation, hue).

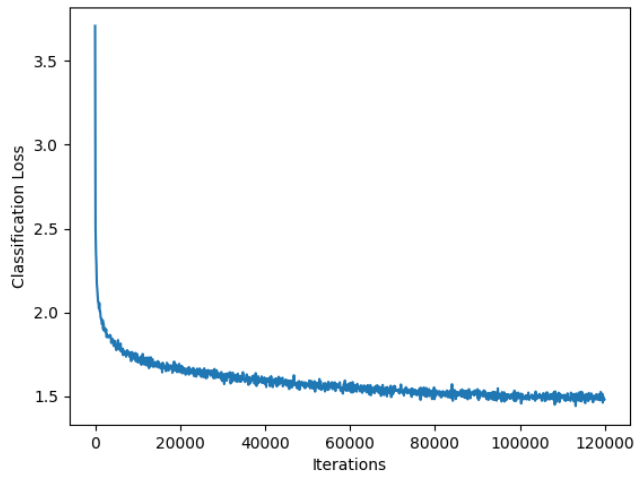
We run 2 main experiments:

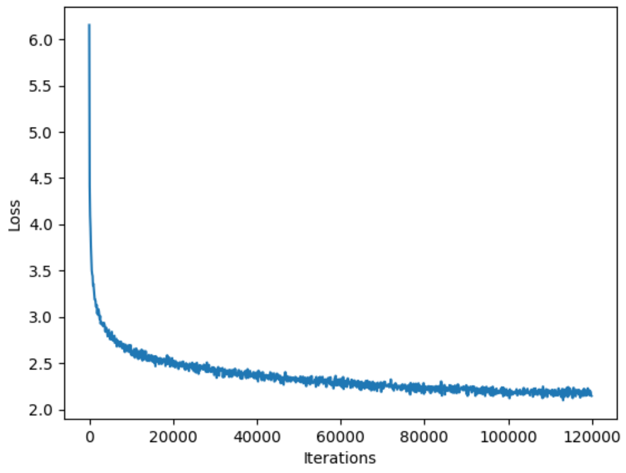
1. All the parameters are trained.
2. The base network parameters are frozen.

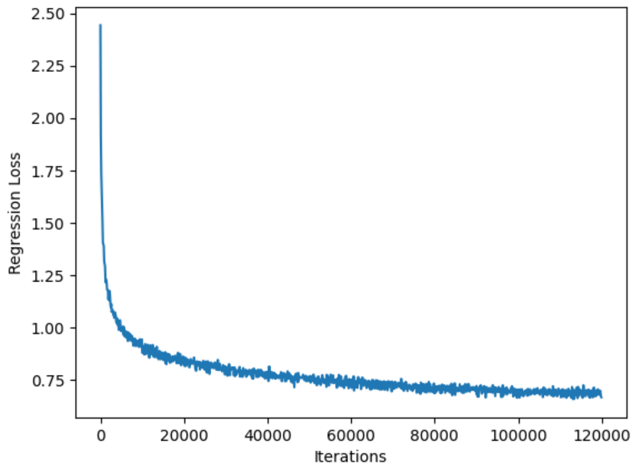
In both experiments, we used batch size 32, cosine learning rate scheduler, learning rate of 0.01, and a 0.001 learning rate for the base network.
The training loss consisted of a classification loss and a regression loss as mentioned in part 1.
To get the curves, we parsed the training logs (see files parse.py, parse_val.py).

The resulting training curves:

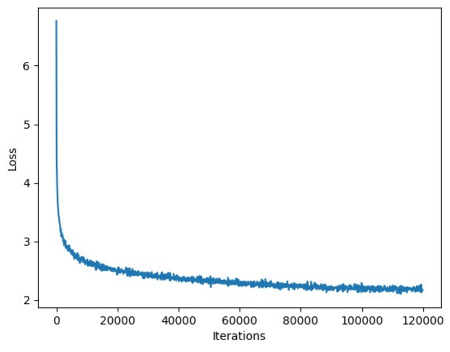
Experiment 1 (Training all parameters):

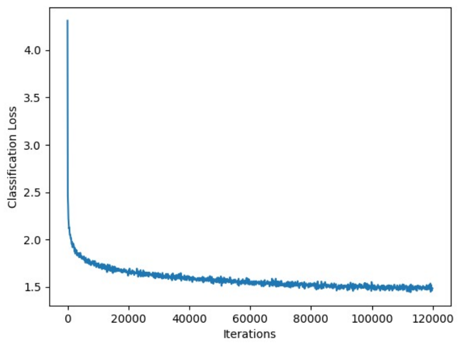


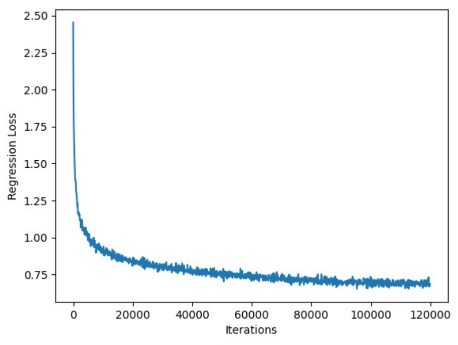




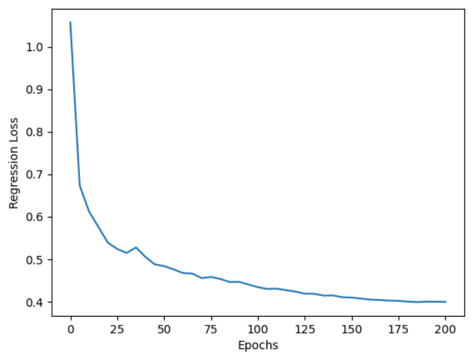
The training curves of experiment 2 (freezing the base net):

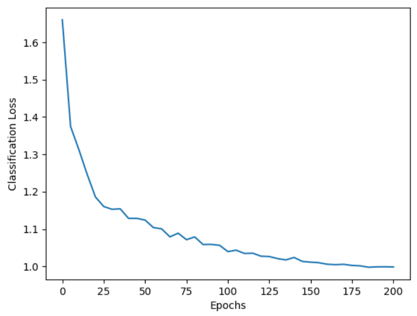


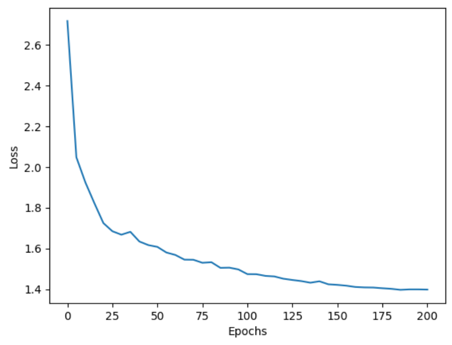




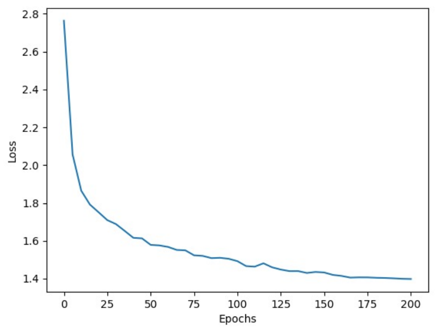
We run validation every 5 epochs. **The validation training curves:**
Experiment 1 (Training all parameters):

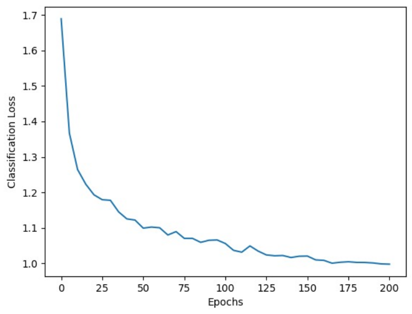


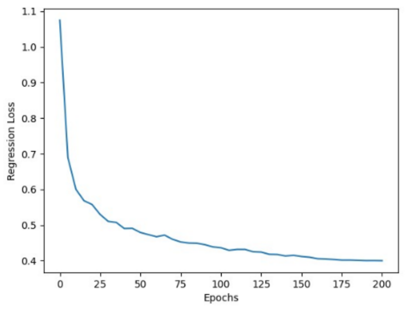




Experiment 2 (freeze base-net weights):





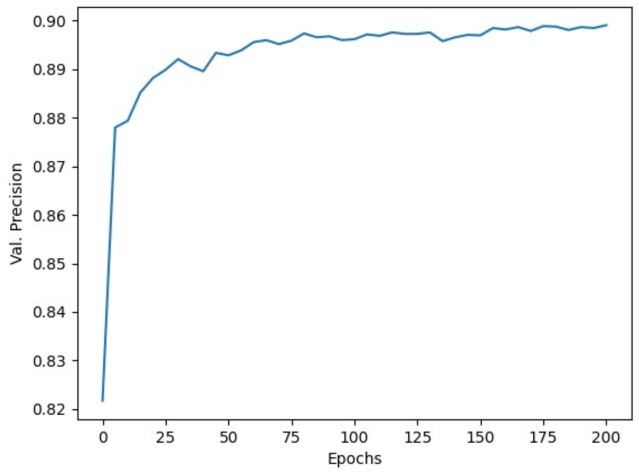


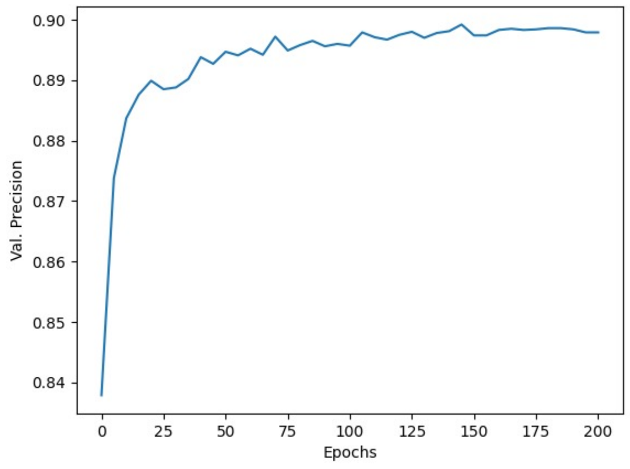
Precision during training

Finally, we compute the precision of the models during training.

The precision is defined as the ratio between the number of detected dogs and the total number of dogs in an image.

To do so we run the eval_ssd.py script while changing the dataset type to "validation" for each checkpoint saved after each validation run.
(Right picture is the second experiment and the left is the first experiment)

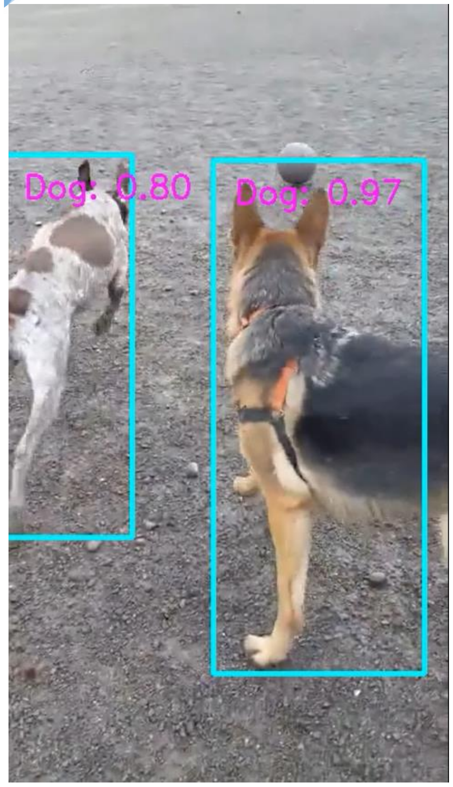


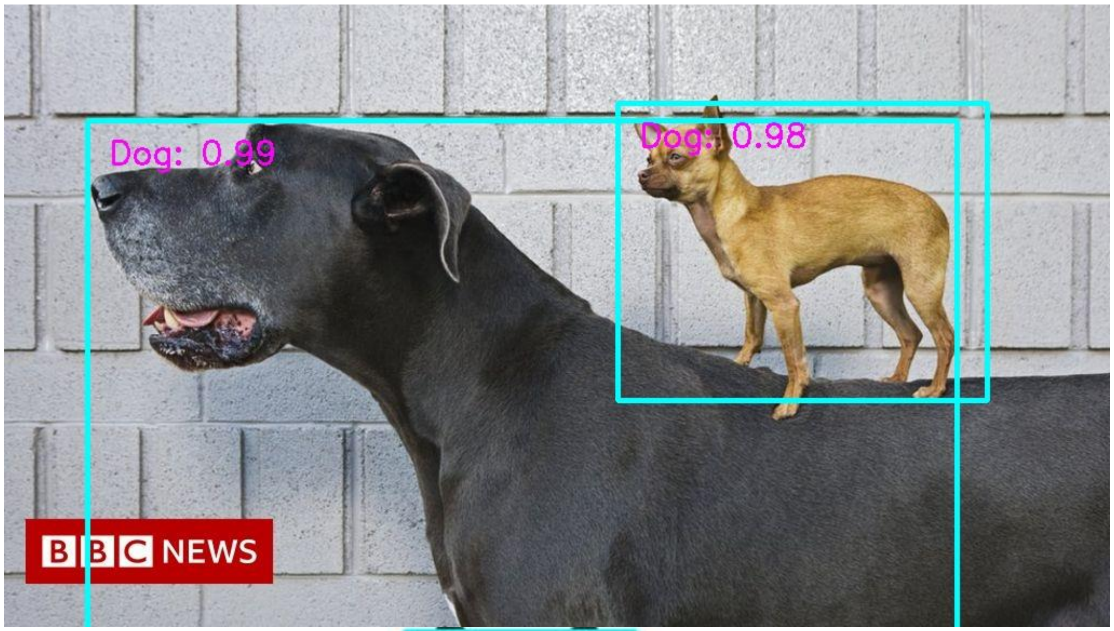


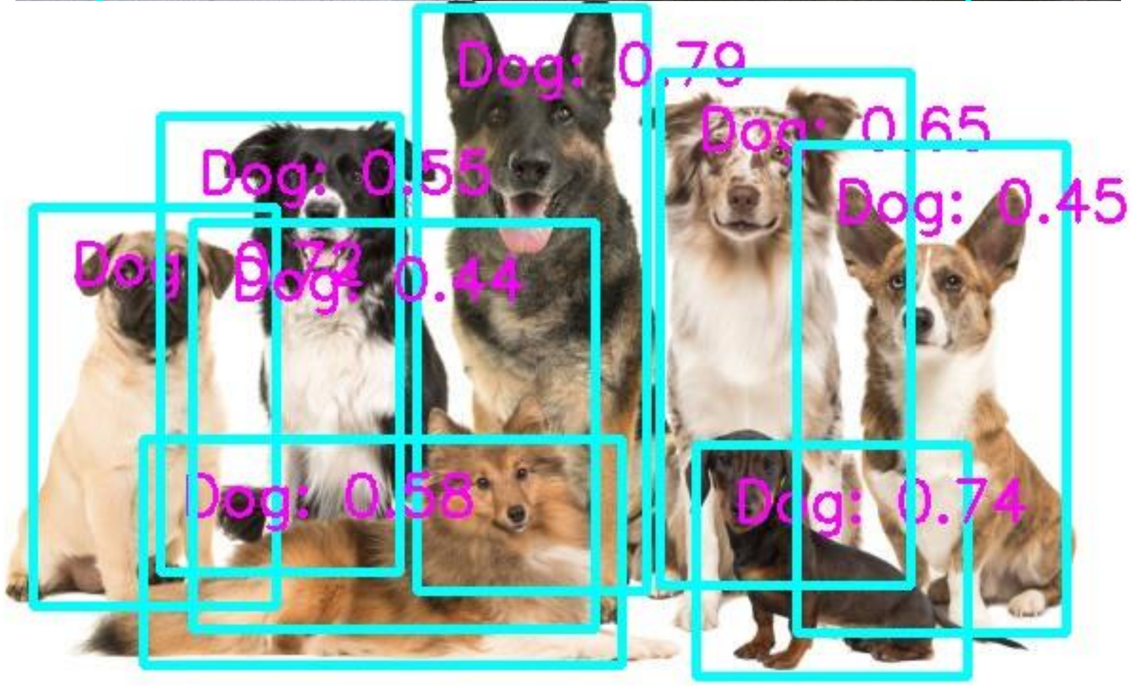
Results

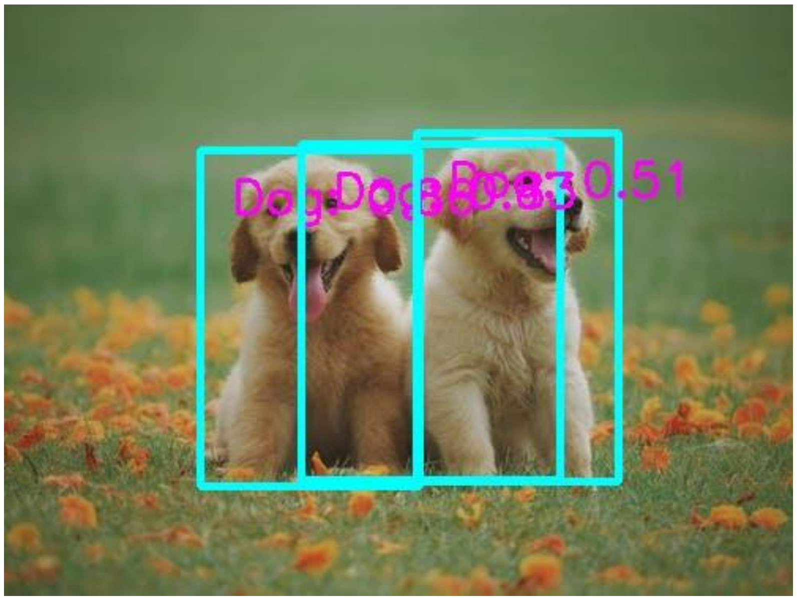
We run evaluation on the test set with the final model.
The precision obtained for experiment 1 was 0.89981 -> 89.981%.
The precision obtained for experiment 2 was 0.89948 - > 89.948%.
As can be seen by the training curves and precision, both models achieve comparable results, with experiment 1 obtaining slightly better precision.

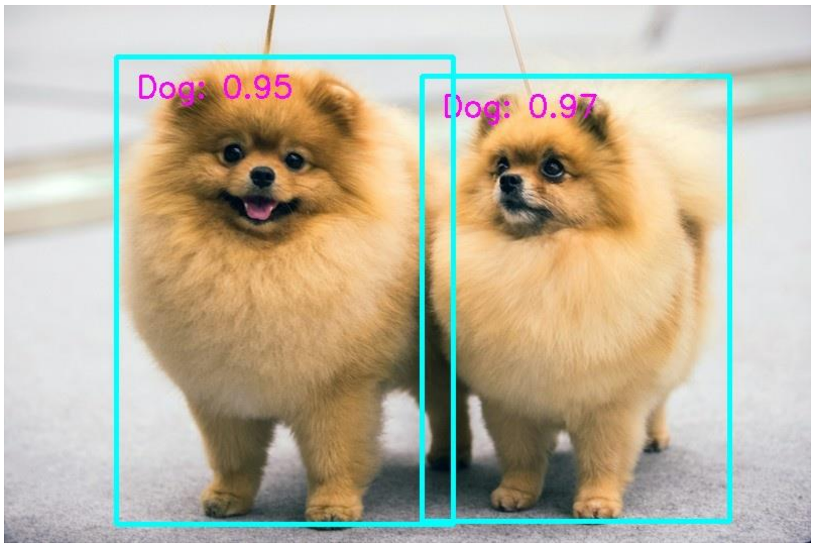
Example results:


Dog: 0.80
Dog: 0.97


Dog: 0.99
Dog: 0.98
BBC NEWS

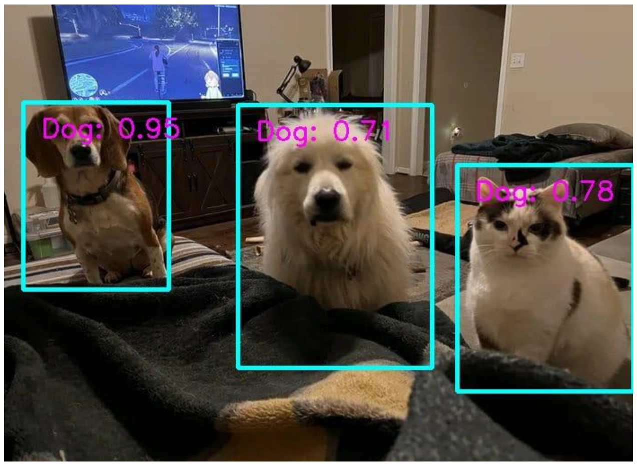

Dog: 0.79
Dog: 0.65
Dog: 0.55
Dog: 0.45
Dog: 0.72
Dog: 0.44
Dog: 0.58
Dog: 0.74

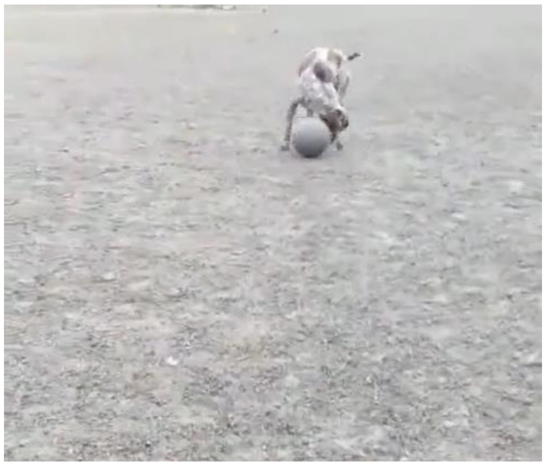

0.51


Dog: 0.95
Dog: 0.97

Fail cases:

Since we train the models for only dogs, it is possible that there were only a few images of cats (that should not be counted as dogs), and the model did not learn to this extent the difference between dogs and cats. Here is an example:





Another case is when the dog in the picture is small, and in the following example (taken from one of the submitted videos) its colors are also similar to the background.

Another problem that we noticed was in pictures that contains many little dogs. Sometimes a single box contains multiple dogs. We suspect it might be related to the NMS performed after the detection, which unite bounding boxes with large overlap.