Capstone Project
Machine Learning Engineer Nanodegree

Agustín Ávila Marín
November 25, 2021

CNN Project: Dog Breed Classifier

Project definition

Project overview

Ever since early days of computing, neural network architectures have been proposed for simple tasks, and their performances were often as good as clever designed algorithms such as Support Vector Machines, random forest, dictionary extraction, etc. For image processing though, neural networks promised high accuracy results, but it was easier to design algorithms using some sort of signal processing techniques such as Wavelet and Fourier transforms, Haar cascade filters or other types of custom designed filters depending on the problem to be solved. CNN methods were often overlooked because it was cumbersome to come up with an architecture that performed well at validation, and at the cost of spending valuable amount of time training and fine tuning it due the need of huge amounts of data and access to expensive hardware.

Since 2006 the CNN discussion started to gain momentum again thanks to the Internet, which facilitated collaboration, access to data, monetary investment and a gradual improvement in hardware for the Gaming industry. In 2012 AlexNet results on ImageNet challenge detonated the use of CNN for image classification tasks due to its impressive 10.8 percentage points reduction of the top-5 error versus the best performing algorithm at the time.

There have been a lot of new proposed CNN models for object detection and classification lately and their speeds and accuracies vary depending on the platform they are intended to be implemented.

The goal of this project is to build and train a custom model from scratch to perform dog breed classification. Since this a challenging area I am going to base my CNN on a MobileDet set of convolutions as proposed in [4]. I be using the *dogImage* dataset and the *lfw* human face dataset [8, 9] to evaluate the trained model.

Problem statement

In ML, image classification is a well-known field of research where there have been multiple approaches proposed. But it is often proved to be a difficult domain because there are few models achieving good results in general datasets. Take a dog breed classification task as an example; even humans will have a hard time distinguishing the breed of a dog, because breeds might share same characteristics. Take Fig. 1 and Fig. 2 as examples, they are different dogs but look alike. That's why for some applications it pays off to specialize a CNN model.

Metrics

When training the model, a cross entropy loss will be used as a loss function to propagate backwards the error and adjust weights, since we hope the model will output a *softmax* probability vector indicating the confidence it has of an image being this or that breed of dog. Since we are using Pytorch as our default framework, a Softmax layer is not necessary since the built-in cross entropy loss layer already computes the output probabilities.

The cross-entropy loss function is defined as:

$$l(x,y) = -\frac{1}{N}\sum_{n=1}^{N} x_n \log\big(s(y_n)\big), \qquad s(y) = \frac{e^{y}}{\sum_{c=1}^{C} e^{y_c}}$$

Where x and y are the input target and predicted output respectively, N is the minibatch size and C is the number of classes in the dataset.

We are going to use an Adam optimizer due to its adaptative strategy, which consists in computing individual learning rates for each parameter, improving model convergence speed as explained in [6, 10, 11].

Analysis

Data exploration and visualizations

The dog image dataset contains 8351 images; 6680 images for training, 835 for validation and 836 for testing, distributed in 133 dog breed classes. Images vary in size, background, image quality, resolution, illumination, pose, some of them have humans in there, dogs often are cropped, and classes are imbalanced.

Figure 1 shows a sample of 9 dog images retrieved from the test folder.

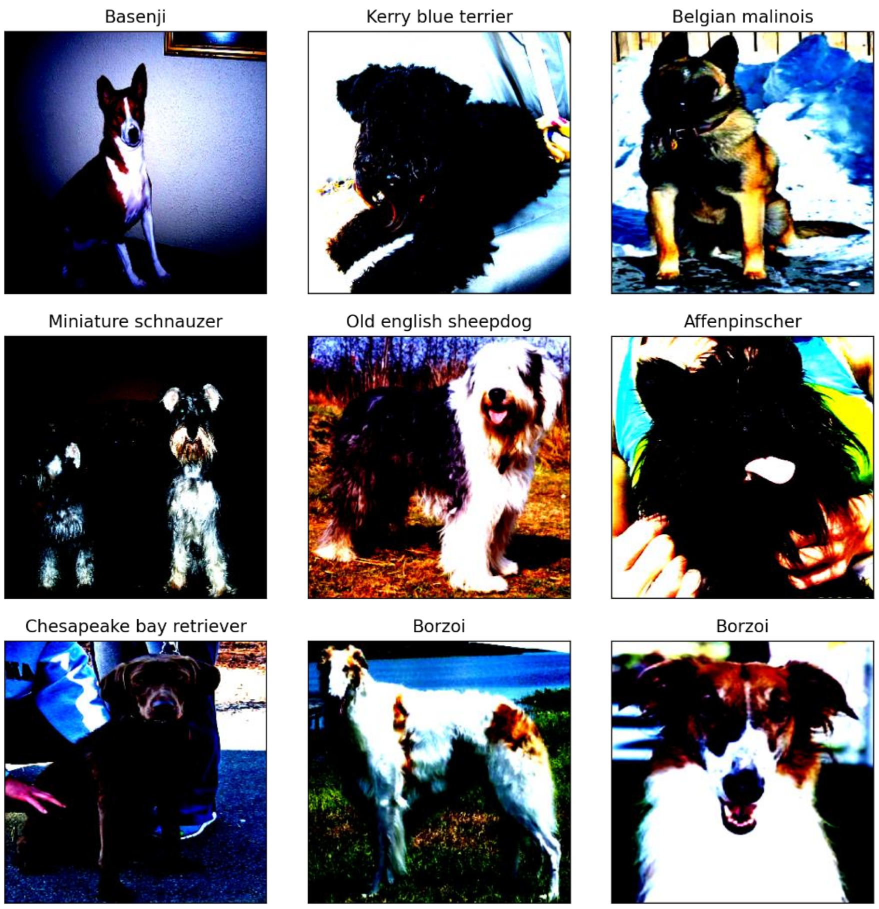


Figure 1 Dog images from test set. These images are preprocessed with resize and normalization transformations.

The human faces dataset contains 13233 human images; distributed in 5750 folders separated by name. Images have the same 250x250 size but vary in background, quality, pixel resolution, illumination, pose and are imbalanced.

Figure 2 shows a sample of 9 human face images from the dataset.

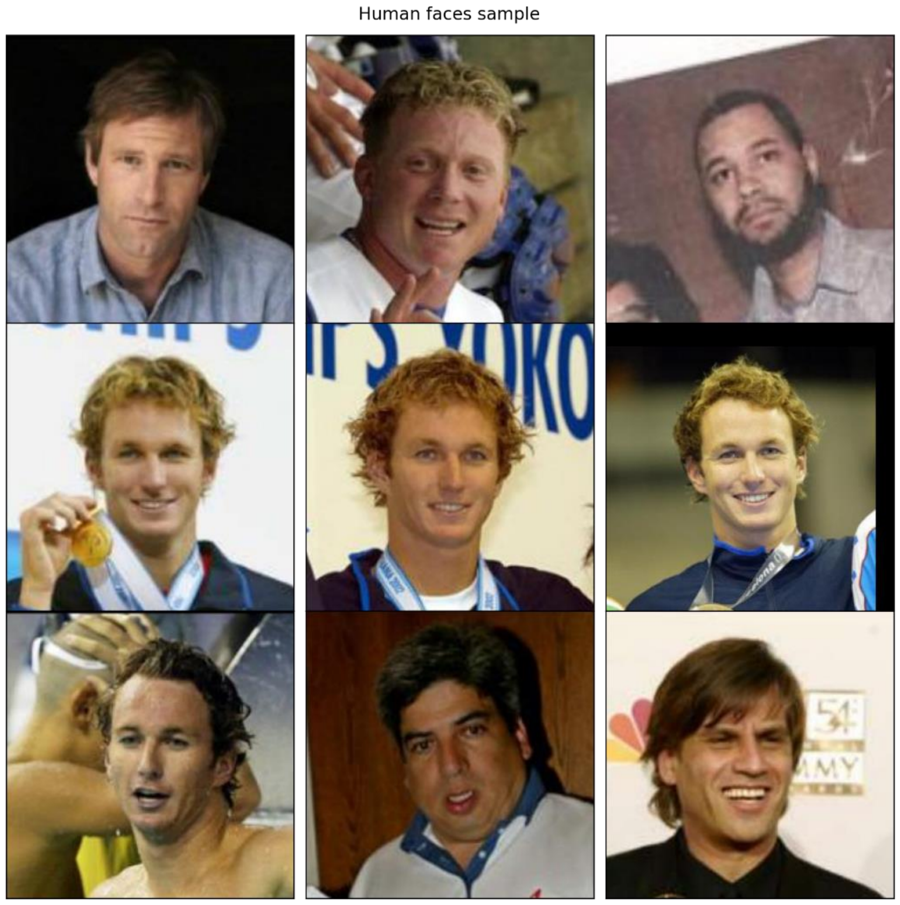


Figure 2 Human face images sample from the dataset.

Methodology

Data preprocessing

An image loader class was written to perform training, validation and testing steps. Since the class is the same for each step there is a parameter that is used only in training which is the transformation argument for data augmentation. The class accepts the batch size parameter to control how many images are used at each epoch step in training.

Thanks to the data augmentation transformation we can (as implied in the name) augment the data we have and increase the number of unique images, since each image is going to be different each time it is requested by the training algorithm.

I used two different sets of augmentation transformations; one was used when I trained the model from scratch the first time, and the second set of transformations was used in the second training step known as transfer learning, using the weights from the first training.

Here is a list of transformations used in the first training:

- *Color jitter*: this transformation changes the hue, contrast and saturation with probabilities of 0.1, 0.5 and 0.5 respectively. In this first training I'm interested and teaching the model to distinguish the same dog shape without the proper color information, hence the change in hue. Hue changes are suppressed in the second set of transformation, since I'm expecting the model to distinguish color variations as much as possible.
- *Random mirror*: this transformation applies a horizontal flip to the image with a 0.5 probability of occurrence each time.
- *Random translation*: with this transformation we teach the model how a dog looks like in different positions in the image.
- *Random resize and crop*: this transformation crops the image at random location and resize it to a specified size, which in this case is the expected model input size, which is 320x320.
- *Normalization*: this transformation normalizes the image given mean and standard deviation values recommended in Pytorch documentation for the R, G and B channels, which are [0.485, 0.456, 0.406] as mean and [0.229, 0.224, 0.225] as standard deviation. These values were calculated using millions of color images.

Figure 3 shows how the same dog image is shown to the model every step.

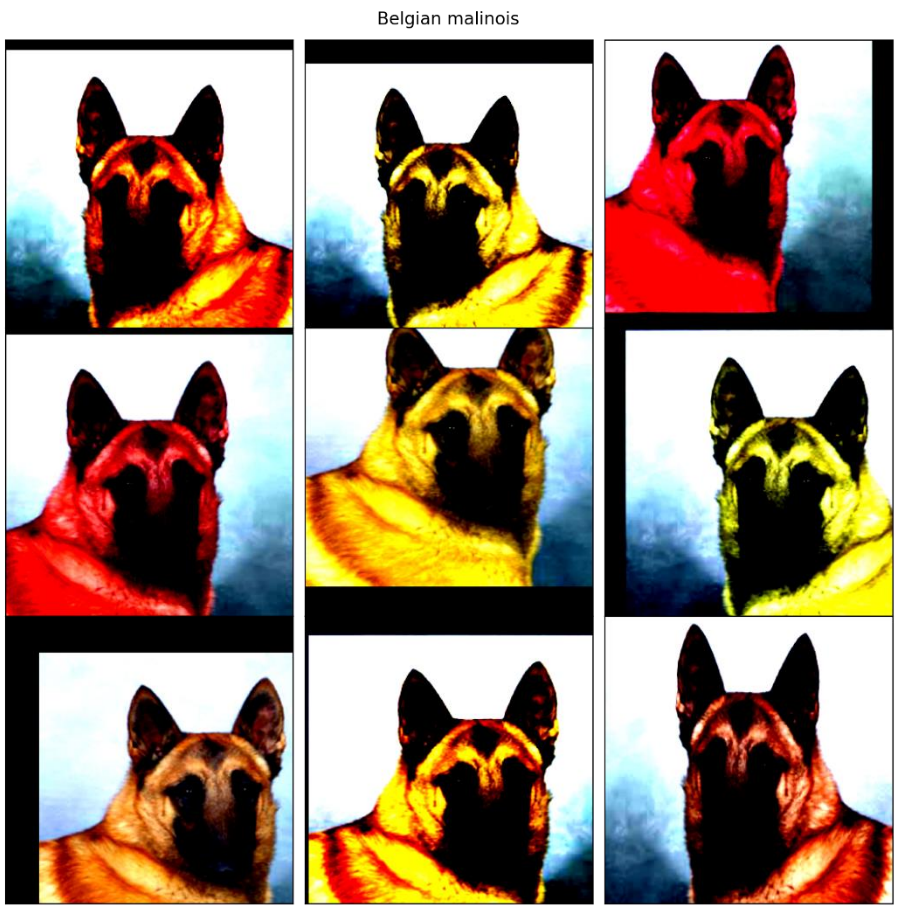


Figure 3 Nine different images of the same dog.

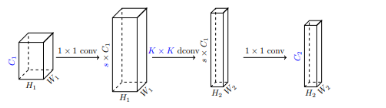
Implementation

The proposed model is a MobileDet architecture [4]. The authors use a neural architecture search (NAS) to find the best architecture configuration based on the type of target hardware the model is going to be implemented, but instead of using only inverted bottleneck (IBN) convolution layers, which are the default choice for state-of-the-art architectures due their low latency and high performance, they add two convolution layer families to the search space; Fused (expansion) and Tucker (compression) convolution layers.

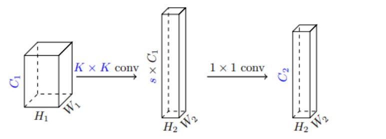
The idea behind these additions is that an IBN-only architecture, with less floating-point operations per second (FLOPS) is going to be fast and optimized in CPU but will be slower than full convolution with orders of magnitude more FLOPS in TPU or GPU, since these hardware architectures are designed to accelerate these operations.

Since my goal is to study this architecture, I have implemented the Bottleneck, Tucker and Fused layer classes in the pytorch notebook. I base myself mainly on previous knowledge and work with bottleneck architectures such as MobileNetV1 to V3, EfficientNetV1Bx [1, 2, 3], the paper [4] and the Tensorflow research repository [7].

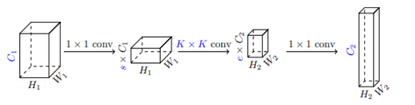
These convolutional layers are shown in Figure 4.



Inverted bottleneck layer: 1×1 pointwise convolution transforms the input channels from C_1 to $s \times C_1$ with input expansion ratio $s > 1$, then $K \times K$ *depthwise* convolution transforms the input channels from $s \times C_1$ to $s \times C_1$, and the last 1×1 pointwise convolution transforms the channels from $s \times C_1$ to C_2. The highlighted C_1, s, K, C_2 in IBN layer are searchable.



Fused inverted bottleneck layer: $K \times K$ *regular* convolution transforms the input channels from C_1 to $s \times C_1$ with input expansion ratio $s > 1$, and the last 1×1 pointwise convolution transforms the channels from $s \times C_1$ to C_2. The highlighted C_1, K, s, C_2 in the fused inverted bottleneck layer are searchable.



Tucker layer: 1×1 pointwise convolution transforms the input channels C_1 to $s \times C_1$ with input compression ratio $s < 1$, then $K \times K$ *regular* convolution transforms the input channels from $s \times C_1$ to $e \times C_2$ with output compression ratio $e < 1$, and the last 1×1 pointwise convolution transforms the channels from $e \times C_2$ to C_2. The highlighted C_1, s, K, e, C_2 in Tucker layer are searchable.

Figure 4 Bottleneck, Fused and Tucker layers explanation [4].

Bottleneck layer implemented in pytorch:

```
class Bottleneck(nn.Module):
    Params = namedtuple('BottleneckParams', 'filters kernel_size stride expansion use_residual use_se activation', defaults=(3, 1, 4, True, False, nn.ReLU6))
    def __init__(self, inputs, filters, kernel_size=3, stride=1, expansion=4, use_residual=True, use_se=False, activation=nn.ReLU6):
        super(Bottleneck, self).__init__()
        expand_filters = int(inputs) * expansion
        layers = []
        if expansion > 1:
            layers.append(RegularConv2d(inputs, expand_filters, 1, activation=activation))
        layers.append(DepthwiseConv2d(expand_filters, expand_filters, kernel_size, stride, activation))
        if use_se:
            hidden_dim = scale(expand_filters, 0.25)
            layers.append(SqueezeAndExcite(expand_filters, hidden_dim, activation))
        layers.append(RegularConv2d(expand_filters, filters, 1, activation=nn.Identity))
        self.net = nn.Sequential(*layers)
        self.use_residual = stride == 1 and inputs == filters and use_residual
    def forward(self, x):
        out = self.net(x)
        return out + x if self.use_residual else out
```

Fused layer implemented in pytorch:

```
class Fused(nn.Module):
    Params = namedtuple('FusedParams', 'filters kernel_size stride expansion use_residual use_se activation', defaults=(3, 1, 8, True, False,
nn.ReLU6))
    def __init__(self, inputs, filters, kernel_size=3, stride=1, expansion=8, use_residual=True, use_se=False, activation=nn.ReLU6):
        super(Fused, self).__init__()
        expand_filters = int(inputs) * expansion
        layers = [ RegularConv2d(inputs, expand_filters, kernel_size, stride, activation=activation) ]
        if use_se:
            hidden_dim = scale(expand_filters, 0.25)
            layers.append(SqueezeAndExcite(expand_filters, hidden_dim, activation))
        layers.append(RegularConv2d(expand_filters, filters, 1, activation=nn.Identity))
        self.net = nn.Sequential(*layers)
        self.use_residual = stride == 1 and inputs == filters and use_residual
    def forward(self, x):
        out = self.net(x)
        return out + x if self.use_residual else out
```

Tucker layer implemented in pytorch:

```
class Tucker(nn.Module):
    Params = namedtuple('TuckerParams', 'filters kernel_size stride in_ratio out_ratio use_residual activation', defaults=(3, 1, 0.25, 0.25, 
True, nn.ReLU6))
    def __init__(self, inputs, filters, kernel_size=3, stride=1, in_ratio=0.25, out_ratio=0.25, use_residual=True, activation=nn.ReLU6):
        super(Tucker, self).__init__()
        self.use_residual = stride == 1 and inputs == filters and use_residual
        hidden_in = scale(inputs, in_ratio)
        hidden_out = scale(filters, out_ratio)
        self.net = nn.Sequential(
            RegularConv2d(inputs, hidden_in, 1, activation=activation),
            RegularConv2d(hidden_in, hidden_out, kernel_size, stride, activation=activation),
            RegularConv2d(hidden_out, filters, 1, activation=nn.Identity),
        )
    def forward(self, x):
        out = self.net(x)
        return out + x if self.use_residual else out
```

I also used the proposed backbone architecture targeting the Pixel-1 CPU accelerator (Figure 5). I decided to use one of the proposed architectures since they are already the best according to the NAS algorithm.

Here is the fragment where I write the backbone settings and the Net class.

```
LAYERS = {
    'conv': RegularConv2d,
    'fused': Fused,
    'tucker': Tucker,
    'bottleneck': Bottleneck
}

SETTINGS = [
    ('conv'      , RegularConv2d.Params(16, stride=2)),
    ('bottleneck', Bottleneck.Params(8, expansion=1)),
    ('fused'     , Fused.Params(16, stride=2, expansion=4)),
    ('bottleneck', Bottleneck.Params(16)),
    ('tucker'    , Tucker.Params(16, out_ratio=0.75)),
    ('tucker'    , Tucker.Params(16, out_ratio=0.75)),
    ('bottleneck', Bottleneck.Params(32, 5, 2, 8)),
    ('bottleneck', Bottleneck.Params(32)),
    ('tucker'    , Tucker.Params(32, in_ratio=0.75, out_ratio=0.75)),
    ('tucker'    , Tucker.Params(32, in_ratio=0.75, out_ratio=0.75)),
    ('bottleneck', Bottleneck.Params(72, 5, 2, 8)),
    ('bottleneck', Bottleneck.Params(72)),
    ('bottleneck', Bottleneck.Params(72)),
    ('tucker'    , Tucker.Params(72, out_ratio=0.75)),
    ('bottleneck', Bottleneck.Params(96, kernel_size=5, expansion=8)),
    ('bottleneck', Bottleneck.Params(96, expansion=8)),
    ('bottleneck', Bottleneck.Params(96, kernel_size=5, expansion=8)),
    ('bottleneck', Bottleneck.Params(96, kernel_size=5)),
    ('bottleneck', Bottleneck.Params(104, kernel_size=5, stride=2)),
    ('tucker'    , Tucker.Params(104, 3, 1, 0.25, 0.75)),
    ('tucker'    , Tucker.Params(104, 3, 1, 0.75, 0.75)),
    ('tucker'    , Tucker.Params(104, 3, 1, 0.25, 0.75)),
    ('bottleneck', Bottleneck.Params(192, kernel_size=5, expansion=8))
]

class Net(nn.Module):
    def __init__(self, n_classes=133, settings=SETTINGS, hidden_top=256, dropout=0.25, use_se=False, depth_multiplier=1.0):
        super(Net, self).__init__()
        inputs = 3
        layers = []
        for layer, params in settings:
            if layer in ['fused', 'bottleneck']:
                params = params._replace(use_se=use_se)
            params = params._replace(filters=scale(params[0], depth_multiplier), activation=Swish6)
            layers.append(LAYERS[layer](inputs, **params._asdict()))
            inputs = params[0]
        layers.extend([
            GlobalAvgPool2d(True),
            nn.Dropout(dropout),
            nn.Linear(inputs, hidden_top),
            nn.Linear(hidden_top, n_classes)
        ])
        self.net = nn.Sequential(*layers)
    def forward(self, x):
        return self.net(x)

model = Net(hidden_top=512, use_se=True, depth_multiplier=1.5)
```

In the classification head I use a global average pool to map the 288 feature maps in the last convolution to a fully connected block with 512 hidden connections in the hidden layer and use a dropout rate of 0.25 in between. The depth multiplier of 1.5 is to expand the number of original filters used in the Bottleneck, Fused and Tucker convolution layers.

These decisions helped me to build a model that when trained from scratch, with randomly initialized weights achieves acceptable results when trained in fewer epochs.

The first time I decided to train the model for 30 epochs. I used the Adam optimizer with 0.001 as learning rate. The image loader feeds 20 images per training step augmenting the data and the validation loaders feeds 8 images per validation step. I decided to use these values due to hardware limitations.

Refinement

In the transfer learning step, where I have used the previously trained model, I decided to do the following changes to the augmentation transformation:

- Remove hue changes in the *color jitter* transform.
- Added random rotations to the images between -20^0 and 20^0
- Random gaussian blur with a kernel size of 3 and sigma x and y of 0.1 and 2 respectively with 0.5 probability

Then used a dropout rate of 0.1 to allow the network to rely on more connections and trained for 30 epochs with an early stop of 10 epochs if the validation loss doesn't improve. The learning rate of the optimizer was initialized with 0.0001 to let the model improve much slower than the first time.

For the first training and the second training I used a reduce-on-plateau learning rate scheduler to limit the maximum lambda to be used for the Adam optimizer. This scheduler doesn't do much since the Adam adapts the learning rate for each parameter accordingly.

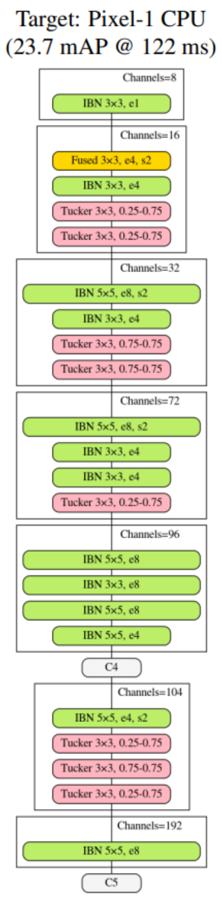


Figure 5 Proposed Pixel-1 CPU accelerator architecture.

Results

Model evaluation and validation

After the first training step the validation loss stopped improving at epoch 25 with a validation loss of 1.6539 and a validation accuracy of 0.5599. The test loss and test accuracy were 1.9752 and 0.48 respectively, classifying 409 out of 836 images correctly.

In the second training step, with the changes made in the augmentation transform, the validation loss stopped improving at epoch 3 with validation loss of 1.3228 and validation accuracy of 0.6206. The test loss and test accuracy were 1.3385 and 0.62 respectively, classifying 523 out of 836 images correctly.

The following table summarizes the results.

Training step	*Epochs*	*Test loss*	*Test accuracy*
From scratch	25	1.9752	48% (409/836)
Transfer learning	3	1.3385	62% (523/836)

Justification

References

1. A. G. Howard, W. Wang et al., *"MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications"*, https://arxiv.org/pdf/1704.04861.pdf
2. M. Sandler, A. Howard et al., *"MobileNetV2: Inverted Residual and Linear Bottlenecks"*, https://arxiv.org/pdf/1801.04381.pdf
3. A. Howard, M. Sandler et al., *"Searching for MobileNetV3"*, https://arxiv.org/pdf/1905.02244.pdf

4. Y. Xiong, H. Liu et al., *"MobileDets: Searching for Object Detection Architectures for Mobile Accelerators"*, https://arxiv.org/pdf/2004.14525.pdf
5. *Udacity's dog breed classifier repository*, https://github.com/udacity/deep-learning-v2-pytorch/tree/master/project-dog-classification
6. *Pytorch documentation*, https://pytorch.org/docs/stable/index.html
7. *Tensorflow models research repository*, https://github.com/tensorflow/models
8. *Dog images dataset*, https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/dogImages.zip
9. *Human face dataset*, https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/lfw.zip
10. D. P. Kingman and J. L. Ba, "*Adam: A Method for Stochastic Optimization*", https://arxiv.org/pdf/1412.6980.pdf
11. *Adam – latest trends in deep learning optimization*, https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c