Dog breed classifier

Pablo Serna

20 February 2020

Introduction - Overview

In this project, we faced the task of building an application that is able to determine if a picture contains a dog, a human or something else, and in the case of a dog or human, to which dog breed the individual would belong. This problem is part of a set of tasks called fine-grained category classification, a subset of problems in image classification. There are many other examples of this subset of problems, classifying the type of birds, plants, trees, etc. About dog breeds, the pioneering work on this problem was published by J. Liu *et al* [1], who created a first dataset of dog breeds by gathering pictures from Google, Flickr and ImageNet.

To address this problem we prepared a two-steps solution, Fig. 1. First, we used standard algorithms to determine if the picture indeed has a dog or a human. To detect a human in pictures we used a face detector implemented in OpenCV [2], based on Haar cascades [3]. Then for detecting whether there is a dog or not, we tried both VGG16 [4] and the state-of-the-art mobilenet v2 [5] algorithms. We kept finally mobilenet given its weight and efficiency. Finally, if any of the previous algorithms gives a positive answer, we provide the image to a custom network that classifies it into 133 possible breeds. The custom network is either a convolutional network trained from scratch or a network with the architecture of mobilenet v2, where we used transfer-learning techniques, adapted to 133 classes.

The first work in this task achieved 67% accuracy [1] and surpassing that value was our goal with our network. However the task is not easy, and training required some trial and error. In order to evaluate our model, we used accuracy on the test set, given that the task is multi-class classification. We also kept in mind performance and portability, by using architectures that do not occupy that much space and are efficient.

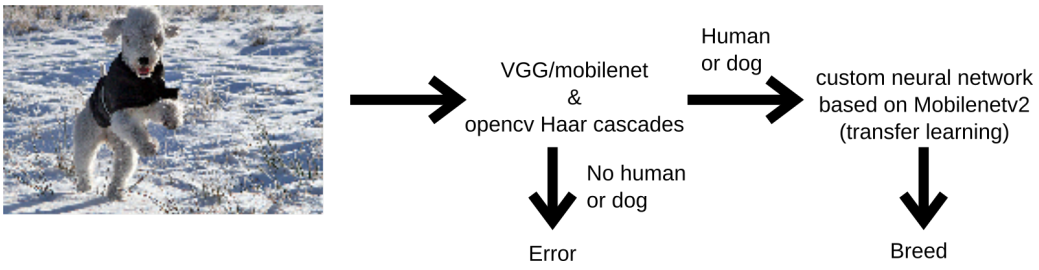

Figure 1: Pipeline scheme for two-steps solution for a dog breed classifier.

Analysis

Data exploration

We use two datasets in this project. One is a set of pictures of dogs, labeled by their breed, the other is a set of pictures of people, labeled by their names. This second set, Labeled Faces in the Wild (LFW) [6, 7], was used as a test-case to distinguish images from humans and dogs. The dataset contains 13233 images of humans and is available at (webpage). A few examples are shown in the first row of panels of Fig. 2. We have only used the first dataset (dog breeds dataset) for training the algorithms in this project.

Figure 2: Upper row - 4 samples of the human faces dataset, with the labels (their names) on top of the panels. Lower row - 4 samples of the dog dataset, with the labels (their breed) on top of the panels

We look now into more detail this set of 8351 pictures of dogs (lower row of panels in Fig. 2), classified in 133 breeds, that can be found in an Udacity's bucket in AWS. The dataset is split in a training set (80%), a validation set (10%) and a test set (10%). The number of images per class, in the training set, ranges from 26 to 77 with a relatively broad distribution Fig. 3 left panel. On average there are 50 images per breed, with a standard deviation of 12, Fig. 3 central panel. The classes are not completely well balanced since roughly 20% of the classes have less than 40 images, and 25% have more than 60 images. The class with the least number of images is the Xoloitzcuintli (upper-right panel), with 26 images. The breed with the most number of images is Alaskan Malamute (lower-right panel), with 77.

We can also check the characteristics of the images themselves. The images have 3 channels (RGB) and do not follow any rule for size. As a matter of fact, the distribution of sizes spans almost two orders of magnitude, as it is shown in the left panel of Fig.4, where each point corresponds to an image in the training dataset. Inspecting the average value of the different channels for each

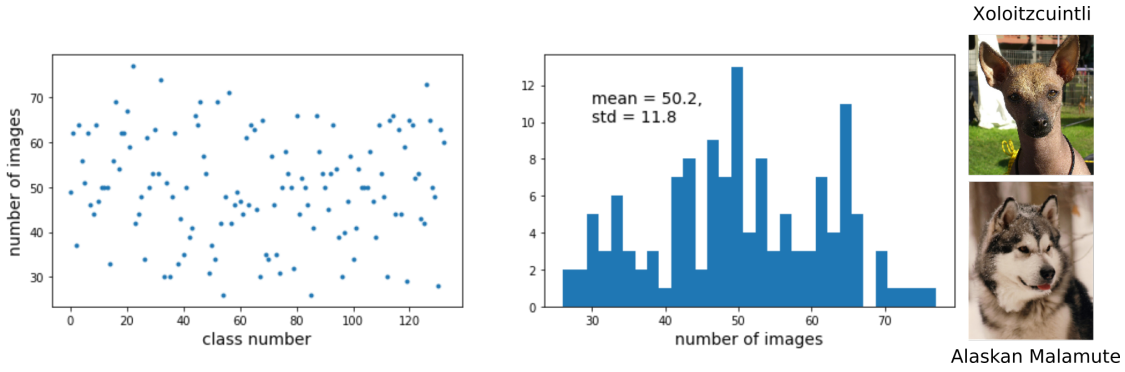

Figure 3: Left - Number of images for each class in the dataset. Center - histogram of the number of images per class, with mean 50.2 and standard deviation 11.8. Right - a sample of the less (top) and the most (bottom) represented breeds.

image, we can see that the red channel is systematically lower than the blue and green channel, as featured in their distribution, right panel Fig. 4. This had to be compensated when pre-processing the images for training. This could be a property of digital photos in general since this correction is used when providing data to well-established algorithms such as VGG, ResNets or mobilenet.

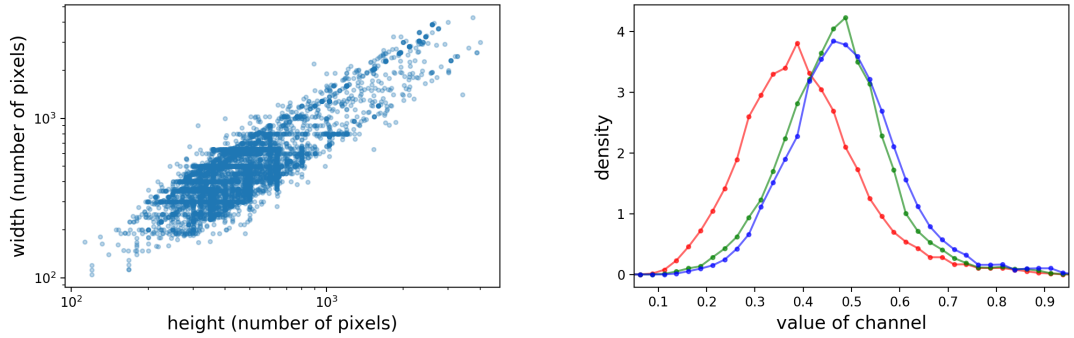

Figure 4: Left - width versus height of the images in the training set in logarithmic scale. Right - distribution of the average value of each channel per image.

Algorithms and techniques

Convolutional neural network from scratch

First, we developed a convolutional neural network (CNN) with a simple architecture: 4 convolutional layers (5×5) each one of them with a ReLU activation, and a max-pooling layer (5×5, stride 2 and same padding - it halves the spatial dimensions), and 2 fully-connected layers at the end with a ReLU in-between. We also add a dropout (probability 0.4) between the convolutions and the

fully-connected layers to ease overfitting (without being too strong). A more detailed description of the network can be extracted with the help of the library torchsummary

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 32, 224, 224]           2,432
              ReLU-2         [-1, 32, 224, 224]               0
         MaxPool2d-3         [-1, 32, 112, 112]               0
            Conv2d-4         [-1, 64, 112, 112]          51,264
              ReLU-5         [-1, 64, 112, 112]               0
         MaxPool2d-6           [-1, 64, 56, 56]               0
            Conv2d-7          [-1, 128, 56, 56]         204,928
              ReLU-8          [-1, 128, 56, 56]               0
         MaxPool2d-9          [-1, 128, 28, 28]               0
           Conv2d-10          [-1, 128, 28, 28]         409,728
             ReLU-11          [-1, 128, 28, 28]               0
        AvgPool2d-12            [-1, 128, 2, 2]               0
          Dropout-13                  [-1, 512]               0
           Linear-14                  [-1, 500]         256,500
             ReLU-15                  [-1, 500]               0
           Linear-16                  [-1, 133]          66,633
================================================================
Total params: 991,485
Trainable params: 991,485
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 49.78
Params size (MB): 3.78
Estimated Total Size (MB): 54.14
----------------------------------------------------------------
```

Training this network proved to be difficult. Initialization of weights was probably an issue since the training of the same network with the same hyperparameters would fail or succeed to converge (in the very first epochs), depending on the instance. Also, since it takes a long time to train a single epoch, it is hard to know whether the network is actually learning or not at the beginning. I had to do several attempts using both Udacity's provided server and my own personal computer (where I have an Nvidia RTX 2060). Because of this, I could not store properly the training and validation loss of the network. Also, I only realized I was missing the accuracy in the reporting messages when I tested the network. This was too late and I gave up in storing them.

In any case, the network reached 14% of accuracy in the test set after a few hours of training. This was enough to understand this approach requires much longer time of computation. Therefore we moved to a different approach.

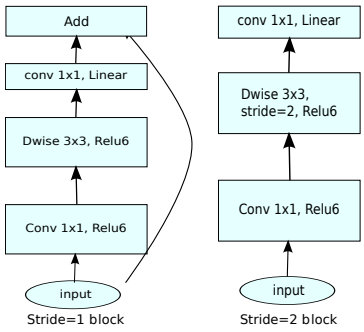

Figure 5: Bottleneck block for mobilenetv2 (extracted from [5])

Transfer learning

The technique of transfer learning consists in taking a successful pre-trained architecture, and use the weights to provide a starting point for training. The architecture can be modified, for example adding more layers or substituting any of the layers of the model. In our case, we will take one of the models for image classification, from the library *model* in PyTorch, and substitute the last layer that classifies into 1000 classes for another layer that does the same but to 133 classes.

Here, I tried two different models: Mobilenetv2[5] and ResNet16 [8]. While ResNet16 was training in Udacity's notebook, Mobilenetv2 was being trained on my laptop. Residual Networks or ResNets were a revolution when they were introduced. The concept of skip connection was indeed a key factor to be able to increase the depth of neural networks. Then with regularization techniques such as batch normalization, their training became easier and more accessible. Their derivates are top performers in the ImageNet classification task. However, in contrast with Mobilenetv2 the different versions of ResNets are heavy and not very efficient. Mobilenetv2 is a state-of-the-art algorithm where the performance does not only takes into account the accuracy, which might be lower than some ResNets, but it also has the possibility of using it in devices such as phones or even CPUs without GPUs. During the training of both algorithms, the training loss of Mobilenetv2 decreased much faster than ResNet16, not only per epoch, but also partially due to the fact it takes less time to train an epoch. Given that I had a restrictive time limit, I focused on a mobilenet-based architecture.

The architecture of Mobilenetv2 is designed for performance. We describe it here briefly following its developers [5]. It utilizes blocks called bottleneck, Fig. 5, that can be repeated several times. In Table 1, we can see the architecture of the neural network (reproduced from the original paper [5]). In the table each line describes a sequence of 1 or more identical (except for stride) layers, repeated n times. All layers in the same sequence have c output channels. The first layer of each sequence has a stride s, and the rest have stride 1. Spatial convolutions use 3×3 kernels.

Benchmark

Training this network was easier than for the model developed from scratch. Both training and validation loss decayed rather smoothly for at least 70 epochs, Fig. 6 left panel. We can see that validation loss (orange line) was saturating around epoch 50. This is also confirmed in the

Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

Table 1: Summary of the architecture of mobilenetv2 (extracted from [5])

accuracy, Fig. 6 right panel. Unsurprisingly, it took our network less than 1 epoch to surpass our model developed from scratch. After 36 epochs, the model had already reached 67%, our goal. However for later epochs, the accuracy increased more slowly, without reaching saturation before epoch 70.

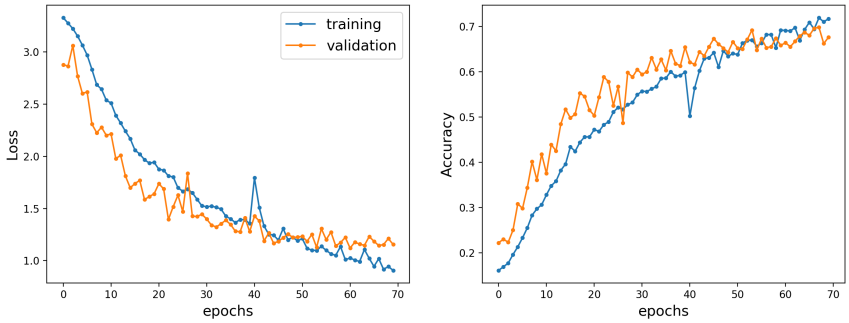

Figure 6: Left - Loss as a function of the epochs for training of the transferred-learning network both for the training (validation) and validation sets (orange). Right - Accuracy as a function of the epochs for training (validation) and validation sets (orange).

Overfitting is a possibility to which we would face if we keep training the network with the same conditions. Possibly including more aggressive augmentation or using other techniques could help to improve on this point.

Methodology

Data preprocessing

For preprocessing data, we do a series of transformations,

- The image is resized to 256x256 pixels to standardize all the images.
- A random section of the image of 224x224 pixels is cropped.
- A rotation with a random angle from $-\theta_{\max}$ to $\theta_{\max}$ is applied, where $\theta_{\max} = 30$ degrees.
- Possible vertical and/or horizontal flips are applied.
- Color jitter is added in the image, small enough so that it cannot be seen.
- The image is transformed into a tensor variable.
- The image is normalized, using the averages and standard deviations that were used in ImageNet dataset, much larger than our dog dataset.
- Small (additive) Gaussian noise, with standard deviation of 0.05, is applied.

After this series of transformations, images are provided to the network for training purposes. We can visualize what the protocol does to the images by reversing the normalization procedure, Fig. 7

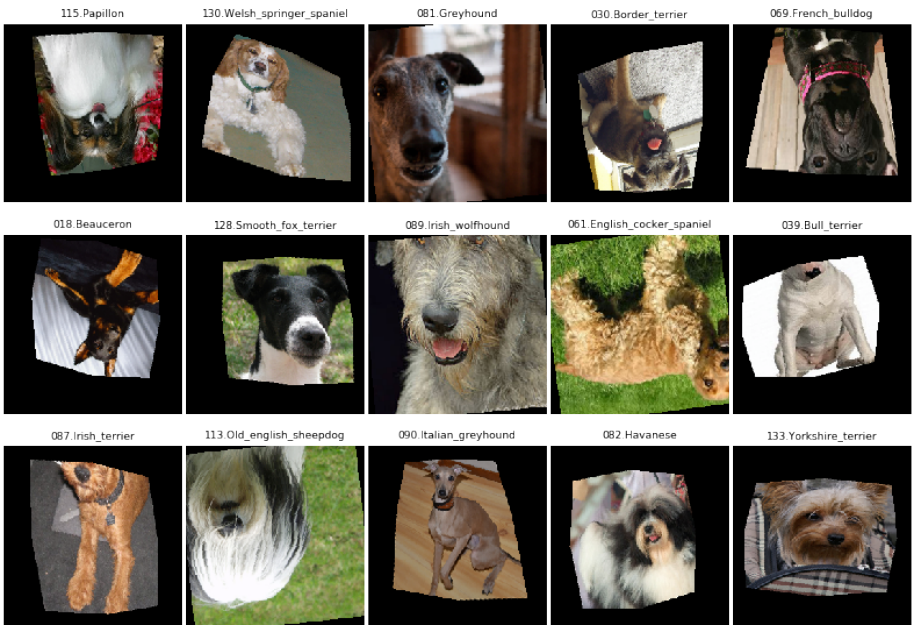

Figure 7: Examples of augmented images of dogs.

For the validation and test set the pre-processing part does not use augmentation and it consists only in: resizing images, cropping the center, transforming them to tensors and normalization.

Implementation details and Refinement

For the model developed from scratch, originally, I had in mind creating an evolutionary algorithm and training several neural networks with different architectures. For that, I programmed a subroutine to create models with different architecture with three arrays as inputs. However, once I started training the model, I realized that I was way too optimistic. Training this dataset takes a long time even with decent graphic cards and loss did not decrease fast, particularly when the network is deep. Because of that, I went back to the simple architecture described in section Algorithms and techniques, although now it was programmed with this small subroutine.

For the models using transfer learning. I left training the ResNet-based network during the time the mobilenet-based network was training too. The performance of the first network was relatively bad compared to the second, with accuracy between 40-50% (in around 40 epochs).

Tuning data augmentation to avoid overfitting was problematic, both because for the model from scratch it was an obstacle, and for the model with transfer learning it was not enough. However even using data augmentation, our mobilenet-based network was close to overfitting the training set. How could we avoid this without increasing the dataset? We could think about a different approach creating a triple Generative Adversarial Network: with a generator, a discriminator and a classifier (our mobilenet-based network).

Results

Model evaluation and validation

As stated in the Benchmark and architecture description, the validation accuracy reached 70%, while training accuracy 72%. The behaviour of the curves, Fig. 6, suggest that the network was starting to overfit the training dataset. The performance in the test set is 68% accuracy. This is slightly ahead of the benchmark by [1], but with only 70 epochs. Qualitatively the model seems to perform well, although it gets confused when the dog does not belong to any breed, consider Fig. 8.

Performance can also be observed in other statistical features. The algorithm is relatively fast, single image classification takes 9.38 ms $\pm$ 51.1 µs (with Nvidia RTX 2060). The weight of the network is 162.57 MB (see appendix).

Testing the model with pictures obtained from wikipedia, I observed that the model fails a lot with specific breeds, for instance with border collies, particularly when they are not black and white. This particular example is easy to understand, inspecting the training dataset, there is no picture of a border collie that is not black and white. However this suggests that the dataset could be expanded adding more variability. It also failed with the breed Daschund, which seems to be better represented in the training dataset. In order to solve this issue we may need to approach the problem from a different perspective, even considering the possibility of using GANs.

Ending remarks

We have indeed surpassed the benchmark of [1] with the added bonus that our application can distinguish first between humans and dogs. The model however can still be improved with more time of training, which was restricted for this particular project, and possibly different approaches or more data. In particular, an advantage of the solution proposed here is that we provide a very

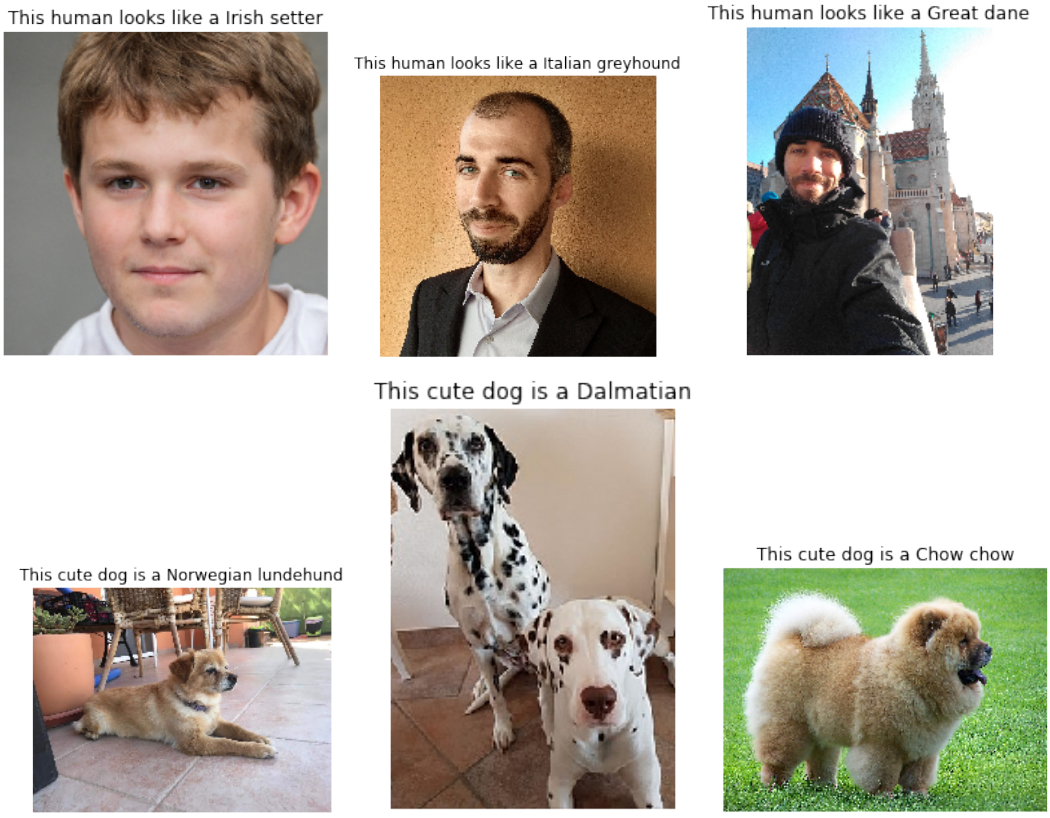

Figure 8: Output of developed app for 6 custom images. Upper row - left panel is a human generated by a GAN and the other two pictures is the author. Lower row - left panel is author's dog (no breed!) and the other two have been extracted from wikimedia.

light and fast network that can be used to run in mobile phone in a small app. This could be a next step after the completion of this project.

References

[1] Jiongxin Liu, Angjoo Kanazawa, David Jacobs, and Peter Belhumeur. Dog breed classification using part localization. In *European conference on computer vision*, pages 172–185. Springer, 2012.

[2] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[3] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001*, volume 1, pages I–I. IEEE, 2001.

[4] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[5] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[6] Gary B Huang, Marwan Mattar, Tamara Berg, and Eric Learned-Miller. Labeled faces in the wild: A database for studying face recognition in unconstrained environments. 2008.

[7] Gary B Huang and Erik Learned-Miller. Labeled faces in the wild: Updates and new reporting procedures. *Dept. Comput. Sci., Univ. Massachusetts Amherst, Amherst, MA, USA, Tech. Rep*, pages 14–003, 2014.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Appendix. Mobilenet v2 architecture

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 32, 112, 112]	864
BatchNorm2d-2	[-1, 32, 112, 112]	64
ReLU6-3	[-1, 32, 112, 112]	0
Conv2d-4	[-1, 32, 112, 112]	288
BatchNorm2d-5	[-1, 32, 112, 112]	64
ReLU6-6	[-1, 32, 112, 112]	0
Conv2d-7	[-1, 16, 112, 112]	512
BatchNorm2d-8	[-1, 16, 112, 112]	32
InvertedResidual-9	[-1, 16, 112, 112]	0
Conv2d-10	[-1, 96, 112, 112]	1,536
BatchNorm2d-11	[-1, 96, 112, 112]	192
ReLU6-12	[-1, 96, 112, 112]	0
Conv2d-13	[-1, 96, 56, 56]	864
BatchNorm2d-14	[-1, 96, 56, 56]	192
ReLU6-15	[-1, 96, 56, 56]	0
Conv2d-16	[-1, 24, 56, 56]	2,304
BatchNorm2d-17	[-1, 24, 56, 56]	48
InvertedResidual-18	[-1, 24, 56, 56]	0
Conv2d-19	[-1, 144, 56, 56]	3,456
BatchNorm2d-20	[-1, 144, 56, 56]	288
ReLU6-21	[-1, 144, 56, 56]	0
Conv2d-22	[-1, 144, 56, 56]	1,296
BatchNorm2d-23	[-1, 144, 56, 56]	288
ReLU6-24	[-1, 144, 56, 56]	0
Conv2d-25	[-1, 24, 56, 56]	3,456
BatchNorm2d-26	[-1, 24, 56, 56]	48
InvertedResidual-27	[-1, 24, 56, 56]	0
Conv2d-28	[-1, 144, 56, 56]	3,456
BatchNorm2d-29	[-1, 144, 56, 56]	288
ReLU6-30	[-1, 144, 56, 56]	0
Conv2d-31	[-1, 144, 28, 28]	1,296
BatchNorm2d-32	[-1, 144, 28, 28]	288
ReLU6-33	[-1, 144, 28, 28]	0
Conv2d-34	[-1, 32, 28, 28]	4,608
BatchNorm2d-35	[-1, 32, 28, 28]	64
InvertedResidual-36	[-1, 32, 28, 28]	0
Conv2d-37	[-1, 192, 28, 28]	6,144
BatchNorm2d-38	[-1, 192, 28, 28]	384
ReLU6-39	[-1, 192, 28, 28]	0
Conv2d-40	[-1, 192, 28, 28]	1,728
BatchNorm2d-41	[-1, 192, 28, 28]	384

```
           ReLU6-42         [-1, 192, 28, 28]               0
         Conv2d-43          [-1, 32, 28, 28]           6,144
     BatchNorm2d-44          [-1, 32, 28, 28]              64
InvertedResidual-45          [-1, 32, 28, 28]               0
          Conv2d-46         [-1, 192, 28, 28]           6,144
     BatchNorm2d-47         [-1, 192, 28, 28]             384
           ReLU6-48         [-1, 192, 28, 28]               0
          Conv2d-49         [-1, 192, 28, 28]           1,728
     BatchNorm2d-50         [-1, 192, 28, 28]             384
           ReLU6-51         [-1, 192, 28, 28]               0
          Conv2d-52          [-1, 32, 28, 28]           6,144
     BatchNorm2d-53          [-1, 32, 28, 28]              64
InvertedResidual-54          [-1, 32, 28, 28]               0
          Conv2d-55         [-1, 192, 28, 28]           6,144
     BatchNorm2d-56         [-1, 192, 28, 28]             384
           ReLU6-57         [-1, 192, 28, 28]               0
          Conv2d-58         [-1, 192, 14, 14]           1,728
     BatchNorm2d-59         [-1, 192, 14, 14]             384
           ReLU6-60         [-1, 192, 14, 14]               0
          Conv2d-61          [-1, 64, 14, 14]          12,288
     BatchNorm2d-62          [-1, 64, 14, 14]             128
InvertedResidual-63          [-1, 64, 14, 14]               0
          Conv2d-64         [-1, 384, 14, 14]          24,576
     BatchNorm2d-65         [-1, 384, 14, 14]             768
           ReLU6-66         [-1, 384, 14, 14]               0
          Conv2d-67         [-1, 384, 14, 14]           3,456
     BatchNorm2d-68         [-1, 384, 14, 14]             768
           ReLU6-69         [-1, 384, 14, 14]               0
          Conv2d-70          [-1, 64, 14, 14]          24,576
     BatchNorm2d-71          [-1, 64, 14, 14]             128
InvertedResidual-72          [-1, 64, 14, 14]               0
          Conv2d-73         [-1, 384, 14, 14]          24,576
     BatchNorm2d-74         [-1, 384, 14, 14]             768
           ReLU6-75         [-1, 384, 14, 14]               0
          Conv2d-76         [-1, 384, 14, 14]           3,456
     BatchNorm2d-77         [-1, 384, 14, 14]             768
           ReLU6-78         [-1, 384, 14, 14]               0
          Conv2d-79          [-1, 64, 14, 14]          24,576
     BatchNorm2d-80          [-1, 64, 14, 14]             128
InvertedResidual-81          [-1, 64, 14, 14]               0
          Conv2d-82         [-1, 384, 14, 14]          24,576
     BatchNorm2d-83         [-1, 384, 14, 14]             768
           ReLU6-84         [-1, 384, 14, 14]               0
          Conv2d-85         [-1, 384, 14, 14]           3,456
     BatchNorm2d-86         [-1, 384, 14, 14]             768
           ReLU6-87         [-1, 384, 14, 14]               0
```

```
           Conv2d-88          [-1, 64, 14, 14]          24,576
      BatchNorm2d-89          [-1, 64, 14, 14]             128
 InvertedResidual-90          [-1, 64, 14, 14]               0
           Conv2d-91         [-1, 384, 14, 14]          24,576
      BatchNorm2d-92         [-1, 384, 14, 14]             768
             ReLU6-93         [-1, 384, 14, 14]               0
           Conv2d-94         [-1, 384, 14, 14]           3,456
      BatchNorm2d-95         [-1, 384, 14, 14]             768
             ReLU6-96         [-1, 384, 14, 14]               0
           Conv2d-97          [-1, 96, 14, 14]          36,864
      BatchNorm2d-98          [-1, 96, 14, 14]             192
 InvertedResidual-99          [-1, 96, 14, 14]               0
          Conv2d-100         [-1, 576, 14, 14]          55,296
     BatchNorm2d-101         [-1, 576, 14, 14]           1,152
            ReLU6-102         [-1, 576, 14, 14]               0
          Conv2d-103         [-1, 576, 14, 14]           5,184
     BatchNorm2d-104         [-1, 576, 14, 14]           1,152
            ReLU6-105         [-1, 576, 14, 14]               0
          Conv2d-106          [-1, 96, 14, 14]          55,296
     BatchNorm2d-107          [-1, 96, 14, 14]             192
InvertedResidual-108          [-1, 96, 14, 14]               0
          Conv2d-109         [-1, 576, 14, 14]          55,296
     BatchNorm2d-110         [-1, 576, 14, 14]           1,152
            ReLU6-111         [-1, 576, 14, 14]               0
          Conv2d-112         [-1, 576, 14, 14]           5,184
     BatchNorm2d-113         [-1, 576, 14, 14]           1,152
            ReLU6-114         [-1, 576, 14, 14]               0
          Conv2d-115          [-1, 96, 14, 14]          55,296
     BatchNorm2d-116          [-1, 96, 14, 14]             192
InvertedResidual-117          [-1, 96, 14, 14]               0
          Conv2d-118         [-1, 576, 14, 14]          55,296
     BatchNorm2d-119         [-1, 576, 14, 14]           1,152
            ReLU6-120         [-1, 576, 14, 14]               0
          Conv2d-121           [-1, 576, 7, 7]           5,184
     BatchNorm2d-122           [-1, 576, 7, 7]           1,152
            ReLU6-123           [-1, 576, 7, 7]               0
          Conv2d-124           [-1, 160, 7, 7]          92,160
     BatchNorm2d-125           [-1, 160, 7, 7]             320
InvertedResidual-126           [-1, 160, 7, 7]               0
          Conv2d-127           [-1, 960, 7, 7]         153,600
     BatchNorm2d-128           [-1, 960, 7, 7]           1,920
            ReLU6-129           [-1, 960, 7, 7]               0
          Conv2d-130           [-1, 960, 7, 7]           8,640
     BatchNorm2d-131           [-1, 960, 7, 7]           1,920
            ReLU6-132           [-1, 960, 7, 7]               0
          Conv2d-133           [-1, 160, 7, 7]         153,600
```

```
     BatchNorm2d-134            [-1, 160, 7, 7]             320
InvertedResidual-135            [-1, 160, 7, 7]               0
          Conv2d-136            [-1, 960, 7, 7]         153,600
     BatchNorm2d-137            [-1, 960, 7, 7]           1,920
           ReLU6-138            [-1, 960, 7, 7]               0
          Conv2d-139            [-1, 960, 7, 7]           8,640
     BatchNorm2d-140            [-1, 960, 7, 7]           1,920
           ReLU6-141            [-1, 960, 7, 7]               0
          Conv2d-142            [-1, 160, 7, 7]         153,600
     BatchNorm2d-143            [-1, 160, 7, 7]             320
InvertedResidual-144            [-1, 160, 7, 7]               0
          Conv2d-145            [-1, 960, 7, 7]         153,600
     BatchNorm2d-146            [-1, 960, 7, 7]           1,920
           ReLU6-147            [-1, 960, 7, 7]               0
          Conv2d-148            [-1, 960, 7, 7]           8,640
     BatchNorm2d-149            [-1, 960, 7, 7]           1,920
           ReLU6-150            [-1, 960, 7, 7]               0
          Conv2d-151            [-1, 320, 7, 7]         307,200
     BatchNorm2d-152            [-1, 320, 7, 7]             640
InvertedResidual-153            [-1, 320, 7, 7]               0
          Conv2d-154           [-1, 1280, 7, 7]         409,600
     BatchNorm2d-155           [-1, 1280, 7, 7]           2,560
           ReLU6-156           [-1, 1280, 7, 7]               0
         Dropout-157                 [-1, 1280]               0
          Linear-158                  [-1, 133]         170,373
================================================================
Total params: 2,394,245
Trainable params: 2,394,245
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 152.86
Params size (MB): 9.13
Estimated Total Size (MB): 162.57
----------------------------------------------------------------
```