Machine Learning Engineer Nanodegree

Capstone Project

Noel Mathew
October 6th, 2020

Dog Breed Classification

I. Definition

Project Overview

Computer vision is a field of artificial intelligence that trains computers to interpret and understand the visual world. The idea of building intelligent machines that can understand and interpret visual world existed from 1960s. It all started with Hubel and Wiesel's experiment to understand how a cat's neuron responded to visual stimuli. MIT started a Summer Vision Project in the summer of 1966, with a goal to solve the vision problem in a single summer. Though it was not a success, it led to people getting interested and further pushed the field. Today we have not solved the problem yet, but we have surely come far. Far enough be used with medical imaging, object detection, image captioning and so on. The field was revolutionized with the introduction of Deep Learning and the hardware support provided by Nvidia GPUs and was demonstrated by Alexnet winning the Imagenet Competition in 2012.

My whole journey with Machine Learning started with being amused by what deep learning has made possible in the fields of Computer Vision. I will be using Deep learning to estimate dog breeds which is not a trivial problem to solve due to the interclass variance and the intraclass variance between the dog breeds.

Problem Statement

The goal of the project is to determine which dog breed a given image contains. I will use Convolutional Neural Network (CNN) to classify the images by dog breeds. If the image contains a dog, the output will be the identified dog breed. If the image contains a human then output will the resembling dog breed. If the image contains neither of the two then it will report an error. I will be using a custom loss function, a combination of two Binary Cross Entropy loss `dog_loss` & `face_loss` and a Cross Entropy loss for `breed_loss`. The breed_loss will be conditional; i.e., the breed_loss will only be calculated in case the image is actually a dog.

Furthermore, One Cycle Scheduling with cosine annealing for scheduling learning rates and momentums will be used.

Metrics

The project demands a minimum of 10% accuracy on dog breeds for a CNN Model trained from scratch, and a minimum of 60% accuracy on dog breeds for a CNN Model trained with transfer learning. We also need to detect human faces and dogs as stated in the problem statement.

For this purpose, I will be using the following metrics:

1. **Dog Breed Accuracy**: The accuracy for the dog breed prediction.
2. **Face Accuracy**: Accuracy for detecting faces.
3. **Dog Accuracy**: Accuracy for detecting dogs.
4. **Breed F1 Score**: The f1 score for dog breed prediction.

We have a slight imbalance in the dataset. This imbalance may lead to misleading accuracy scores as the accuracy does not account for the minor classes in the dataset. I will be using **weight-average f1 score** along with accuracy as a second metric to keep us from interpreting a misleading accuracy score wrongly.

In **weighted-average f1 score**, we use number of samples from each class as weight while calculating the f1 score. This makes sure that we don't get an inflated metric value due to the imbalance in the dataset.

II. Analysis

Data Exploration

The datasets are provided by Udacity.

Udacity has provided us with two sets:

1. **Dog dataset**: The dataset contains 133 folders; each corresponding to a different dog breed. The dataset is split into train, test and valid folders. There 8351 total dog images.

    - **Train dataset**:
      We have 6680 images in the training dataset. We have an average of 50 images with a minimum of 26 images a breed and a maximum 77 images a breed.
    - **Test dataset**:
      We have 836 images in the training dataset. We have an average of 6 images with a minimum of 3 images a breed and a maximum 10 images a breed.
    - **Valid dataset**:
      We have 835 images in the training dataset. We have an average of 6 images with a minimum of 4 images a breed and a maximum 9 images a breed.
    - We have enough images to not have any class imbalance problem with transfer learning.
    - Each image is an RGB image.
    - The dataset is arranged as follows:

```
train/
    breed-1/
        breed_name_number.jpg
        .
        .
    breed-2/
    breed-3/
valid/
test/
```

2. **Human face dataset**: The dataset contains folders corresponding to different people containing photos. There are 13233 total human images. The human images are arranged by names of the people. I won't be using the names of the people respecting their privacy. We will be only using the images for training the face detection classifiers. The images are RGB.

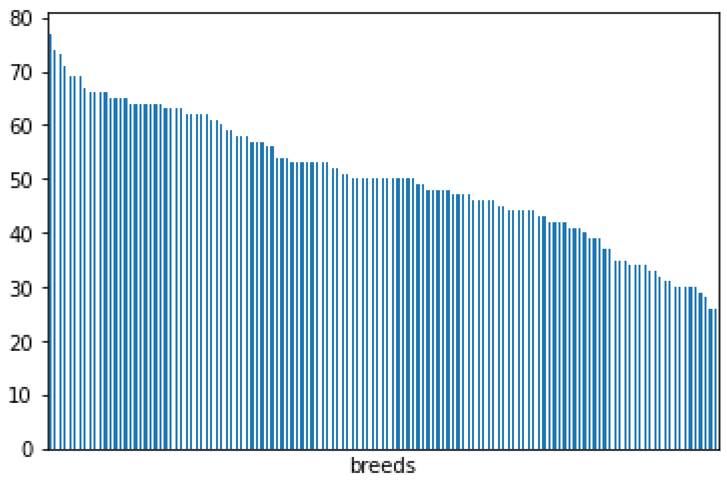
   The dataset has photos for 5749 people, arrange by their names into separate folders. I will be using the first 3749 folders with 8926 images for the train set, next 1000 folders with 2014 images for validation set and the remaining 1000 folders with 2293 images for the test set. We cannot split this dataset randomly as all the photos in the same folder are of the same person, it might lead to some leakage.

The dog dataset will be used to detect dogs and predict the dog breeds. The human face dataset will be used to detect faces.

Exploratory Visualization

- **Dog dataset**:
  The images are named starting with the dog breed. I will be extracting labels from the image names.
  We can see a small subset of the dataset below

We can also look at the breed distribution.



We can see the data is not perfectly balanced but not significantly imbalanced either.

- **Human dataset**:

  We can see a small subset of the dataset below:

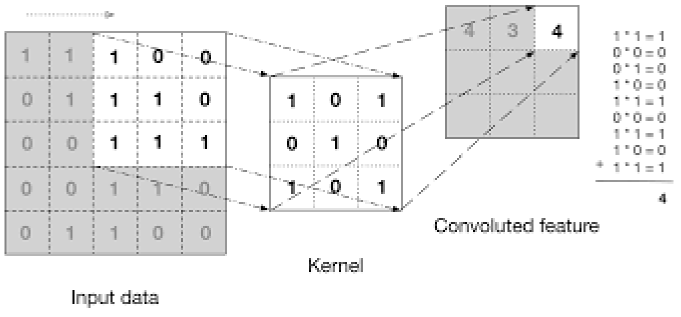
Algorithms and Techniques

This project uses Convolutional Neural Network(CNN) for extracting features from the images. Let us understand what a CNN is and how it is significant when it comes to our task.

Convolutional Neural Network:

A convolutional neural network is a neural network created by stacking layers of operation called convolution.

Convolution :
A convolution consists of a kernel; also called a filter or a convolutional kernel. The kernel is a matrix of size 3x3 or 5x5 or 7x7. This kernel is slid across the input image in steps called strides. A **stride** is the number of steps a convolutional kernel moves horizontally or vertically. At each step a matrix multiplication of the kernel with the input region the kernel is over; called a **receptive field**, is calculated. A basic convolution with a 3x3 kernel is shown below:



The output of a convolution is called a **Feature map**. We can see in the image above that the feature map is smaller than the input image. To avoid this, we pad the input on all sides. There are a few approaches generally used for padding; zero padding, reflective padding, symmetric padding. The important thing to get right with padding is the padding size.

We can adjust the size of the **feature map** using this formula:

$$\text{output}_w = \left\lfloor \frac{\text{image}_w + 2\text{pad} - \text{kernel}_{\text{size}}}{\text{stride}} \right\rceil + 1$$

$$\text{output}_h = \left\lfloor \frac{\text{image}_h + 2\text{pad} - \text{kernel}_{\text{size}}}{\text{stride}} \right\rceil + 1$$

ReLU:
In a neural network, an activation function is responsible to transform the output of the linear calculations into an activation of the node. This is responsible for when a part of the network will be activated.

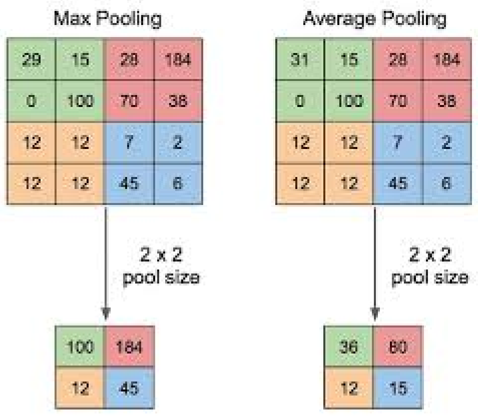
The ReLU (rectified linear unit), is a piecewise linear function. This means the functions acts linearly for some part of the input and non linearly for the other part.

The ReLU can be defined with the formula:

```
acts = max(0, input)
```

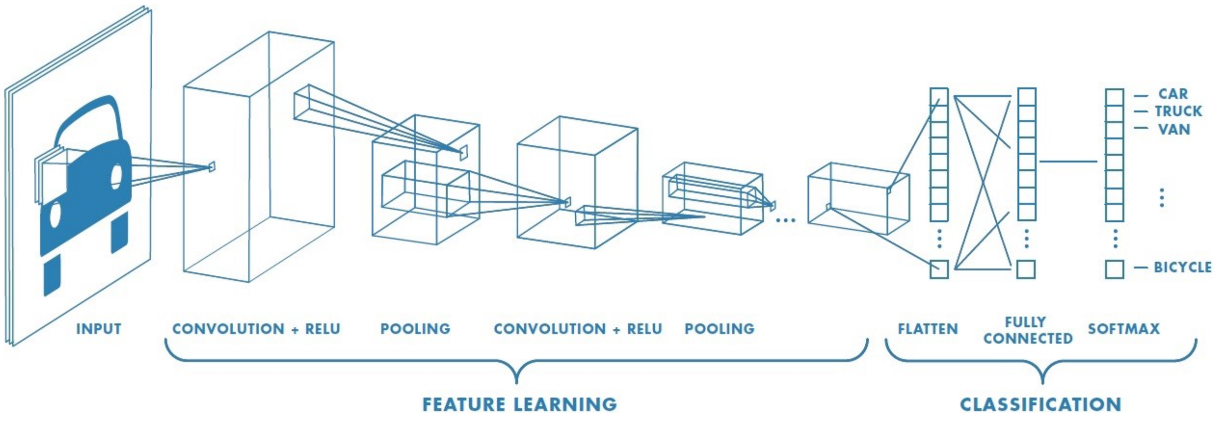
Pooling:
Pooling is a operation used to reduce the dimensionality of the feature map. There are two common types of pooling, average pooling and max pooling. The pooling operation applied the pooling function we choose on the feature map using a kernel size similarly to the convolutional layer.

We can see a pooling applied to a feature map in the image below:



Nowadays, a stride 2 convolution is used to replace the pooling operation in a CNN. But a pooling operation is useful when we want reduce the number of parameters in the model.

In practice, kernels are stacked in order to extract high-dimensional representations of the input. We can see an example in the image below:



Significance of CNNs:

1. We can see that the spatial relationship between the input features is preserved in the feature map. This means that the location of an object in a particular location of the input

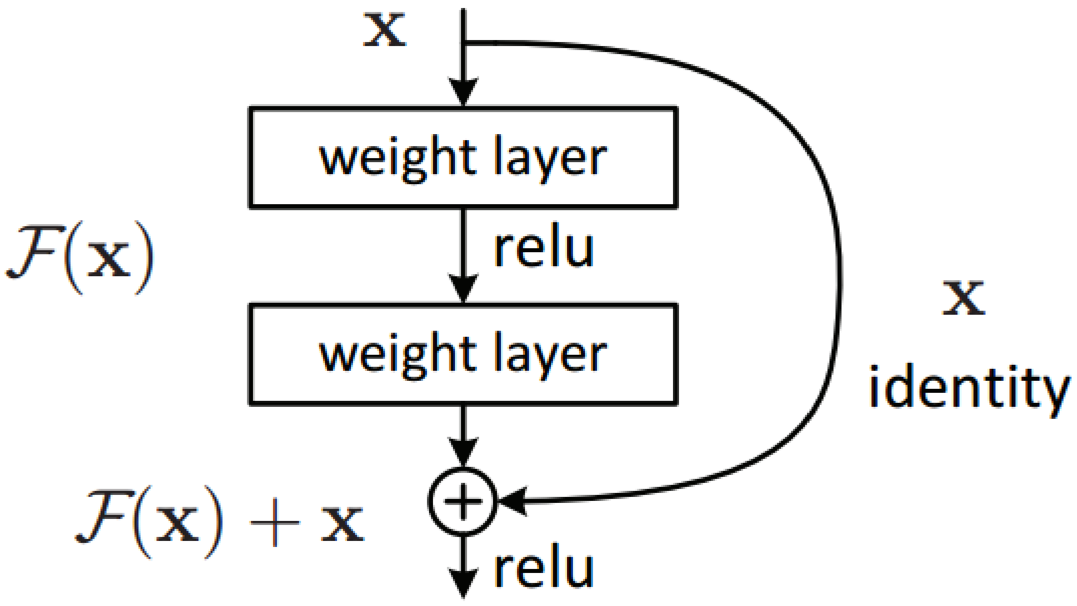
image is preserved in the feature map generated by a convolutional layer.

2. Since the kernel in a convolution looks at a region in the image of the size of its receptive field, it is able to learn features in localized regions like edges, color gradients, brightness changes, etc.
3. When convolutional layers are stacked on top of each other, the kernels in the deeper layers have a larger effective receptive fields. So the deeper layers can learn more complex features.

Architecture:

I have used a custom CNN architecture, trained from scratch on the provided datasets. The key highlight to the architecture I have used is Residual Blocks and Dense Blocks. These are simple to understand techniques that have pushed the field of deep learning further ahead into deeper layers.

In a Residual Block, the input to the layer is passed through two convolutional layers(2D Convolution > BatchNorm 2D > ReLU) and the activations are summed with the input thus added a skip connection. This means that the model can learn to ignore the convolutions and use the inputs if it finds necessary.


x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Similarly, in a Dense Block, the same logic is followed but instead of adding the inputs to the activations, the inputs are concatenated to the activation. This leads to better result than Residual Block but is more computationally expensive.

The architecture for the project is a single model which does two tasks.

1. Dog vs human classification:
   This part of the model is used for dog and human detection.

2. Breed classification:
   This part of the model is used for breed classification.

Optimizer:

I used Adam optimizer for training the model. The Adam optimizer helps to speed up the training of the model.

Scheduler:

Something that is often overlooked while training a deep learning model is a scheduler. Scheduling the hyperparameters of the model is a very important task to achieve the best results. I have used pytorch's implementation of OneCycleLR from torch.optim.lr_scheduler for this purpose. This scheduler was described in the paper Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates.. It is used to schedule the learning rate and momentum.

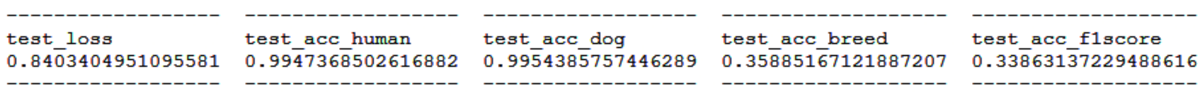
Loss:

The model is trained with a custom loss function. The loss function is a combination of the dog and human detection loss and the breed loss.

Benchmark

For defining a benchmark for the task, I trained a custom CNN model from scratch. The architecture uses ResBlocks and Dense Blocks to improve the model complexity. The model was trained with Adam optimizer and One Cycle training Policy described above, with a maximum learning rate of 0.1 for 20 epochs.

The results obtained with the benchmark are quite promising. I was able to achieve an accuracy of 35% and f1score of 0.3386 on the test set.



```
------------------  ------------------  ------------------  -------------------  -------------------
test_loss           test_acc_human      test_acc_dog        test_acc_breed       test_acc_f1score
0.8403404951095581  0.9947368502616882  0.9954385757446289  0.35885167121887207  0.33863137229488616
------------------  ------------------  ------------------  -------------------  -------------------
```

III. Methodology

Data Preprocessing

Data Preprocessing is used to get the data ready to train the model.

Image Library:
For this project, I used PIL(Python Imaging Library) to load the image files and convert them to RGB. Converting to RGB is an important step that helps to make sure that even grayscale images do not cause shape issues.

Labeller:
Once the images files are read, we need to process the image labels. For processing the image labels, I have created Labeller class.

```
class Labeller:
    def __init__(self, label_func):
        self.label_dict = None
        self.label_func = label_func
        self.label_lookup = None

    def get_labels(self, files):
        if self.label_dict is not None:
            return self
        labels = set()
        for file in files:
            labels.add(self.label_func(file))
        self.label_dict = {label: i for i, label in enumerate(labels)}
        self.label_lookup = {v: k for k, v in self.label_dict.items()}
        return self

    def get_label(self, img_path):
        lbl_class = self.label_func(img_path)
        return self.label_dict[lbl_class], lbl_class
```

This class accepts a labelling function. The labeller has a get_labels functions which accepts the files in the dataset and creates a label dictionary (`string label` to `integer label`) and a reverse lookup.

I use this Labeller class with different labelling functions for breed labels and dog vs human labels.

1. Breed Labelling function:

```
def get_breed_label(path):
    return ' '.join(path.name.split('_')[:-1])
```

2. Dog vs human labelling function:

```
def human_or_dog(path):

    return 'human' if 'lfw' in str(path).split('/') else 'dog'
```

Data Augmentations:
When working with Deep Learning, the more data we can get, the better the model can generalize. But collecting data is expensive. So to get the most out of the available data, we augment the data to look different enough that it helps our model to generalize better. Data Augmentation can sometimes be thought as data for free.

For this task, I am using the following data augmentations:

1. **Presizing**:
   Presizing is more of a preprocessing for data augmentation than the latter. Presizing is what the name says it is. We resize the data to a significantly bigger size than the final size we want.
   For example, if we want our images to be 224x224, we may presize the images at 480x480. This helps in not losing out huge chunks of our images when other transforms like cropping and prespective transforms are applied.

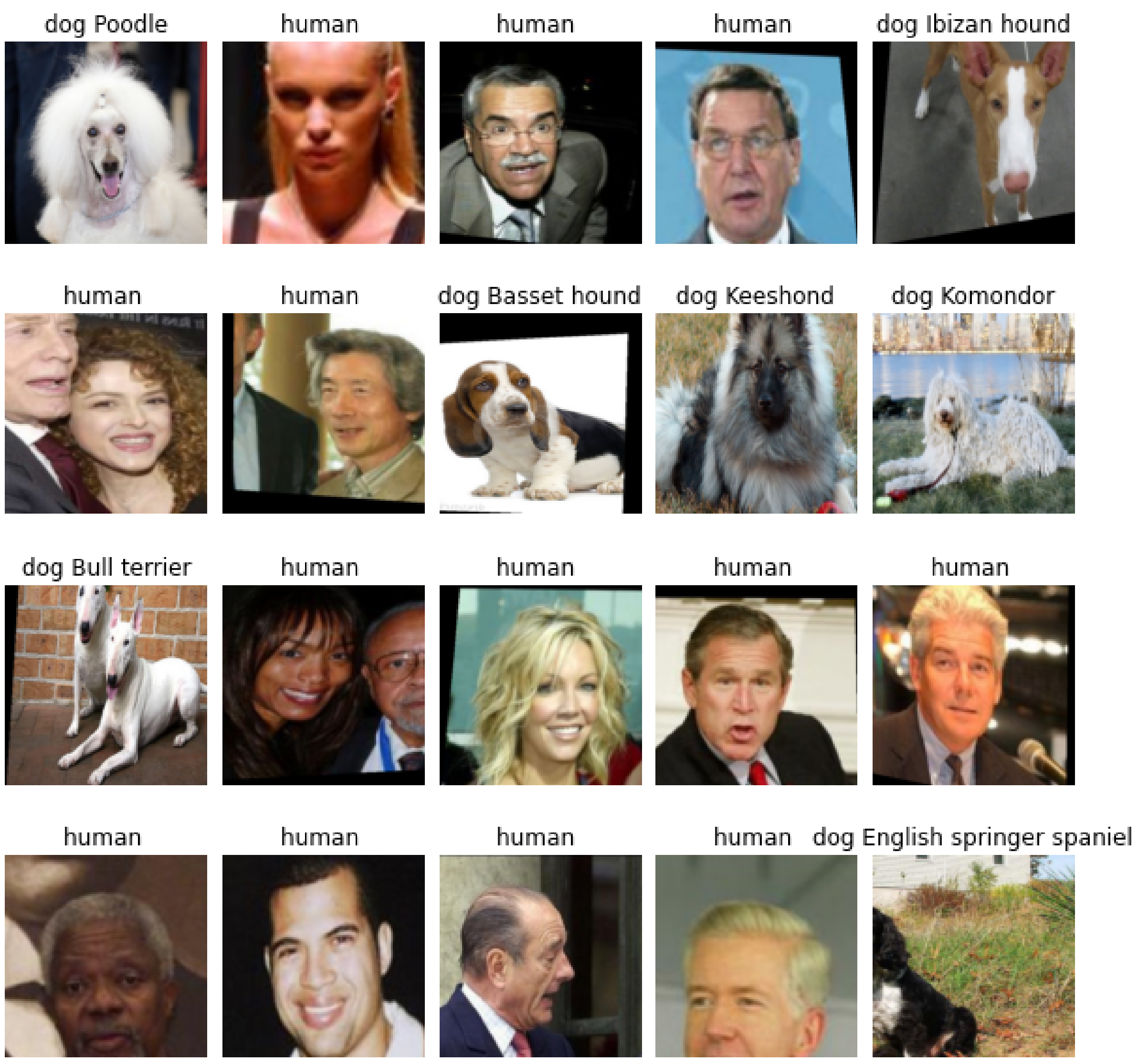
2. **RandomPerspective**:
   Perspective transforms distorts the image such that it looks like the image is captured from a different perspective.

3. **RandomResizedCrop**:
   Crops the image to a random size and aspect ratio.

4. **RandomHorizontalFlip**:
   Randomly flips the images horizontally given a p value.

5. **Normalize**: Normalizes the image to imagenet stat as we are using a model pre-trained on imagenet.

We can see the result of using augmentation on one batch below.



DataLoaders:

To create dataloaders, we need to create Dataset class which does two things:

1. returns the length of the dataset
2. returns a tuple of independent and dependent values given an index.

This is the basic requirement for a class to be considered a dataset. We can see the implementation of Dataset class below

```
class Dataset:

    def __init__(self, path_dogs, human_paths, folder, breed_labeller,
                 dog_human_labeller, tfms=None, stats=None, size=224):
        self.files = utils.get_files(path_dogs/folder)
        self.breed_labeller = breed_labeller.get_labels(self.files)
        self.human_files = []
        [self.human_files.extend(utils._get_files(path))
         for path in human_paths]
        self.files += self.human_files
        self.dog_human_labeller =
dog_human_labeller.get_labels(self.files)
        self.size = size
        if tfms is None:
            self.tfms = [
                transforms.Resize(size=(self.size, self.size)),
                transforms.ToTensor()
            ]
        else:
            self.tfms = tfms
        if stats is not None:
            self.tfms.append(transforms.Normalize(**stats))
        self.tfms = transforms.Compose(self.tfms)

    def __len__(self):
        return len(self.files)

    def __getitem__(self, idx):
        img_path = self.files[idx]
        img = Image.open(img_path)
        dog_human_label, lbl_str =
self.dog_human_labeller.get_label(img_path)
        breed_label = -1
        if lbl_str == 'dog':
            breed_label, _ = self.breed_labeller.get_label(img_path)
        img = self.tfms(img)
        dog_human_target = torch.zeros(2)
        dog_human_target[dog_human_label] = 1
        breed_target = torch.tensor(breed_label, dtype=torch.long)
        return img, dog_human_target, breed_target
```

In the above code, we do the following:

1. On initialization, we load all the files using a utility function and we generate the label mappings.
2. We calculate the required transforms and save them as a torchvision.transforms.Compose object.
3. One labelling trick we use here is we set the breed_label to -1 for all non dog images. This will be used while calculating the loss.

Implementation

Architecture:

The final architecture used is a single model which does two tasks as mentioned above. We have 133 breeds of dogs and we want to detect dogs and human faces. So the architecture is designed to have 135 output activations. The first two activations are used for detecting dogs and human faces and the remaining 133 activations are used for dog breed classification.

```
class ModelTransfer(nn.Module):

    def __init__(self, pretrained=True):
        super().__init__()
        self.model = models.resnet34(pretrained=pretrained)
        self.model.fc = nn.Linear(512, 300)
        self.head = nn.Sequential(nn.BatchNorm1d(300),
                                  nn.ReLU(),
                                  nn.Dropout(p=0.2),
                                  nn.Linear(300, 135),
                                  nn.BatchNorm1d(135))

    def forward(self, x):
        x = self.head(self.model(x))
        return x
```

Loss Function:

We use Binary Cross Entropy Loss with each of the dog and human activations and a Cross Entropy Loss for Breed Classification.
The loss functions stores the index of the dog and human activation to calculate the loss.
The complete loss is:

> `final_loss = dog_loss + face_loss + is_dog*breed_loss`

The code for the loss function can be scene below:

```
class CustomLoss(nn.Module):

    def __init__(self, dog_human_labeller):
        super().__init__()
        self.dog_loss = nn.BCEWithLogitsLoss()
        self.human_loss = nn.BCEWithLogitsLoss()
        self.breed_loss = nn.CrossEntropyLoss()
        self.dog_human_labeller = dog_human_labeller
        self.human_idx = self.dog_human_labeller.label_dict['human']
        self.dog_idx = self.dog_human_labeller.label_dict['dog']
    def forward(self, outputs, t1, t2):
        h_loss = self.human_loss(outputs[:, [self.human_idx]],
                                 t1[:, [self.human_idx]])
        d_loss = self.dog_loss(outputs[:, [self.dog_idx]],
                               t1[:, [self.dog_idx]])
        # non dog images have a breed label -1.
        # mask to get only dogs.
        mask = t2 >= 0
        b_loss = self.breed_loss(outputs[mask, 2:], t2[mask])
        b_loss = 0. if b_loss.isnan() else b_loss
        loss = h_loss + d_loss + b_loss
        return loss
```

Transfer learning method used for the final model:

1. A Resnet34 model, pre-trained on ImageNet is used as a base model.
2. The final fully connected layer (Linear Layer) is replaced with a new Linear Layer with 300 activations.
3. A new head is added to the model which outputs 135 activations through a BatchNorm1d Layer.
4. The model is trained using a custom training and evaluation loop with the above stated CustomLoss.
5. A Recorder class is used to track the metrics and the loss after each epoch.

Recorder:

I have created a Recorder function to track the loss and the other metrics for each epoch.

```
class Recorder:
    def __init__(self):
        self.train_acc_human, self.train_acc_dog = [], []
        self.train_acc_breed, self.train_loss = [], []
        self.valid_acc_human, self.valid_acc_dog = [], []
        self.valid_acc_breed, self.valid_loss = [], []

    def update(self, train_metrics, valid_metrics):
        self.train_loss.append(train_metrics['loss'])
        self.train_acc_human.append(train_metrics['accuracy_human'])
        self.train_acc_dog.append(train_metrics['accuracy_dog'])
        self.train_acc_breed.append(train_metrics['accuracy_breed'])
        self.valid_loss.append(valid_metrics['loss'])
        self.valid_acc_human.append(valid_metrics['accuracy_human'])
        self.valid_acc_dog.append(valid_metrics['accuracy_dog'])
        self.valid_acc_breed.append(valid_metrics['accuracy_breed'])
```

Refinement

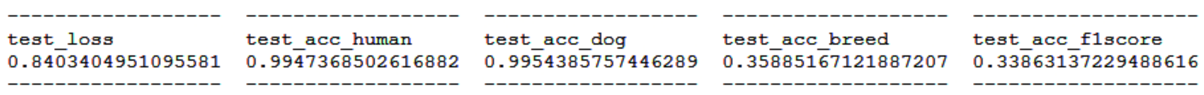
I trained two models for this project.

1. Custom CNN Architecture trained from scratch:
2. Transfer learning with resnet34 model and data augmentations.

1. Custom CNN Architecture:

- This model is trained from scratch with the above stated data augmentations. All the above mentioned techniques have been used; one cycle policy and Adam optimizer.
- We can see that the model has 135 output activations as stated before.

The performance of the model on test set can be seen below:

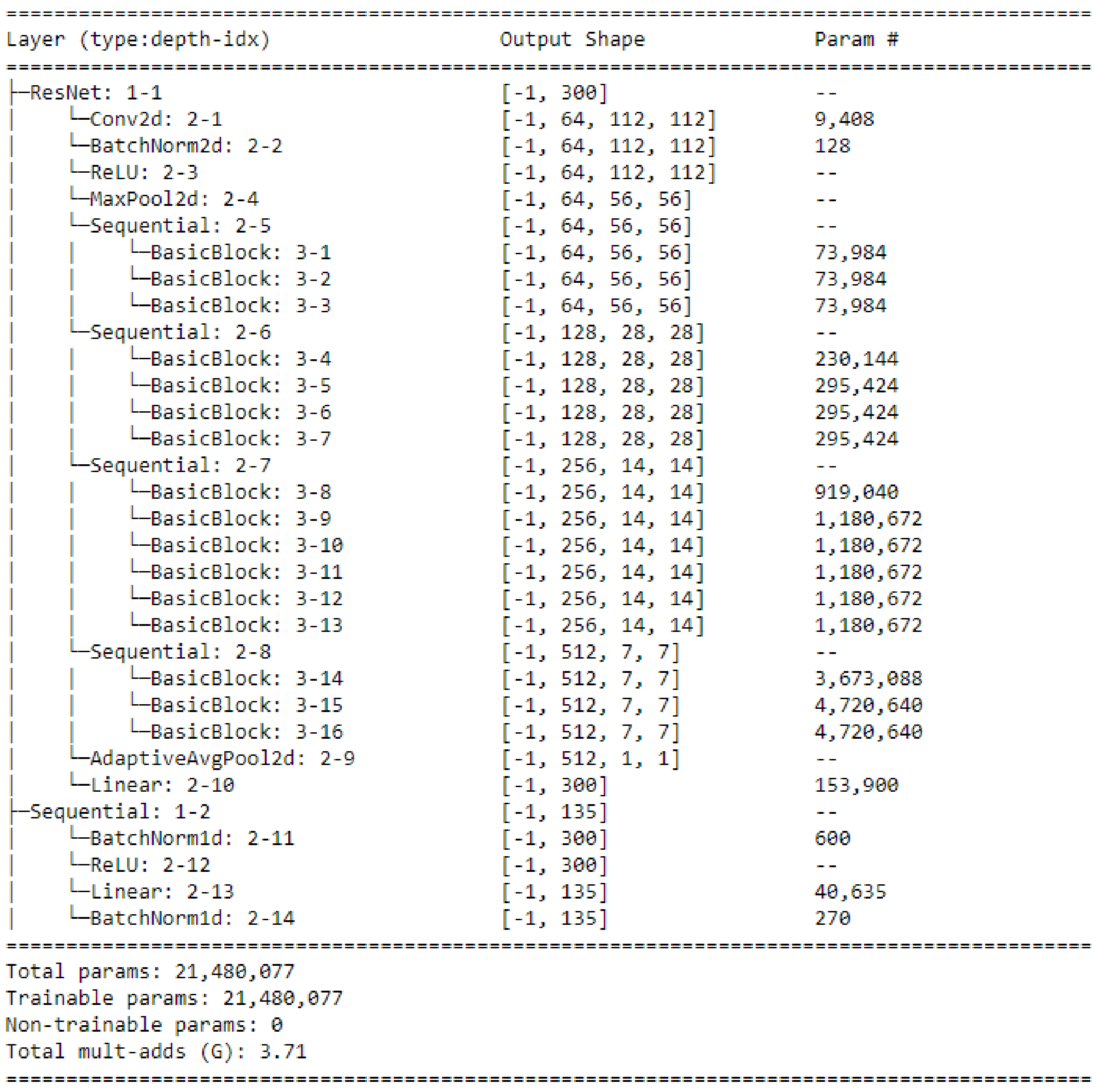


```
------------------  ------------------  ------------------  -------------------  -------------------
test_loss           test_acc_human      test_acc_dog        test_acc_breed       test_acc_f1score
0.8403404951095581  0.9947368502616882  0.9954385757446289  0.35885167121887207  0.33863137229488616
------------------  ------------------  ------------------  -------------------  -------------------
```

We get an accuracy of 35% and f1score of 0.3386 as a benchmark which is quite good considering the complexity of the task.

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Sequential: 1-1                        [-1, 135]                 --
|    └─Sequential: 2-1                   [-1, 16, 112, 112]        --
|    |    └─Conv2d: 3-1                  [-1, 16, 112, 112]        432
|    |    └─BatchNorm2d: 3-2             [-1, 16, 112, 112]        32
|    |    └─ReLU: 3-3                    [-1, 16, 112, 112]        --
|    └─Dropout2d: 2-2                    [-1, 16, 112, 112]        --
|    └─DenseBlock: 2-3                   [-1, 24, 112, 112]        --
|    |    └─Sequential: 3-4              [-1, 8, 112, 112]         1,168
|    |    └─Sequential: 3-5              [-1, 8, 112, 112]         592
|    └─Dropout2d: 2-4                    [-1, 24, 112, 112]        --
|    └─Sequential: 2-5                   [-1, 64, 56, 56]          --
|    |    └─Conv2d: 3-6                  [-1, 64, 56, 56]          13,824
|    |    └─BatchNorm2d: 3-7             [-1, 64, 56, 56]          128
|    |    └─ReLU: 3-8                    [-1, 64, 56, 56]          --
|    └─Dropout2d: 2-6                    [-1, 64, 56, 56]          --
|    └─DenseBlock: 2-7                   [-1, 128, 56, 56]         --
|    |    └─Sequential: 3-9              [-1, 64, 56, 56]          36,992
|    |    └─Sequential: 3-10             [-1, 64, 56, 56]          36,992
|    └─Dropout2d: 2-8                    [-1, 128, 56, 56]         --
|    └─Sequential: 2-9                   [-1, 256, 28, 28]         --
|    |    └─Conv2d: 3-11                 [-1, 256, 28, 28]         294,912
|    |    └─BatchNorm2d: 3-12            [-1, 256, 28, 28]         512
|    |    └─ReLU: 3-13                   [-1, 256, 28, 28]         --
|    └─Dropout2d: 2-10                   [-1, 256, 28, 28]         --
|    └─ResBlock: 2-11                    [-1, 256, 28, 28]         --
|    |    └─Sequential: 3-14             [-1, 256, 28, 28]         590,336
|    |    └─Sequential: 3-15             [-1, 256, 28, 28]         590,336
|    └─Dropout2d: 2-12                   [-1, 256, 28, 28]         --
|    └─Sequential: 2-13                  [-1, 512, 14, 14]         --
|    |    └─Conv2d: 3-16                 [-1, 512, 14, 14]         1,179,648
|    |    └─BatchNorm2d: 3-17            [-1, 512, 14, 14]         1,024
|    |    └─ReLU: 3-18                   [-1, 512, 14, 14]         --
|    └─AdaptivePooling: 2-14             [-1, 512, 1, 1]           --
|    |    └─AdaptiveMaxPool2d: 3-19      [-1, 512, 1, 1]           --
|    |    └─AdaptiveAvgPool2d: 3-20      [-1, 512, 1, 1]           --
|    └─Lambda: 2-15                      [-1, 512]                 --
|    └─Linear: 2-16                      [-1, 300]                 153,900
|    └─BatchNorm1d: 2-17                 [-1, 300]                 600
|    └─ReLU: 2-18                        [-1, 300]                 --
|    └─Dropout: 2-19                     [-1, 300]                 --
|    └─Linear: 2-20                      [-1, 135]                 40,635
|    └─BatchNorm1d: 2-21                 [-1, 135]                 270
==========================================================================================
Total params: 2,942,333
Trainable params: 2,942,333
Non-trainable params: 0
Total mult-adds (G): 1.70
==========================================================================================
Input size (MB): 0.57
Forward/backward pass size (MB): 26.04
Params size (MB): 11.22
Estimated Total Size (MB): 37.84
==========================================================================================
```

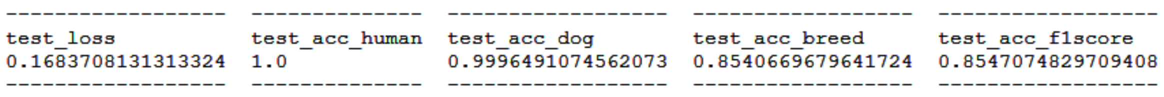
2. Transfer learning with Resnet34:



```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─ResNet: 1-1                            [-1, 300]                 --
|    └─Conv2d: 2-1                       [-1, 64, 112, 112]        9,408
|    └─BatchNorm2d: 2-2                  [-1, 64, 112, 112]        128
|    └─ReLU: 2-3                         [-1, 64, 112, 112]        --
|    └─MaxPool2d: 2-4                    [-1, 64, 56, 56]          --
|    └─Sequential: 2-5                   [-1, 64, 56, 56]          --
|    |    └─BasicBlock: 3-1              [-1, 64, 56, 56]          73,984
|    |    └─BasicBlock: 3-2              [-1, 64, 56, 56]          73,984
|    |    └─BasicBlock: 3-3              [-1, 64, 56, 56]          73,984
|    └─Sequential: 2-6                   [-1, 128, 28, 28]         --
|    |    └─BasicBlock: 3-4              [-1, 128, 28, 28]         230,144
|    |    └─BasicBlock: 3-5              [-1, 128, 28, 28]         295,424
|    |    └─BasicBlock: 3-6              [-1, 128, 28, 28]         295,424
|    |    └─BasicBlock: 3-7              [-1, 128, 28, 28]         295,424
|    └─Sequential: 2-7                   [-1, 256, 14, 14]         --
|    |    └─BasicBlock: 3-8              [-1, 256, 14, 14]         919,040
|    |    └─BasicBlock: 3-9              [-1, 256, 14, 14]         1,180,672
|    |    └─BasicBlock: 3-10             [-1, 256, 14, 14]         1,180,672
|    |    └─BasicBlock: 3-11             [-1, 256, 14, 14]         1,180,672
|    |    └─BasicBlock: 3-12             [-1, 256, 14, 14]         1,180,672
|    |    └─BasicBlock: 3-13             [-1, 256, 14, 14]         1,180,672
|    └─Sequential: 2-8                   [-1, 512, 7, 7]           --
|    |    └─BasicBlock: 3-14             [-1, 512, 7, 7]           3,673,088
|    |    └─BasicBlock: 3-15             [-1, 512, 7, 7]           4,720,640
|    |    └─BasicBlock: 3-16             [-1, 512, 7, 7]           4,720,640
|    └─AdaptiveAvgPool2d: 2-9            [-1, 512, 1, 1]           --
|    └─Linear: 2-10                      [-1, 300]                 153,900
├─Sequential: 1-2                        [-1, 135]                 --
|    └─BatchNorm1d: 2-11                 [-1, 300]                 600
|    └─ReLU: 2-12                        [-1, 300]                 --
|    └─Linear: 2-13                      [-1, 135]                 40,635
|    └─BatchNorm1d: 2-14                 [-1, 135]                 270
==========================================================================================
Total params: 21,480,077
Trainable params: 21,480,077
Non-trainable params: 0
Total mult-adds (G): 3.71
==========================================================================================
```

- This model is trained using transfer learning. The first part of the model is frozen and only the final layers are trained.
- OneCycle policy and Adam optimizer are used for training.
- We can see that the model has 135 output activations as stated before.

The performance of the model on test set can be seen below:



```
------------------  --------------  ------------------  ------------------  ------------------
test_loss           test_acc_human  test_acc_dog        test_acc_breed      test_acc_f1score
0.1683708131313324  1.0             0.9996491074562073  0.8540669679641724  0.8547074829709408
------------------  --------------  ------------------  ------------------  ------------------
```

The Transfer learning along with data augmentations helps us achieve accuracy of 85% and a f1 score of 0.85 on the test set.

IV. Results

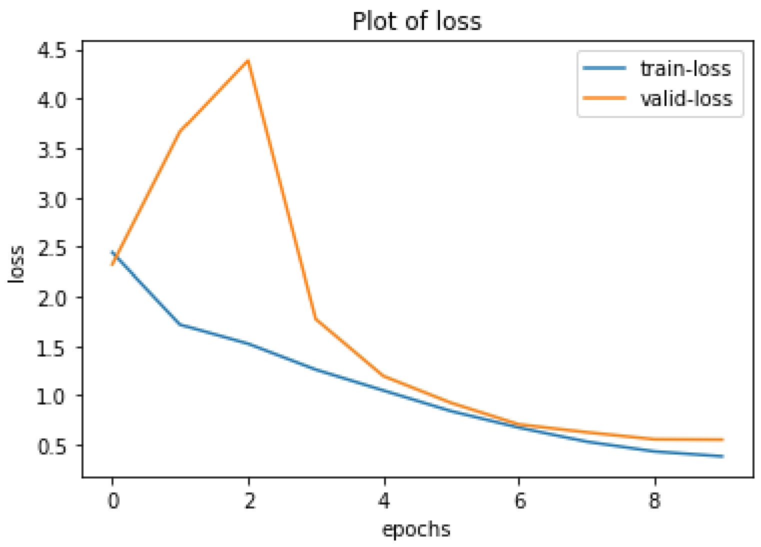
Model Evaluation and Validation

The final model chosen was a Resnet34 with custom head for our task. The Resnet34 was pre-trained on imagenet dataset for a top 1 error of 26.70 and a top 5 error of 8.58 torchvision models accuracy.

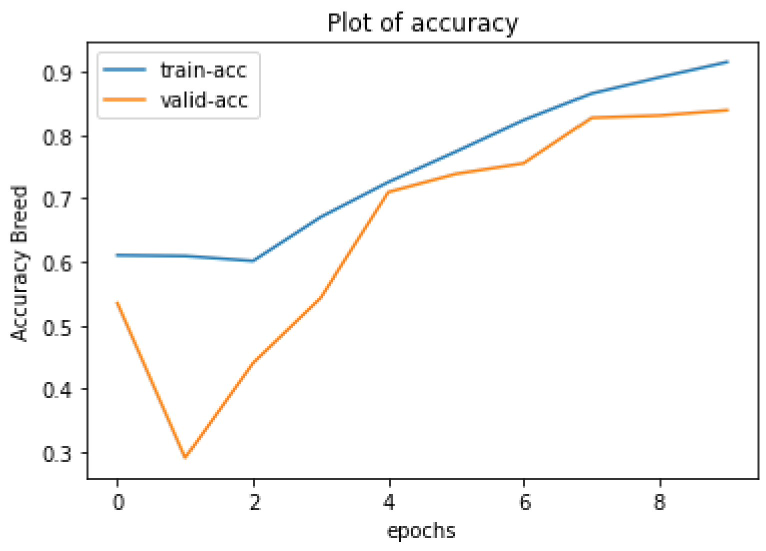
This model was finetuned for 10 epochs to accuracy of 83% on the validation set.
The model achieved 85% breed accuracy and a f1 score of 0.85 on the test set.

```
------------------  --------------  ------------------  ------------------  ------------------
test_loss           test_acc_human  test_acc_dog        test_acc_breed      test_acc_f1score
0.1683708131313324  1.0             0.9996491074562073  0.8540669679641724  0.8547074829709408
------------------  --------------  ------------------  ------------------  ------------------
```

Lets have a look at the plot of loss



Lets have a look at the plot of accuracy



The bump we see at first in both loss and accuracy plots is due to OneCycle policy. The learning rate starts small, goes to max lr and then goes to a minimum value. This leads to super convergence very quickly.

Justification

The benchmark model; the custom CNN, gave an accuracy of about 35%, while the final model gave an accuracy of 85% and a f1 score of 0.85 on the test set.
This clearly suggests that the final model is good for the task.

The key points that have led to these results are architecture of Residual Network and the Data Augmentation.

ResNet was introduced in the paper Deep Residual Learning for Image Recognition
, which won the 1st place on ImageNet detection, ImageNet localization, COCO detection and COCO segmentation.

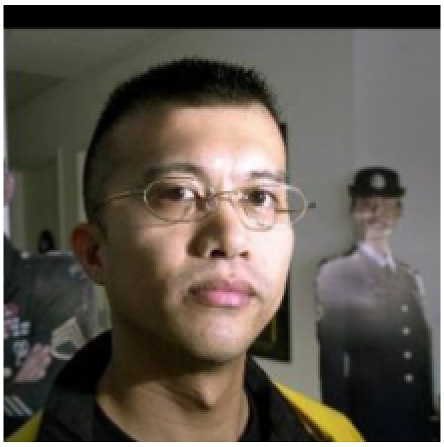
Data Augmentation , in layman terms, is a simple way to get more data for free. This paper talks on The Effectiveness of Data Augmentation in Image Classification using Deep Learning

V. Conclusion

Free-Form Visualization

Lets have a look at a few predictions:

```
Hi human, You look like a Cane corso
```

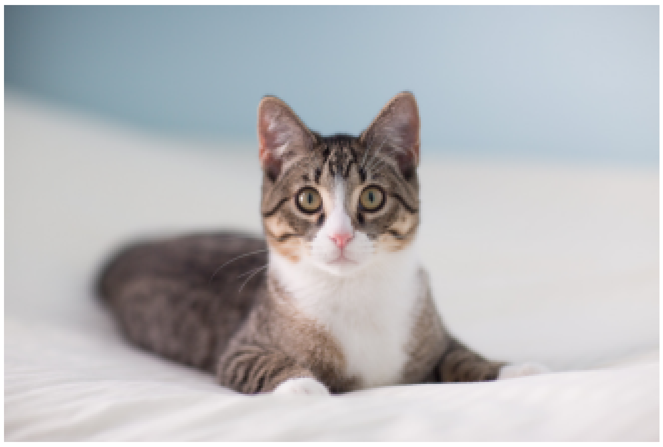


Correct Prediction

```
Hi Dog. The predicted breed is Affenpinscher
```

Correct Prediction

```
Hi Dog. The predicted breed is Boston terrier
```



Wrong Prediction. But A Boston Terrier looks similar.

```
Hi Dog. The predicted breed is Mastiff
```

Wrong Prediction.

Reflection

The complete project can be summarized using the following steps:

1. Get the data provided by udacity. The dataset has labels in the file names.
2. Create labellers to extract the labels from the file names and label the data.
3. Create Dataloaders for loading the dataset.
4. Use image augmentations to transform and normalize the data to prepare for training.
5. Use a Resnet34 architecture with a custom head for our purpose of dog vs human detection and breed classification.
6. Use a custom loss function, a combination of detection losses and breed classification loss.
7. Fine tune the model with Adam optimizer and OneCycle policy.
8. Predict for unseen test data.

The most challenging aspect for the project was labelling the data. The datasets are in completely different formats. I solved it using a Labeller class and two custom labelling functions for each task.

The interesting aspect of the project was that I chose to build a single model for two tasks. I had heard about such an approach in a talk by Andrej Karpathy, when he talked about how Tesla auto pilot works. This approach gets all the required predictions in one go rather than using a step based approach.

The final model has surpassed the minimum requirement of the project which was 60% accuracy. The model achieved a staggering 83% accuracy and f1score 0.85 which is quite impressive given the complexity of the task at hand. The model can be used in a general setting to solve these types of problems, though some improvements might be required.

Improvement

I might consider the following improvements in the future:

1. Collect more data as the dataset is not very huge. The provided dataset has enough to test our hypothesis that the model will learn.
2. Mixup and label smoothing can be used to further improve the accuracy of the models.
3. Using different architecture.
4. There are more transfer learning techniques that can be implemented:
    1. Gradual Unfreezing
    2. Using differential learning rates for the different parts of the model.