Dog breeds identification using Neural Networks

Aina Belloni 5007697

Giulia Beatrice Crespi 5009457

Libraries

In [1]:
```
%%capture
!pip install torch torchvision d2l
```

In [2]:
```
import pandas as pd
import torch
from torch.utils import data
import torchvision
from torchvision import transforms, datasets
import torch.optim as optim
import torch.nn as nn
from d2l import torch as d2l
import numpy as np
import matplotlib.pyplot as plt
d2l.use_svg_display()
%matplotlib inline

#libraries from object detection
from pathlib import Path
import cv2
import torch
import torchvision.transforms.functional as F
from torchvision.io import read_image
import os
```

In [3]:
```
from google.colab import drive
drive.mount('/content/drive', force_remount=True)
```

```
Mounted at /content/drive
```

Dataset Exploration

For this project, we chose Stanford Dog Breed dataset which contains images of 120 breeds of dogs from around the world. The dataset contains 20,580 images in total. The dataset has been built using images and annotations from ImageNet for the task of image categorization.

(Source Link for downloading the dataset: http://vision.stanford.edu/aditya86/ImageNetDogs/)

Goal

The goal of our project is to implement the best architecture to classify dogs into their breeds. We will start to look and explore our data and then we will implement with different models.

Importing the dataset

We import the dataset from Google Drive where each image folder is considered as one category.

In [4]:
```
my_dataset = datasets.ImageFolder(root='/content/drive/MyDrive/Advanced Programming/Dati dogs/Images')
```

Our dataset has 20580 images of 120 breeds of dogs.

In [5]:
```
len(my_dataset)
```

Out[5]:
```
20580
```

In [6]:
```python
len(my_dataset.classes)
```
Out[6]:
```
120
```

Since our folders have complex names with numbers and symbols we simplify the names of the classes to be more clear.

In [7]:
```python
my_dataset.classes[:5]
```
Out[7]:
```
['n02085620-Chihuahua',
 'n02085782-Japanese_spaniel',
 'n02085936-Maltese_dog',
 'n02086079-Pekinese',
 'n02086240-Shih_Tzu']
```

In [8]:
```python
name_classes = []
for element in my_dataset.classes:
  name_classes.append(element.split('-')[1])
my_dataset.classes = name_classes

my_dataset.classes[:5]
```
Out[8]:
```
['Chihuahua', 'Japanese_spaniel', 'Maltese_dog', 'Pekinese', 'Shih_Tzu']
```

Images per category

Since our dataset is a dictionary we had to count the numerosity of the indexes of the classes.

In [9]:
```python
calculate = False # if you want to do the calculations set True, if we want to upload from file set False
```

In [10]:
```python
if calculate:
  name_class = []
  i=0
  for key, value in my_dataset:
    if i%1000 == 0:
      print(i)
    name_class.append(value)
    i += 1

  from itertools import groupby
  count = [len(list(group)) for key, group in groupby(name_class)]
```

It takes very long time so we choose to save the counts once.

In [11]:
```python
if calculate:
  textfile = open("/content/drive/MyDrive/Advanced Programming/Dati dogs/count_length.txt", "w")
  for element in count:
    textfile.write(str(element) + "\n")
  textfile.close()
```

Upload the counts and visualize **quantitatively** and **graphically** how many images we have per category.

In [12]:
```python
if calculate == False:
  count0 = open("/content/drive/MyDrive/Advanced Programming/Dati dogs/count_length.txt", "r")
  count = count0.read().split()
  for i in range(0, len(count)):
      count[i] = int(count[i])
```

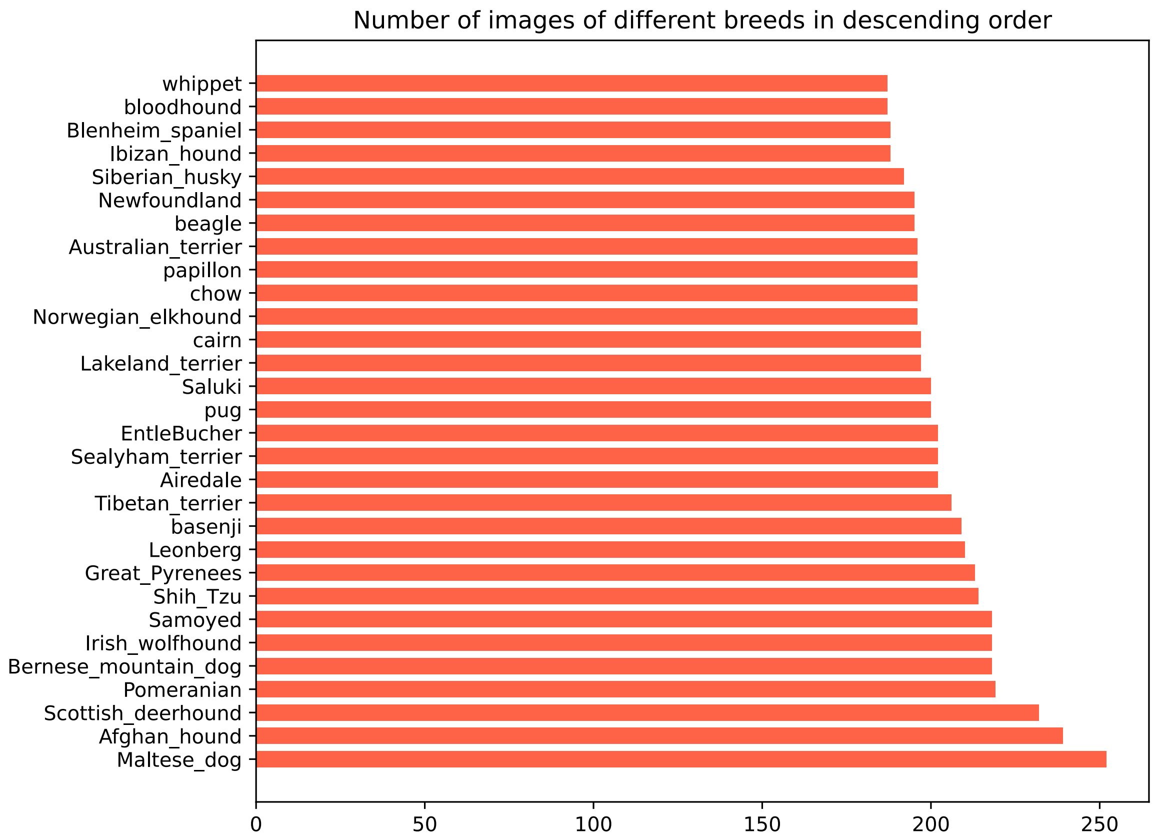
In [13]:
```python
data = {'Breeds':my_dataset.classes,
        'Number of Images': count}

df = pd.DataFrame(data)
df[:10]
```
Out[13]:

	Breeds	Number of Images
0	Chihuahua	152
1	Japanese_spaniel	185
2	Maltese_dog	252
3	Pekinese	149
4	Shih_Tzu	214
5	Blenheim_spaniel	188
6	papillon	196
7	toy_terrier	172

	Breeds	Number of Images
8	Rhodesian_ridgeback	172

In [14]:
```
import seaborn as sns
plt.figure(figsize=(8,7))
plt.title('Number of images of different breeds in descending order')
#sns.barplot(x = 'Number of Images', y = 'Breeds', data = df.sort_values('Number of Images', ascending =
plt.barh('Breeds', 'Number of Images', height=0.7, data=df.sort_values('Number of Images', ascending = Fa
```

Out[14]: `<BarContainer object of 30 artists>`



Boundary boxes

Our data folder contains also information on the boundary boxes, in the *Annoation* folder, so we had to extract information from that and we used the xml.etree.ElementTree package. Then we cropped all the images with the right boundary boxes and we saved all the cropped images in a folder *data* with the same structure as the Images original one.

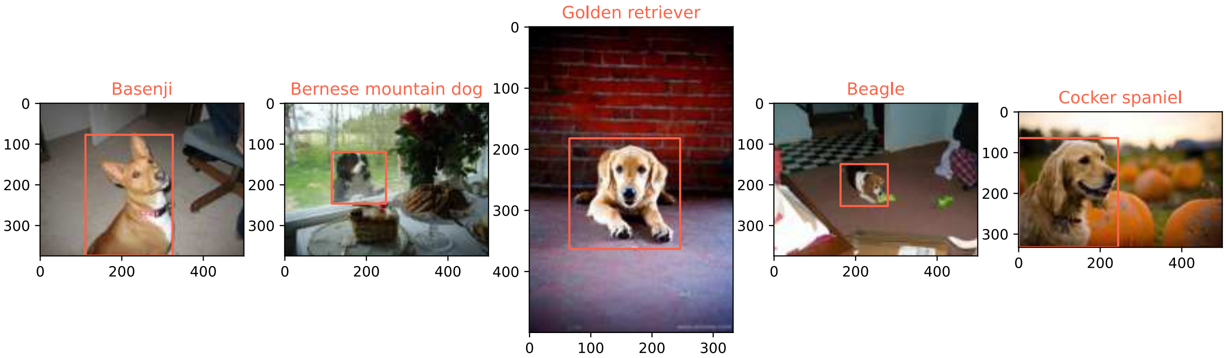
In [15]:
```
from PIL import Image
import xml.etree.ElementTree as ET
```

In [16]:
```
if 'data' not in os.listdir('/content/drive/MyDrive/Advanced Programming/Dati dogs'):
    os.mkdir('/content/drive/MyDrive/Advanced Programming/Dati dogs/data')

    for breed in my_dataset.classes:
      os.mkdir('/content/drive/MyDrive/Advanced Programming/Dati dogs/data/' + breed)

    for breed in my_dataset.classes:
      for file in os.listdir('/content/drive/MyDrive/Advanced Programming/Dati dogs/Annotation/' + breed)
        img = Image.open('/content/drive/MyDrive/Advanced Programming/Dati dogs/Images/' + breed + '/' +
        tree = ET.parse('/content/drive/MyDrive/Advanced Programming/Dati dogs/Annotation/' + breed + '/'
        xmin = int(tree.getroot().findall('object')[0].find('bndbox').find('xmin').text)
        xmax = int(tree.getroot().findall('object')[0].find('bndbox').find('xmax').text)
        ymin = int(tree.getroot().findall('object')[0].find('bndbox').find('ymin').text)
        ymax = int(tree.getroot().findall('object')[0].find('bndbox').find('ymax').text)
        img = img.crop((xmin,ymin,xmax,ymax))
        img = img.convert('RGB')
        img.save('/content/drive/MyDrive/Advanced Programming/Dati dogs/data_new/' + breed + '/' + file +
```

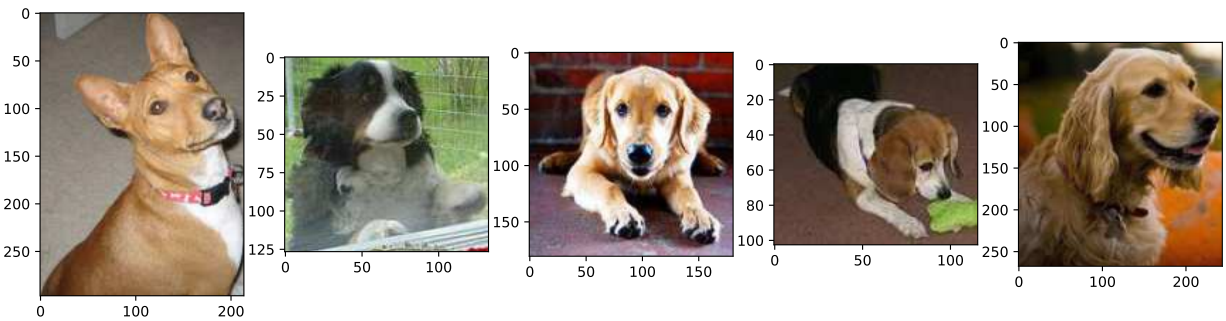
Show an example of 5 images

In [17]:
```python
list_5 = ['/n02110806-basenji/n02110806_18',
          '/n02107683-Bernese_mountain_dog/n02107683_1003',
          '/n02099601-golden_retriever/n02099601_109',
          '/n02088364-beagle/n02088364_769',
          '/n02102318-cocker_spaniel/n02102318_2073']

breeds = ['Basenji','Bernese mountain dog','Golden retriever','Beagle','Cocker spaniel']

link_images = '/content/drive/MyDrive/Advanced Programming/Dati dogs/Images'
link_data = '/content/drive/MyDrive/Advanced Programming/Dati dogs/data'
link_annotation = '/content/drive/MyDrive/Advanced Programming/Dati dogs/Annotation'

bbox = [] ; images = [] ; img_cropped = []
for file in list_5:
  img = Image.open(link_images + file + '.jpg')
  tree = ET.parse(link_annotation + file)
  xmin = int(tree.getroot().findall('object')[0].find('bndbox').find('xmin').text)
  xmax = int(tree.getroot().findall('object')[0].find('bndbox').find('xmax').text)
  ymin = int(tree.getroot().findall('object')[0].find('bndbox').find('ymin').text)
  ymax = int(tree.getroot().findall('object')[0].find('bndbox').find('ymax').text)
  bbox.append([(xmin,ymin), (xmax,ymax)])
  images.append(img)
  img_cropped.append(Image.open(link_data + file + '.jpg'))
```

In [18]:
```python
fig = plt.figure(figsize=(15, 15))
for i in range(5):
  plt.subplot(1,5,i+1)
  #img = plt.imread(data_dir + 'images/Images/' + breed + '/' + dog + '.jpg')
  img=images[i]
  plt.imshow(img)
  xmin,ymin = bbox[i][0]
  xmax, ymax = bbox[i][1]
  plt.plot([xmin, xmax, xmax, xmin, xmin], [ymin, ymin, ymax, ymax, ymin], c='tomato')
  plt.title(breeds[i], c='tomato')
```



In [19]:
```python
fig = plt.figure(figsize=(15, 15))
for i in range(5):
  plt.subplot(1,5,i+1)
  plt.imshow(img_cropped[i])
```



Object detection

Useful **libraries** for the object detection

In [20]:
```
import torch
import torchvision.transforms.functional as F
from torchvision.io import read_image
from torchvision.utils import make_grid
from torchvision.models.detection import fasterrcnn_resnet50_fpn
import torchvision.models as models
from torchvision.transforms.functional import convert_image_dtype
from torchvision.utils import draw_bounding_boxes
```

We defined a function to show images, with a proper grid.

In [21]:
```
def show(imgs, transformed = False):
    if not isinstance(imgs, list):
      imgs = [imgs]
    fig = plt.figure(figsize=(15, 15))
    for i, img in enumerate(imgs):
      plt.subplot(1,len(imgs),i+1)
      img = F.to_pil_image(img)
      plt.imshow(np.asarray(img))
```

Transform images into the right format for the model below, in particular we resized the images to 400x400 and then to a torch Tensor type with 3 channels of colors.

In [22]:
```
dogs0 = []; dogs1 = []; dogs= []
i=0
for file in list_5:
  dogs0.append(cv2.imread(link_images + file + '.jpg'))
  dogs1.append(cv2.resize(dogs0[i],dsize=(400,400),interpolation=cv2.INTER_CUBIC))
  dogs.append(torch.from_numpy(cv2.cvtColor(dogs1[i], cv2.COLOR_BGR2RGB)).permute(2,0,1))
  i += 1

print(dogs0[0].shape)
print(dogs1[0].shape)
print(dogs[0].shape)
print(type(dogs[0]))
```

```
(375, 500, 3)
(400, 400, 3)
torch.Size([3, 400, 400])
<class 'torch.Tensor'>
```

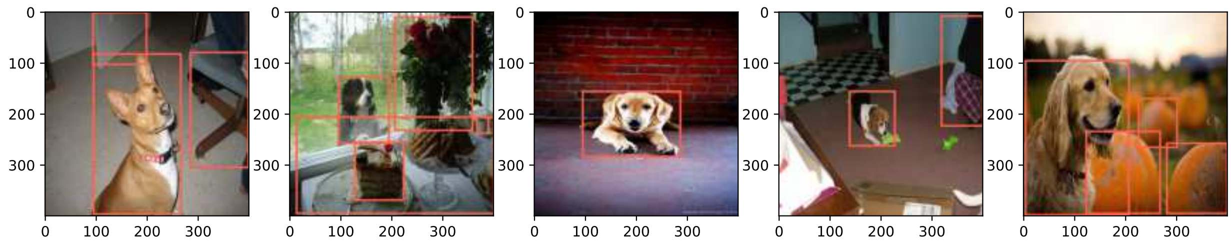
We tried with a neural network architecture to detect objects, in particular a **ResNet50 model** : fasterrcnn_resnet50_fpn. This is a pretrained model downloaded from PyTorch, we only use it in evaluation mode on our 5 images.

In [23]:
```
batch_int = torch.stack(dogs)
batch = convert_image_dtype(batch_int, dtype=torch.float)

model = fasterrcnn_resnet50_fpn(pretrained=True, progress=False)
model = model.eval()

outputs = model(batch)
print(outputs[0])
```

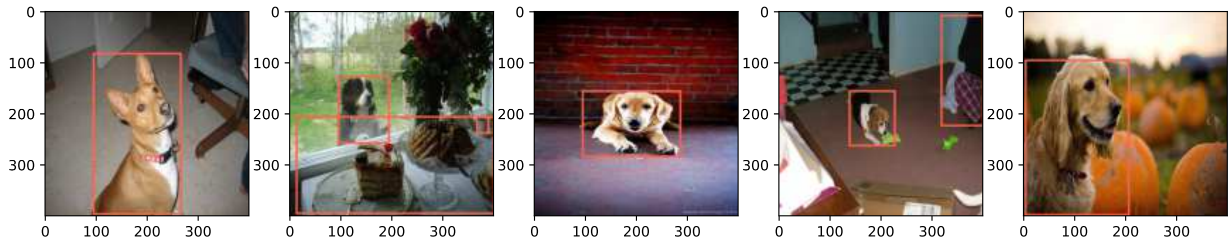
```
Downloading: "https://download.pytorch.org/models/fasterrcnn_resnet50_fpn_coco-258fb6c6.pth" to /root/.ca
che/torch/hub/checkpoints/fasterrcnn_resnet50_fpn_coco-258fb6c6.pth
/usr/local/lib/python3.7/dist-packages/torch/functional.py:445: UserWarning: torch.meshgrid: in an upcomi
ng release, it will be required to pass the indexing argument. (Triggered internally at  ../aten/src/ATen
/native/TensorShape.cpp:2157.)
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
{'boxes': tensor([[ 93.8180,  82.0909, 266.0742, 397.5267],
        [282.1686,  79.5740, 398.8734, 306.9021],
        [ 92.4846,   0.0000, 199.9191, 105.2620],
        [275.0713,   4.7055, 396.7171, 185.4989],
        [318.8956,   0.4815, 395.6828, 147.0160],
        [380.2063, 254.5570, 399.4890, 361.4636],
        [276.9422,  24.4986, 399.4294, 219.0072],
        [376.1681, 189.4520, 397.1863, 371.3427],
        [289.7142,   0.0000, 399.7518, 359.5795],
        [380.5258,  45.3594, 400.0000, 144.6729],
        [276.1216,  27.1378, 395.7346, 182.8127],
        [294.2037, 108.8638, 397.2927, 294.8630],
        [283.4308,   0.0000, 320.9474,  72.9320],
        [214.4345,   7.3065, 392.7836, 310.4259]], grad_fn=<StackBackward0>), 'labels': tensor([18, 62, 8
2,  1,  1,  1, 62,  1,  1,  1, 63, 15, 62, 62]), 'scores': tensor([0.9987, 0.8962, 0.8276, 0.6821, 0.571
5, 0.4423, 0.4003, 0.1988, 0.1830,
        0.1780, 0.1421, 0.1172, 0.0680, 0.0671], grad_fn=<IndexBackward0>)}
```

Which are the objects detected with the model? We can show them with the following boxes.

In [24]:
```
score_threshold = 0.8
dogs_with_boxes = [
    draw_bounding_boxes(dog_int, boxes=output['boxes'][output['scores'] > score_threshold], width=4, colo
    for dog_int, output in zip(batch_int, outputs)
]
show(dogs_with_boxes)
```



In [25]:
```
score_threshold = 0.95
dogs_with_boxes = [
    draw_bounding_boxes(dog_int, boxes=output['boxes'][output['scores'] > score_threshold], width=4, colo
    for dog_int, output in zip(batch_int, outputs)
]
show(dogs_with_boxes)
```



As we can see from those images the objects detected are not always dogs, but if we incrase the score threshold it's more probable to idetify dogs. Anyway for the neural networks models used for Image classification it's better if we crop the images using the information that we already have. If we did not have informations about the

Using fasterrcnn_resnet50_fpn could be a good solution with a dataset with no information about the bounding boxes.

Load the data

Now we start to prepare the data for training and testing NN models.

The images have to be transformed, first of all we had to resize to 224 in order to use the following neural networks, then we have to normalize but to do that we should know from the dataset the mean and the standard deviation of the images for all the channels (R,B,G). In this way, our input images have a size 224x224x3. Since the calculations would be expensive and time consuming, and since the images we have are the same as the basic ones in the ImageNet database, we can use the standard normalization with mean [0.485, 0.456, 0.406] and standard deviation [0.229, 0.224, 0.225]. This normalization transforms each channel of the input torch.Tensor as $output[channel] = \frac{input[channel] - mean[channel]}{std[channel]}$

We decided to only resize the images and not to crop them since we had already cropped them using the boundary boxes.

Finally, since we have few data for each category, to reduce overfitting we decided to do **Data Augmentation**: it is a strategy to significantly increase the diversity of data, in our case by flipping the images horizontally (not vertically since vertically doesn't mean much for dogs), and distorting them.

In [26]:
```
data_transform = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.RandomHorizontalFlip(0.6),
    transforms.RandomPerspective(distortion_scale = 0.5, p=0.6),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
```

We used ImageFolder to create a dataset taking the previous cropped and transformed images from the Drive folder and the correct labels.

In [27]:
```
my_dataset = datasets.ImageFolder(root='/content/drive/MyDrive/Advanced Programming/Dati dogs/data',
                                  transform=data_transform)
```

As previous we rename the classes

In [28]:
```
name_classes = []
for element in my_dataset.classes:
  name_classes.append(element.split('-')[1])
```

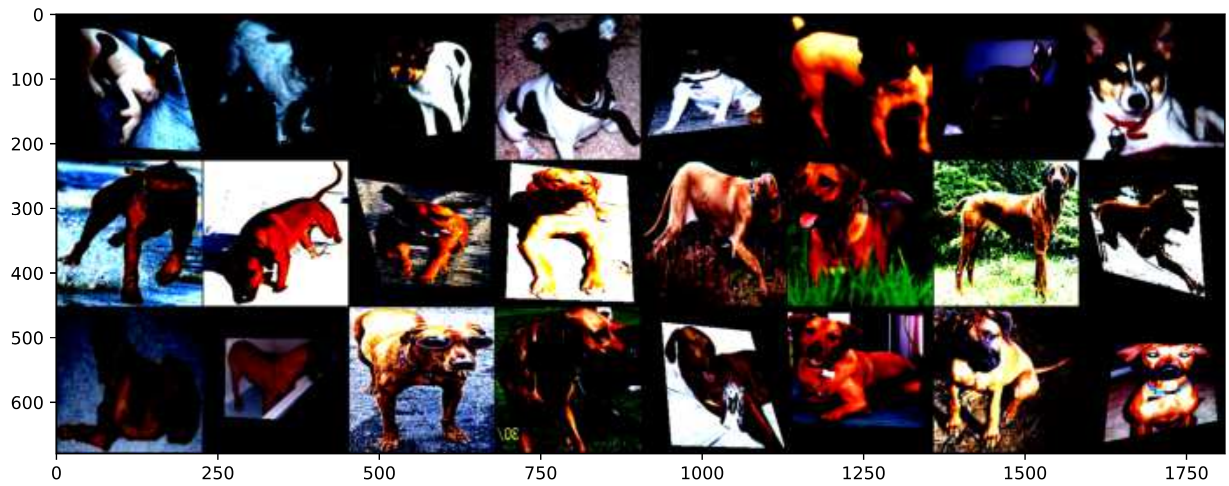
In [29]:
```
my_dataset.classes = name_classes
```

Visualization of 5 transformed images

In [30]:
```
import matplotlib.pyplot as plt
import numpy as np

#Function to show some random images
def imshow(img, single = False):
    #img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    if single == True:
      fig = plt.figure(figsize=(3, 3))
    else:
      fig = plt.figure(figsize=(12, 12))
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()
```

In [31]:
```
images = []
for i in range(24):
  images.append(my_dataset[1500+i][0])
imshow(torchvision.utils.make_grid(images))
```

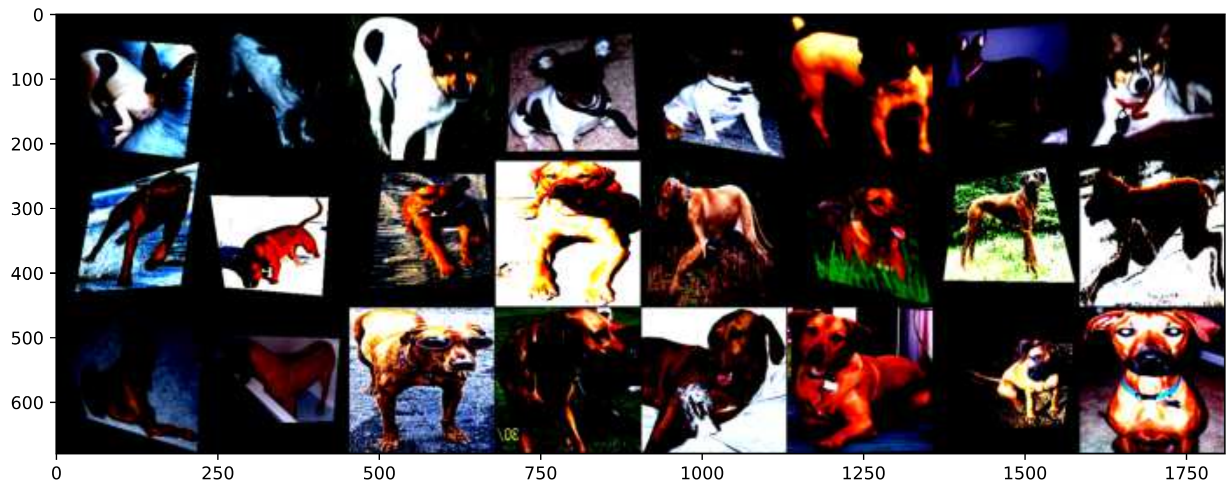
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for intege
rs).
```



We can notice that if we run the same code again the images are the same but some of them are transformed for the data augmentation.

In [32]:
```
images = []
for i in range(24):
  images.append(my_dataset[1500+i][0])
imshow(torchvision.utils.make_grid(images))
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for intege
rs).
```



Divide into train and test

We took 80% of the images for the training, so 16464, and 20% for the test, so the remaining 4116.

In [33]:
```
train_size = int(0.8 * len(my_dataset))  # train 0.8
test_size = len(my_dataset) - train_size
train_dataset, test_dataset = torch.utils.data.random_split(my_dataset, [train_size, test_size])
```

In [34]:
```
len(train_dataset), len(test_dataset), len(my_dataset)
```

Out[34]:
```
(16464, 4116, 20580)
```

We checked if the number of images for training and test dataset for each category are balanced.

In [35]:
```
# train
calculate = False
if calculate:
  a = []
  i=0
  for key, value in train_dataset:
    if i%1000 == 0:
      print(i)
    a.append(value)
    i += 1

  # save
  textfile = open("/content/drive/MyDrive/Advanced Programming/Dati dogs/count_train_per_class.txt", "w")
  for element in a:
    textfile.write(str(element) + "\n")
  textfile.close()
```

In [36]:
```
# test
calculate = False
if calculate:
  a = []
  i=0
  for key, value in test_dataset:
    if i%1000 == 0:
      print(i)
    a.append(value)
    i += 1

  # save
  textfile = open("/content/drive/MyDrive/Advanced Programming/Dati dogs/count_test_per_class.txt", "w")
  for element in a:
    textfile.write(str(element) + "\n")
  textfile.close()
```

In [37]:
```
train_len_class = open("/content/drive/MyDrive/Advanced Programming/Dati dogs/count_train_per_class.txt",
train_len_class = train_len_class.read().split()
for i in range(0, len(train_len_class)):
  train_len_class[i] = int(train_len_class[i])

test_len_class = open("/content/drive/MyDrive/Advanced Programming/Dati dogs/count_test_per_class.txt", '
test_len_class = test_len_class.read().split()
for i in range(0, len(test_len_class)):
  test_len_class[i] = int(test_len_class[i])
```

In [38]:
```
df_train = pd.DataFrame({'Classes': train_len_class,'count':[1]*len(train_len_class)})
df_test = pd.DataFrame({'Classes': test_len_class,'count':[1]*len(test_len_class)})
new_df = pd.DataFrame({'Breeds': name_classes,'Count train': df_train.groupby(['Classes']).sum()['count']
new_df
```

Out[38]:

	Breeds	Count train	Count test
Classes			
0	Chihuahua	121	33
1	Japanese_spaniel	151	43
2	Maltese_dog	195	49
3	Pekinese	122	26
4	Shih_Tzu	173	43
...	...	...	...
115	standard_poodle	130	32
116	Mexican_hairless	120	34

	Breeds	Count train	Count test
Classes			
117	dingo	128	30
118	dhole	125	29

Let's define the **Dataloader** that will feed the data in batches to the neural network. Dataloader is used to for creating training and testing dataloader that load data to the neural network in a defined manner. This is needed because all the data from the dataset cannot be loaded to the memory at once, so the amount of dataloaded to the memory and then passed to the neural network needs to be controlled using parameters such as *batch_size* (batch_size controls how many samples per batch to load). Dataloader is a construct of PyTorch library.

In [39]:
```
from torch.utils import data
batch_size = 16
train_dataloader = data.DataLoader(train_dataset, batch_size = 16, shuffle = True, num_workers=2)
test_dataloader = data.DataLoader(test_dataset, batch_size = 16, shuffle = False, num_workers=2)
```

In [40]:
```
train_iter = iter(train_dataloader)
test_iter = iter(test_dataloader)
X, y = next(train_iter)  # get one minibatch

print(my_dataset[0][0].shape)
print(X.shape) # X is (batch_size, channels, img height, img width)
print(X[0].shape) # one image in proper channel(s)
print(y.shape)
print(y) # y: 0-based index values representing class labels of the minibatch
print(torch.is_tensor(X[0]))
```

```
torch.Size([3, 224, 224])
torch.Size([16, 3, 224, 224])
torch.Size([3, 224, 224])
torch.Size([16])
tensor([ 85,  74, 111,  25,  82,  51,  64,  29,  88,  67,  39,  39,  31,  38,
         17,  67])
True
```

Define useful functions to train and test the model

We define a function to train the model. The outputs are two lists to save the accuracy and the loss for each epoch.

In [41]:
```
import copy
train_loss = []
train_accu = []

def train(epoch, model):
    print('\nEpoch : %d'%epoch)
    running_loss = 0.0
    steps = 0
    correct = 0
    total = 0
    best_model = copy.deepcopy(model.state_dict())

    model.train()
    for i, data in enumerate(train_dataloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data[0].to(device), data[1].to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        #loss
        running_loss += loss.item()
        steps += 1
        #accuracy
        _, predicted = torch.max(outputs.data, dim=1)
        correct += (predicted==labels).sum().item()
        total+=labels.size(0)
        best_model = copy.deepcopy(model.state_dict())

        # print statistics
        if i % 100 == 0:    # print every 100 mini-batches
            print('[batch: %d] train loss: %.3f, train accuracy: %.2f %%' %( i + 1, running_loss / steps,
            running_loss = 0.0

    epoch_loss = running_loss/steps
    train_loss.append(epoch_loss)

    epoch_accu = (correct*100)/total
    train_accu.append(epoch_accu)
    print('\n [EPOCH %d] train loss: %.3f, train accuracy: %.2f %%' %( epoch, running_loss / steps, (corr

    return train_loss, train_accu
```

We define a function to test our model on some data that was not seen before by the model. The function outputs the test accuracy and the predicted labels for each image.

In [42]:
```
def test(model):
  correct = 0
  total = 0
  predictions = []
  model.eval()
  with torch.no_grad():
    for data in test_dataloader:
        images, labels = data[0].to(device), data[1].to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)

        #find predictions
        predictions.extend(predicted)
        #accuracy
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

  test_accuracy = (correct*100)/total
  predictions = torch.stack(predictions).cpu()
  return predictions, test_accuracy
```

We alse defined the accuracy for each class of the model in the following function.

In [43]:
```
def test_accuracy_per_class(model, name_classes):
  n_classes = len(name_classes)
  class_correct = list(0. for i in range(n_classes))
  class_total = list(0. for i in range(n_classes))
  accuracy = list(0. for i in range(n_classes))
  with torch.no_grad():
    for data in test_dataloader:
      images, labels = data[0].to(device), data[1].to(device)
      outputs = model(images)
      _, predicted = torch.max(outputs, 1)
      for label, pred in zip(labels, predicted):
        if label == pred:
          class_correct[label] += 1
        class_total[label] += 1

  for i in range(n_classes):
    if class_total[i] > 0:
          accuracy[i] = 100 * class_correct[i] / class_total[i]
    else:
          accuracy[i] = 0

  return accuracy
```

Alexnet - not pretrained

The first model we implemented is AlexNet.

AlexNet is a CNN architecture that consists of 5 two-dimensional **convolution layers**, followed by 3 **fully connected layers**, total of eight layers.

The convolution layers uses trainable kernels or filters to perform convolution operations, which involves moving the kernels over the input in steps called strides. The output is then passed through a non-linearity **ReLu activation function**. Some convolutional layers are followed by a **Max Pooling layer**, which helps reducing overfitting and uses kernels of dimension 3x3.

The first two fully connected layers have a **dropout layer** associated with them, with a dropout ratio of 0.5, which also help reducing overfitting. The final fully connected layer has **120 outputs** because of the number of our classes.

In [44]:
```
#DEFINE THE NUMBER OF CLASSES
n_classes = 120
```

We uploaded the model from the pytorch website.

In [ ]:
```
#Now using the AlexNet
AlexNet_model = torch.hub.load('pytorch/vision:v0.10.0', 'alexnet', pretrained=False)

#Model description
AlexNet_model.eval()
```

```
Using cache found in /root/.cache/torch/hub/pytorch_vision_v0.10.0
```

Out[ ]:
```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

Since we want a model that classifies 120 classes, we changed the last layer in order to have the output size equal to 120. We also update the previous linear layer in order to smooth the resizing of the output.

In [ ]:
```
AlexNet_model.classifier[4] = nn.Linear(4096,1024)
AlexNet_model.classifier[6] = nn.Linear(1024, n_classes)
```

We now verify the device where the model is running and we move the input and AlexNet_model to GPU for speed, if available.

In [ ]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)

AlexNet_model.to(device)
```

```
cpu
```

Out[ ]:
```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=1024, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=1024, out_features=120, bias=True)
  )
)
```

Here we define the loss function and optimizer. The *criterion* is used the calculate the difference in the output created by the model and the actual output. We use the cross entropy loss function since we are doing a classification task. The *Optimizer* is used to update the weights of the neural network to improve its performance. As optimizer we use stochastic gradient descent.

In [ ]:
```
#Loss
criterion = nn.CrossEntropyLoss()

#Optimizer(SGD)
optimizer = optim.SGD(AlexNet_model.parameters(), lr=0.001, momentum=0.9)
```

Training the model

Let's train our model. We set the number of epochs and a early stopping value to stop our model in case there is not great improvement in the accuracy.

In [ ]:
```
%%time
Early_stopping_value = 0.00001
EPOCHS = 100
train_accu = []
train_loss = []

for epoch in range(1, EPOCHS+1):     # loop over the dataset multiple times
  train(epoch, AlexNet_model)

  if epoch > 2:
    if abs(train_accu[epoch-2] - train_accu[epoch-1]) < Early_stopping_value:
      print("\nEarly stopping. Epoch:", epoch)
      break

print('\nFinished Training of AlexNet')
```

```
Epoch : 1
[batch: 1] train loss: 4.796, train accuracy: 0.00 %
[batch: 101] train loss: 4.740, train accuracy: 1.11 %
[batch: 201] train loss: 2.381, train accuracy: 1.24 %
[batch: 301] train loss: 1.590, train accuracy: 1.20 %
[batch: 401] train loss: 1.194, train accuracy: 1.15 %
[batch: 501] train loss: 0.955, train accuracy: 1.09 %
[batch: 601] train loss: 0.797, train accuracy: 1.10 %
[batch: 701] train loss: 0.683, train accuracy: 1.08 %
[batch: 801] train loss: 0.598, train accuracy: 1.08 %
[batch: 901] train loss: 0.531, train accuracy: 1.09 %
[batch: 1001] train loss: 0.478, train accuracy: 1.09 %

 [EPOCH 1] train loss: 0.130, train accuracy: 1.10 %
```

```
Epoch : 96
[batch: 1] train loss: 0.950, train accuracy: 75.00 %
[batch: 101] train loss: 0.466, train accuracy: 84.84 %
[batch: 201] train loss: 0.239, train accuracy: 84.92 %
[batch: 301] train loss: 0.164, train accuracy: 84.93 %
[batch: 401] train loss: 0.136, train accuracy: 84.71 %
[batch: 501] train loss: 0.102, train accuracy: 84.58 %
[batch: 601] train loss: 0.080, train accuracy: 84.68 %
[batch: 701] train loss: 0.076, train accuracy: 84.50 %
[batch: 801] train loss: 0.061, train accuracy: 84.54 %
[batch: 901] train loss: 0.062, train accuracy: 84.39 %
[batch: 1001] train loss: 0.049, train accuracy: 84.45 %

 [EPOCH 96] train loss: 0.013, train accuracy: 84.49 %

Epoch : 97
[batch: 1] train loss: 0.330, train accuracy: 87.50 %
[batch: 101] train loss: 0.459, train accuracy: 85.52 %
[batch: 201] train loss: 0.228, train accuracy: 85.82 %
[batch: 301] train loss: 0.161, train accuracy: 85.71 %
[batch: 401] train loss: 0.124, train accuracy: 85.27 %
[batch: 501] train loss: 0.089, train accuracy: 85.44 %
[batch: 601] train loss: 0.083, train accuracy: 85.15 %
[batch: 701] train loss: 0.078, train accuracy: 84.90 %
[batch: 801] train loss: 0.062, train accuracy: 84.82 %
[batch: 901] train loss: 0.053, train accuracy: 84.81 %
[batch: 1001] train loss: 0.052, train accuracy: 84.78 %

 [EPOCH 97] train loss: 0.014, train accuracy: 84.80 %

Epoch : 98
[batch: 1] train loss: 0.587, train accuracy: 81.25 %
[batch: 101] train loss: 0.488, train accuracy: 85.40 %
[batch: 201] train loss: 0.233, train accuracy: 85.48 %
[batch: 301] train loss: 0.182, train accuracy: 84.97 %
[batch: 401] train loss: 0.128, train accuracy: 84.93 %
[batch: 501] train loss: 0.110, train accuracy: 84.72 %
[batch: 601] train loss: 0.074, train accuracy: 84.90 %
[batch: 701] train loss: 0.068, train accuracy: 84.96 %
[batch: 801] train loss: 0.058, train accuracy: 84.96 %
[batch: 901] train loss: 0.058, train accuracy: 84.79 %
[batch: 1001] train loss: 0.053, train accuracy: 84.69 %

 [EPOCH 98] train loss: 0.013, train accuracy: 84.69 %

Epoch : 99
[batch: 1] train loss: 0.874, train accuracy: 75.00 %
[batch: 101] train loss: 0.448, train accuracy: 85.77 %
[batch: 201] train loss: 0.211, train accuracy: 86.19 %
[batch: 301] train loss: 0.160, train accuracy: 85.67 %
[batch: 401] train loss: 0.115, train accuracy: 85.72 %
[batch: 501] train loss: 0.078, train accuracy: 85.98 %
[batch: 601] train loss: 0.084, train accuracy: 85.62 %
[batch: 701] train loss: 0.078, train accuracy: 85.32 %
[batch: 801] train loss: 0.052, train accuracy: 85.49 %
[batch: 901] train loss: 0.058, train accuracy: 85.30 %
[batch: 1001] train loss: 0.048, train accuracy: 85.28 %

 [EPOCH 99] train loss: 0.016, train accuracy: 85.22 %

Epoch : 100
[batch: 1] train loss: 0.170, train accuracy: 93.75 %
[batch: 101] train loss: 0.467, train accuracy: 84.47 %
[batch: 201] train loss: 0.212, train accuracy: 86.07 %
[batch: 301] train loss: 0.150, train accuracy: 86.05 %
[batch: 401] train loss: 0.130, train accuracy: 85.50 %
[batch: 501] train loss: 0.087, train accuracy: 85.68 %
[batch: 601] train loss: 0.086, train accuracy: 85.60 %
[batch: 701] train loss: 0.070, train accuracy: 85.50 %
[batch: 801] train loss: 0.059, train accuracy: 85.42 %
[batch: 901] train loss: 0.047, train accuracy: 85.46 %
[batch: 1001] train loss: 0.054, train accuracy: 85.30 %

 [EPOCH 100] train loss: 0.013, train accuracy: 85.25 %
```

Let's save the model

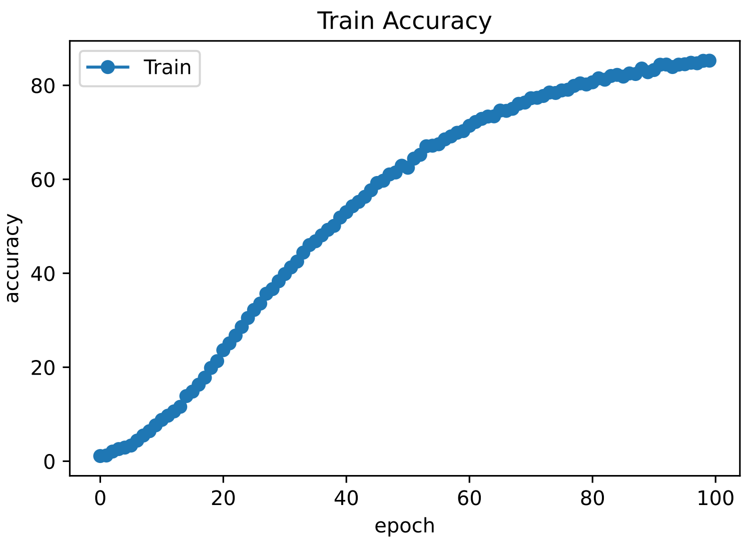
In [ ]:

```
%%capture
import pickle
#with open('/content/drive/MyDrive/Advanced Programming/alexnet_model1_final', 'wb') as files:
#  pickle.dump(AlexNet_model, files)

with open('/content/drive/MyDrive/Advanced Programming/alexnet_model1' , 'rb') as f:
  AlexNet_model = pickle.load(f)
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
AlexNet_model.to(device)
```

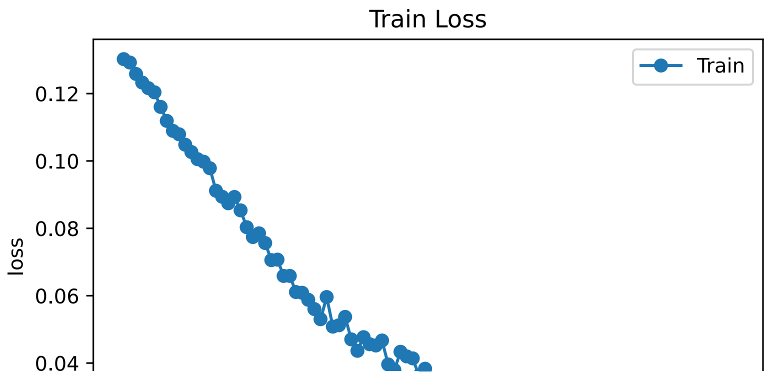
In [ ]:
```
#with open('/content/drive/MyDrive/Advanced Programming/train_accu_alexnet1_final', 'wb') as files:
 # pickle.dump(train_accu, files)

with open('/content/drive/MyDrive/Advanced Programming/train_accu_alexnet1' , 'rb') as f:
  train_accu = pickle.load(f)
```

In [ ]:
```
with open('/content/drive/MyDrive/Advanced Programming/train_loss_alexnet1_final', 'wb') as files:
  pickle.dump(train_loss, files)

#with open('/content/drive/MyDrive/Advanced Programming/train_loss_alexnet1_final' , 'rb') as f:
  #train_loss = pickle.load(f)
```

In [ ]:
```
import matplotlib.pyplot as plt
from IPython.display import HTML, display, Image

display(HTML("""
<style>
#output-body {
    display: flex;
    align-items: center;
    justify-content: center;
}
</style>
"""))

plt.plot(train_accu,'-o')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.legend(['Train'])
plt.title('Train Accuracy')

plt.show()
```



In [ ]:
```
display(HTML("""
<style>
#output-body {
    display: flex;
    align-items: center;
    justify-content: center;
}
</style>
"""))

plt.plot(train_loss,'-o')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend(['Train'])
plt.title("Train Loss")

plt.show()
```



Testing the model

In [ ]:
```
t = test(AlexNet_model)
predictions, acc = t[0], t[1]
print('Accuracy of the network on the test images: %.2f %%' % acc)
```

```
Accuracy of the network on the test images: 85.59 %
```

In [ ]:
```
#Testing classification accuracy for individual classes.

acc = test_accuracy_per_class(AlexNet_model, name_classes)
for i in range(n_classes):
    print('Accuracy of %5s : %2d %%' %(name_classes[i],acc[i]) )
```

```
Accuracy of Chihuahua : 80 %
Accuracy of Japanese_spaniel : 91 %
Accuracy of Maltese_dog : 88 %
Accuracy of Pekinese : 96 %
Accuracy of Shih_Tzu : 82 %
Accuracy of Blenheim_spaniel : 92 %
Accuracy of papillon : 93 %
Accuracy of toy_terrier : 80 %
Accuracy of Rhodesian_ridgeback : 76 %
Accuracy of Afghan_hound : 90 %
Accuracy of basset : 73 %
Accuracy of beagle : 89 %
Accuracy of bloodhound : 91 %
Accuracy of bluetick : 93 %
Accuracy of black_and_tan_coonhound : 87 %
Accuracy of Walker_hound : 90 %
Accuracy of English_foxhound : 93 %
Accuracy of redbone : 86 %
Accuracy of borzoi : 83 %
Accuracy of Irish_wolfhound : 87 %
Accuracy of Italian_greyhound : 80 %
Accuracy of whippet : 79 %
Accuracy of Ibizan_hound : 85 %
Accuracy of Norwegian_elkhound : 86 %
Accuracy of otterhound : 85 %
Accuracy of Saluki : 77 %
Accuracy of Scottish_deerhound : 84 %
Accuracy of Weimaraner : 89 %
Accuracy of Staffordshire_bullterrier : 70 %
Accuracy of American_Staffordshire_terrier : 71 %
Accuracy of Bedlington_terrier : 89 %
Accuracy of Border_terrier : 87 %
Accuracy of Kerry_blue_terrier : 91 %
Accuracy of Irish_terrier : 83 %
Accuracy of Norfolk_terrier : 94 %
Accuracy of Norwich_terrier : 90 %
Accuracy of Yorkshire_terrier : 94 %
Accuracy of wire_haired_fox_terrier : 87 %
Accuracy of Lakeland_terrier : 89 %
Accuracy of Sealyham_terrier : 88 %
Accuracy of Airedale : 77 %
Accuracy of cairn : 92 %
Accuracy of Australian_terrier : 83 %
Accuracy of Dandie_Dinmont : 94 %
Accuracy of Boston_bull : 96 %
Accuracy of miniature_schnauzer : 79 %
Accuracy of giant_schnauzer : 76 %
Accuracy of standard_schnauzer : 83 %
Accuracy of Scotch_terrier : 93 %
Accuracy of Tibetan_terrier : 85 %
Accuracy of silky_terrier : 80 %
Accuracy of soft_coated_wheaten_terrier : 100 %
Accuracy of West_Highland_white_terrier : 93 %
Accuracy of Lhasa : 90 %
Accuracy of flat_coated_retriever : 88 %
Accuracy of curly_coated_retriever : 88 %
Accuracy of golden_retriever : 95 %
Accuracy of Labrador_retriever : 62 %
```

```
Accuracy of Chesapeake_Bay_retriever : 76 %
Accuracy of German_short_haired_pointer : 91 %
Accuracy of vizsla : 93 %
Accuracy of English_setter : 78 %
Accuracy of Irish_setter : 80 %
Accuracy of Gordon_setter : 69 %
Accuracy of Brittany_spaniel : 89 %
Accuracy of clumber : 92 %
Accuracy of English_springer : 96 %
Accuracy of Welsh_springer_spaniel : 83 %
Accuracy of cocker_spaniel : 73 %
Accuracy of Sussex_spaniel : 84 %
Accuracy of Irish_water_spaniel : 91 %
Accuracy of kuvasz : 88 %
Accuracy of schipperke : 78 %
Accuracy of groenendael : 88 %
Accuracy of malinois : 88 %
Accuracy of briard : 86 %
Accuracy of kelpie : 75 %
Accuracy of komondor : 80 %
Accuracy of Old_English_sheepdog : 88 %
Accuracy of Shetland_sheepdog : 89 %
Accuracy of collie : 75 %
Accuracy of Border_collie : 75 %
Accuracy of Bouvier_des_Flandres : 93 %
Accuracy of Rottweiler : 85 %
Accuracy of German_shepherd : 87 %
Accuracy of Doberman : 76 %
Accuracy of miniature_pinscher : 83 %
Accuracy of Greater_Swiss_Mountain_dog : 87 %
Accuracy of Bernese_mountain_dog : 90 %
Accuracy of Appenzeller : 75 %
Accuracy of EntleBucher : 94 %
Accuracy of boxer : 73 %
Accuracy of bull_mastiff : 80 %
Accuracy of Tibetan_mastiff : 88 %
Accuracy of French_bulldog : 74 %
Accuracy of Great_Dane : 80 %
Accuracy of Saint_Bernard : 96 %
Accuracy of Eskimo_dog : 84 %
Accuracy of malamute : 84 %
Accuracy of Siberian_husky : 79 %
Accuracy of affenpinscher : 87 %
Accuracy of basenji : 95 %
Accuracy of   pug : 98 %
Accuracy of Leonberg : 85 %
Accuracy of Newfoundland : 76 %
Accuracy of Great_Pyrenees : 80 %
Accuracy of Samoyed : 95 %
Accuracy of Pomeranian : 94 %
Accuracy of  chow : 91 %
Accuracy of keeshond : 96 %
Accuracy of Brabancon_griffon : 90 %
Accuracy of Pembroke : 96 %
Accuracy of Cardigan : 51 %
Accuracy of toy_poodle : 87 %
Accuracy of miniature_poodle : 92 %
Accuracy of standard_poodle : 72 %
Accuracy of Mexican_hairless : 100 %
Accuracy of dingo : 66 %
Accuracy of dhole : 88 %
Accuracy of African_hunting_dog : 96 %
```

In [ ]:

```
#with open('/content/drive/MyDrive/Advanced Programming/pred_alexnet1_final', 'wb') as files:
#  pickle.dump(predictions, files)

with open('/content/drive/MyDrive/Advanced Programming/pred_alexnet1' , 'rb') as f:
  predictions = pickle.load(f)
```

AlexNet pretrained - as fixed feature extractor

We were curious to see what are the differences with a pretrained model.

Transfer learning is a machine learning method where a model developed for a task is reused as the starting point for a model on a second task.

It is common to use a pretrained ConvNet on a very large dataset (for us the model is pretrained by PyTorch on ImageNet, which contains 1.2 million images with 1000 categories), and then use that ConvNet as a fixed feature extractor for the task of interest.

One of the major Transfer Learning scenarios is fixed **feature extractor**. Take a ConvNet pretrained on ImageNet, remove the last fully-connected layer (this layer's outputs are the 1000 class scores for a different task like ImageNet), then treat the rest of the ConvNet as a fixed feature extractor for the new dataset.

The second strategy is to not only replace and retrain the classifier on top of the ConvNet on the new dataset, but to also **fine-tune** the weights of the pretrained network by continuing the backpropagation. It is possible to fine-tune all the layers of the ConvNet, or it's possible to keep some of the earlier layers fixed (due to overfitting concerns) and only fine-tune some higher-level portion of the network. This is motivated by the observation that the earlier features of a ConvNet contain more generic features that should be useful to many tasks, but later layers of the ConvNet becomes progressively more specific to the details of the classes contained in the original dataset.

Since the **data is very small** for each category, it is not a good idea to fine-tune the ConvNet due to overfitting concerns. Since the **data is similar to the original data** (our dataset has been built using images and annotation from ImageNet) , we expect higher-level features in the ConvNet to be relevant to this dataset as well. Hence, the best idea might be to have a fixed feature extractor and train a linear classifier on the CNN codes.

(Source Note: the code for using a pretrained model is from this link https://pytorch.org/tutorials/beginner/finetuning_torchvision_models_tutorial.html#helper-functions)

In [ ]:
```
# Flag for feature extracting. When False, we finetune the whole model,
# when True we only update the reshaped layer params
feature_extract = True

AlexNet_model2 = torch.hub.load('pytorch/vision:v0.10.0', 'alexnet', pretrained=True)

#Model description
AlexNet_model2.eval()
```

```
Using cache found in /root/.cache/torch/hub/pytorch_vision_v0.10.0
Downloading: "https://download.pytorch.org/models/alexnet-owt-7be5be79.pth" to /root/.cache/torch/hub/che
ckpoints/alexnet-owt-7be5be79.pth
```

Out[ ]:
```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

Here, we will freeze the weights for all of the network except that of the final fully connected layer. This last layer is replaced with a new one with random weights and only this layer is trained. We need to set *requires_grad = False* to freeze the parameters so that the gradients are not computed in *backward()*. Parameters of newly constructed modules have *requires_grad=True* by default.

In [ ]:
```
def set_parameter_requires_grad(model, feature_extracting):
    if feature_extracting:
        for param in model.parameters():
            param.requires_grad = False

set_parameter_requires_grad(AlexNet_model2, feature_extract)
num_ftrs = AlexNet_model2.classifier[6].in_features
AlexNet_model2.classifier[6] = nn.Linear(num_ftrs, n_classes)
```

In [ ]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)

AlexNet_model2.to(device)
```

```
cuda:0
```

Out[ ]:
```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=120, bias=True)
  )
)
```

In [ ]:

```
#Loss
criterion = nn.CrossEntropyLoss()
```

Here, we will specify to the optimizer to only update the weight of the parameters in the list *parameters_to_update*. In fact, we will train only the final layer of the model.

In [ ]:

```
params_to_update = AlexNet_model2.parameters()
print("Params to learn:")
if feature_extract:
    params_to_update = []
    for name,param in AlexNet_model2.named_parameters():
        if param.requires_grad == True:
            params_to_update.append(param)
            print("\t",name)
else:
    for name,param in AlexNet_model2.named_parameters():
        if param.requires_grad == True:
            print("\t",name)

# Observe that all parameters are being optimized
optimizer = optim.SGD(params_to_update, lr=0.001, momentum=0.9)
```

```
Params to learn:
	 classifier.6.weight
	 classifier.6.bias
```

Training the model

In [ ]:

```
%%time
Early_stopping_value = 0.0001
EPOCHS = 75
train_accu = []
train_loss = []

for epoch in range(1, EPOCHS+1):      # loop over the dataset multiple times
  train(epoch, AlexNet_model2)

  if epoch > 2:
    if abs(train_accu[epoch-2] - train_accu[epoch-1]) < Early_stopping_value:
      print("\nEarly stopping. Epoch:", epoch)
      break

print('\nFinished Training of AlexNet')
```

```
Epoch : 1
[batch: 1] train loss: 5.061, train accuracy: 0.00 %
[batch: 101] train loss: 3.602, train accuracy: 19.93 %
[batch: 201] train loss: 1.280, train accuracy: 27.58 %
[batch: 301] train loss: 0.773, train accuracy: 31.33 %
[batch: 401] train loss: 0.557, train accuracy: 33.67 %
[batch: 501] train loss: 0.444, train accuracy: 35.29 %
[batch: 601] train loss: 0.347, train accuracy: 36.99 %
[batch: 701] train loss: 0.284, train accuracy: 38.47 %
[batch: 801] train loss: 0.272, train accuracy: 39.10 %
[batch: 901] train loss: 0.235, train accuracy: 39.75 %
[batch: 1001] train loss: 0.206, train accuracy: 40.28 %

 [EPOCH 1] train loss: 0.057, train accuracy: 40.39 %

Epoch : 2
[batch: 1] train loss: 2.046, train accuracy: 31.25 %
[batch: 101] train loss: 1.769, train accuracy: 52.54 %
[batch: 201] train loss: 0.916, train accuracy: 51.96 %
[batch: 301] train loss: 0.590, train accuracy: 52.43 %
[batch: 401] train loss: 0.464, train accuracy: 52.15 %
[batch: 501] train loss: 0.366, train accuracy: 51.95 %
[batch: 601] train loss: 0.310, train accuracy: 51.85 %
[batch: 701] train loss: 0.263, train accuracy: 51.76 %
[batch: 801] train loss: 0.237, train accuracy: 51.53 %
[batch: 901] train loss: 0.207, train accuracy: 51.45 %
[batch: 1001] train loss: 0.173, train accuracy: 51.74 %
```

```
 [EPOCH 71] train loss: 0.023, train accuracy: 75.46 %

Epoch : 72
[batch: 1] train loss: 1.368, train accuracy: 81.25 %
[batch: 101] train loss: 0.878, train accuracy: 76.79 %
[batch: 201] train loss: 0.469, train accuracy: 75.62 %
[batch: 301] train loss: 0.284, train accuracy: 75.96 %
[batch: 401] train loss: 0.231, train accuracy: 75.62 %
[batch: 501] train loss: 0.166, train accuracy: 75.89 %
[batch: 601] train loss: 0.146, train accuracy: 75.83 %
[batch: 701] train loss: 0.126, train accuracy: 75.65 %
[batch: 801] train loss: 0.116, train accuracy: 75.67 %
[batch: 901] train loss: 0.088, train accuracy: 75.76 %
[batch: 1001] train loss: 0.089, train accuracy: 75.72 %

 [EPOCH 72] train loss: 0.024, train accuracy: 75.72 %

Epoch : 73
[batch: 1] train loss: 1.491, train accuracy: 56.25 %
[batch: 101] train loss: 0.848, train accuracy: 75.31 %
[batch: 201] train loss: 0.455, train accuracy: 75.56 %
[batch: 301] train loss: 0.264, train accuracy: 76.39 %
[batch: 401] train loss: 0.211, train accuracy: 76.37 %
[batch: 501] train loss: 0.182, train accuracy: 76.01 %
[batch: 601] train loss: 0.133, train accuracy: 76.19 %
[batch: 701] train loss: 0.129, train accuracy: 75.96 %
[batch: 801] train loss: 0.111, train accuracy: 75.85 %
[batch: 901] train loss: 0.098, train accuracy: 75.78 %
[batch: 1001] train loss: 0.094, train accuracy: 75.57 %

 [EPOCH 73] train loss: 0.030, train accuracy: 75.52 %

Epoch : 74
[batch: 1] train loss: 0.453, train accuracy: 87.50 %
[batch: 101] train loss: 0.771, train accuracy: 78.34 %
[batch: 201] train loss: 0.409, train accuracy: 77.15 %
[batch: 301] train loss: 0.275, train accuracy: 76.97 %
[batch: 401] train loss: 0.201, train accuracy: 77.06 %
[batch: 501] train loss: 0.180, train accuracy: 76.87 %
[batch: 601] train loss: 0.148, train accuracy: 76.77 %
[batch: 701] train loss: 0.118, train accuracy: 76.65 %
[batch: 801] train loss: 0.116, train accuracy: 76.33 %
[batch: 901] train loss: 0.096, train accuracy: 76.12 %
[batch: 1001] train loss: 0.091, train accuracy: 76.04 %

 [EPOCH 74] train loss: 0.028, train accuracy: 75.97 %

Epoch : 75
[batch: 1] train loss: 0.405, train accuracy: 75.00 %
[batch: 101] train loss: 0.810, train accuracy: 76.79 %
[batch: 201] train loss: 0.411, train accuracy: 76.52 %
[batch: 301] train loss: 0.296, train accuracy: 76.10 %
[batch: 401] train loss: 0.224, train accuracy: 76.00 %
[batch: 501] train loss: 0.173, train accuracy: 76.01 %
[batch: 601] train loss: 0.148, train accuracy: 75.89 %
[batch: 701] train loss: 0.119, train accuracy: 76.12 %
[batch: 801] train loss: 0.117, train accuracy: 75.69 %
[batch: 901] train loss: 0.100, train accuracy: 75.61 %
[batch: 1001] train loss: 0.096, train accuracy: 75.43 %

 [EPOCH 75] train loss: 0.024, train accuracy: 75.41 %
```

In [ ]:

```python
%%capture
import pickle
#with open('/content/drive/MyDrive/Advanced Programming/alexnet_model2_final', 'wb') as files:
#  pickle.dump(AlexNet_model2, files)

with open('/content/drive/MyDrive/Advanced Programming/alexnet_model2_final' , 'rb') as f:
  AlexNet_model2 = pickle.load(f)
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
AlexNet_model2.to(device)
```

In [ ]:

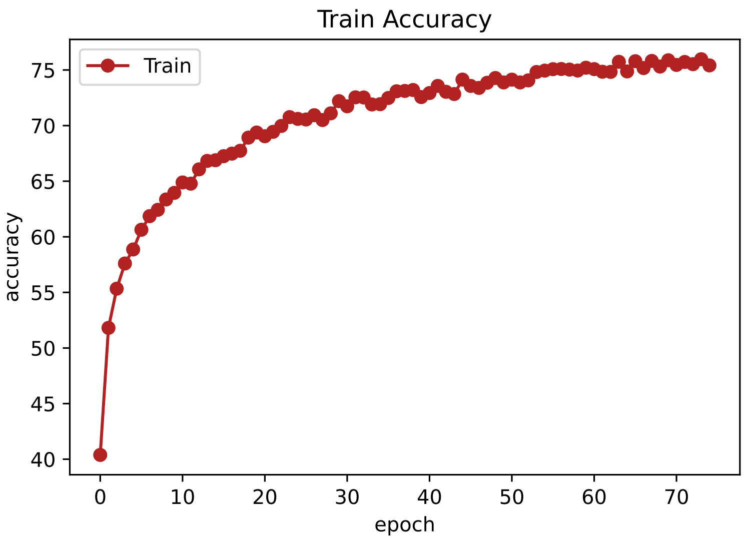
```python
#with open('/content/drive/MyDrive/Advanced Programming/train_accu_alexnet2_final', 'wb') as files:
  #pickle.dump(train_accu, files)

with open('/content/drive/MyDrive/Advanced Programming/train_accu_alexnet2_final' , 'rb') as f:
  train_accu = pickle.load(f)
```

In [ ]:

```python
#with open('/content/drive/MyDrive/Advanced Programming/train_loss_alexnet2_final', 'wb') as files:
 # pickle.dump(train_loss, files)

with open('/content/drive/MyDrive/Advanced Programming/train_loss_alexnet2_final' , 'rb') as f:
  train_loss = pickle.load(f)
```

In [ ]:

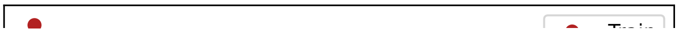
```
import matplotlib.pyplot as plt
from IPython.display import HTML, display, Image

display(HTML("""
<style>
#output-body {
    display: flex;
    align-items: center;
    justify-content: center;
}
</style>
"""))

plt.plot(train_accu,'-o', color = "firebrick")
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.legend(['Train'])
plt.title('Train Accuracy')

plt.show()
```



In [ ]:

```
display(HTML("""
<style>
#output-body {
    display: flex;
    align-items: center;
    justify-content: center;
}
</style>
"""))

plt.plot(train_loss,'-o', color = "firebrick")
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend(['Train'])
plt.title("Train Loss")

plt.show()
```

Train Loss



Testing the model

In [ ]:
```
t = test(AlexNet_model2)
predictions2, acc = t[0], t[1]
print('Accuracy of the network on the test images: %.2f %%' % acc)
```

```
Accuracy of the network on the test images: 82.09 %
```

In [ ]:
```
#Testing classification accuracy for individual classes.
acc = test_accuracy_per_class(AlexNet_model2, name_classes)
for i in range(n_classes):
    print('Accuracy of %5s : %2d %%' %(name_classes[i],acc[i]) )
```

```
Accuracy of Chihuahua : 86 %
Accuracy of Japanese_spaniel : 91 %
Accuracy of Maltese_dog : 86 %
Accuracy of Pekinese : 78 %
Accuracy of Shih_Tzu : 52 %
Accuracy of Blenheim_spaniel : 90 %
Accuracy of papillon : 86 %
Accuracy of toy_terrier : 86 %
Accuracy of Rhodesian_ridgeback : 60 %
Accuracy of Afghan_hound : 84 %
Accuracy of basset : 81 %
Accuracy of beagle : 86 %
Accuracy of bloodhound : 91 %
Accuracy of bluetick : 93 %
Accuracy of black_and_tan_coonhound : 81 %
Accuracy of Walker_hound : 70 %
Accuracy of English_foxhound : 90 %
Accuracy of redbone : 86 %
Accuracy of borzoi : 83 %
Accuracy of Irish_wolfhound : 89 %
Accuracy of Italian_greyhound : 76 %
Accuracy of whippet : 66 %
Accuracy of Ibizan_hound : 81 %
Accuracy of Norwegian_elkhound : 94 %
Accuracy of otterhound : 74 %
Accuracy of Saluki : 70 %
Accuracy of Scottish_deerhound : 80 %
Accuracy of Weimaraner : 92 %
Accuracy of Staffordshire_bullterrier : 91 %
Accuracy of American_Staffordshire_terrier : 75 %
Accuracy of Bedlington_terrier : 93 %
Accuracy of Border_terrier : 81 %
Accuracy of Kerry_blue_terrier : 94 %
Accuracy of Irish_terrier : 72 %
Accuracy of Norfolk_terrier : 75 %
Accuracy of Norwich_terrier : 78 %
Accuracy of Yorkshire_terrier : 86 %
Accuracy of wire_haired_fox_terrier : 90 %
Accuracy of Lakeland_terrier : 83 %
Accuracy of Sealyham_terrier : 91 %
Accuracy of Airedale : 86 %
Accuracy of cairn : 75 %
Accuracy of Australian_terrier : 78 %
Accuracy of Dandie_Dinmont : 68 %
Accuracy of Boston_bull : 93 %
Accuracy of miniature_schnauzer : 93 %
Accuracy of giant_schnauzer : 70 %
Accuracy of standard_schnauzer : 66 %
Accuracy of Scotch_terrier : 82 %
Accuracy of Tibetan_terrier : 72 %
Accuracy of silky_terrier : 83 %
Accuracy of soft_coated_wheaten_terrier : 95 %
Accuracy of West_Highland_white_terrier : 100 %
Accuracy of Lhasa : 82 %
Accuracy of flat_coated_retriever : 66 %
Accuracy of curly_coated_retriever : 80 %
Accuracy of golden_retriever : 81 %
Accuracy of Labrador_retriever : 83 %
Accuracy of Chesapeake_Bay_retriever : 90 %
Accuracy of German_short_haired_pointer : 82 %
Accuracy of vizsla : 73 %
Accuracy of English_setter : 91 %
Accuracy of Irish_setter : 88 %
Accuracy of Gordon_setter : 92 %
Accuracy of Brittany_spaniel : 67 %
Accuracy of clumber : 97 %
Accuracy of English_springer : 88 %
Accuracy of Welsh_springer_spaniel : 80 %
Accuracy of cocker_spaniel : 80 %
Accuracy of Sussex_spaniel : 84 %
Accuracy of Irish_water_spaniel : 91 %
Accuracy of kuvasz : 52 %
Accuracy of schipperke : 90 %
```

```
Accuracy of groenendael : 88 %
Accuracy of malinois : 69 %
Accuracy of briard : 90 %
Accuracy of kelpie : 78 %
Accuracy of komondor : 96 %
Accuracy of Old_English_sheepdog : 94 %
Accuracy of Shetland_sheepdog : 62 %
Accuracy of collie : 87 %
Accuracy of Border_collie : 78 %
Accuracy of Bouvier_des_Flandres : 66 %
Accuracy of Rottweiler : 85 %
Accuracy of German_shepherd : 84 %
Accuracy of Doberman : 69 %
Accuracy of miniature_pinscher : 91 %
Accuracy of Greater_Swiss_Mountain_dog : 84 %
Accuracy of Bernese_mountain_dog : 88 %
Accuracy of Appenzeller : 51 %
Accuracy of EntleBucher : 68 %
Accuracy of boxer : 89 %
Accuracy of bull_mastiff : 83 %
Accuracy of Tibetan_mastiff : 85 %
Accuracy of French_bulldog : 82 %
Accuracy of Great_Dane : 80 %
Accuracy of Saint_Bernard : 93 %
Accuracy of Eskimo_dog : 56 %
Accuracy of malamute : 69 %
Accuracy of Siberian_husky : 87 %
Accuracy of affenpinscher : 90 %
Accuracy of basenji : 91 %
Accuracy of   pug : 86 %
Accuracy of Leonberg : 95 %
Accuracy of Newfoundland : 80 %
Accuracy of Great_Pyrenees : 66 %
Accuracy of Samoyed : 100 %
Accuracy of Pomeranian : 78 %
Accuracy of  chow : 91 %
Accuracy of keeshond : 93 %
Accuracy of Brabancon_griffon : 87 %
Accuracy of Pembroke : 86 %
Accuracy of Cardigan : 81 %
Accuracy of toy_poodle : 70 %
Accuracy of miniature_poodle : 70 %
Accuracy of standard_poodle : 51 %
Accuracy of Mexican_hairless : 90 %
Accuracy of dingo : 71 %
Accuracy of dhole : 80 %
Accuracy of African_hunting_dog : 96 %
```

In [ ]:

```
#with open('/content/drive/MyDrive/Advanced Programming/pred_alexnet2_final', 'wb') as files:
#  pickle.dump(predictions2, files)

with open('/content/drive/MyDrive/Advanced Programming/pred_alexnet2_final' , 'rb') as f:
  predictions2 = pickle.load(f)
```

An example

Let's have a look at an example. We load two random images and see if our model is able to predict the breeds in the pictures.

In [ ]:

```
imageA, labelA = test_dataset[20][0], test_dataset[20][1]
predA1 = predictions[20]
predA2 = predictions2[20]
imageB, labelB = test_dataset[60][0], test_dataset[60][1]
predB1 = predictions[60]
predB2 = predictions2[60]
```

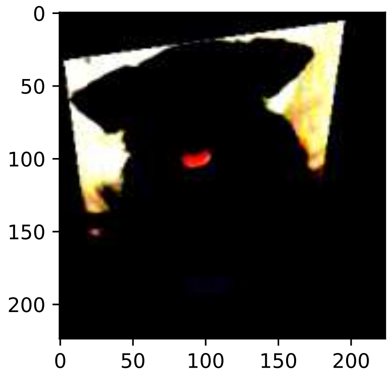
In [ ]:

```
imshow(imageA, single = True)
print('Correct label:', name_classes[labelA],"\n Predicted label model 1:", name_classes[predA1],
      "\n Predicted label model 2:", name_classes[predA2])
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for intege
rs).
```


```
In [ ]: imshow(imageB, single=True)
        print('Correct label:', name_classes[labelB],"\n Predicted label model 1:",  name_classes[predB1],
              "\n Predicted label model 2:",  name_classes[predB2])
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```

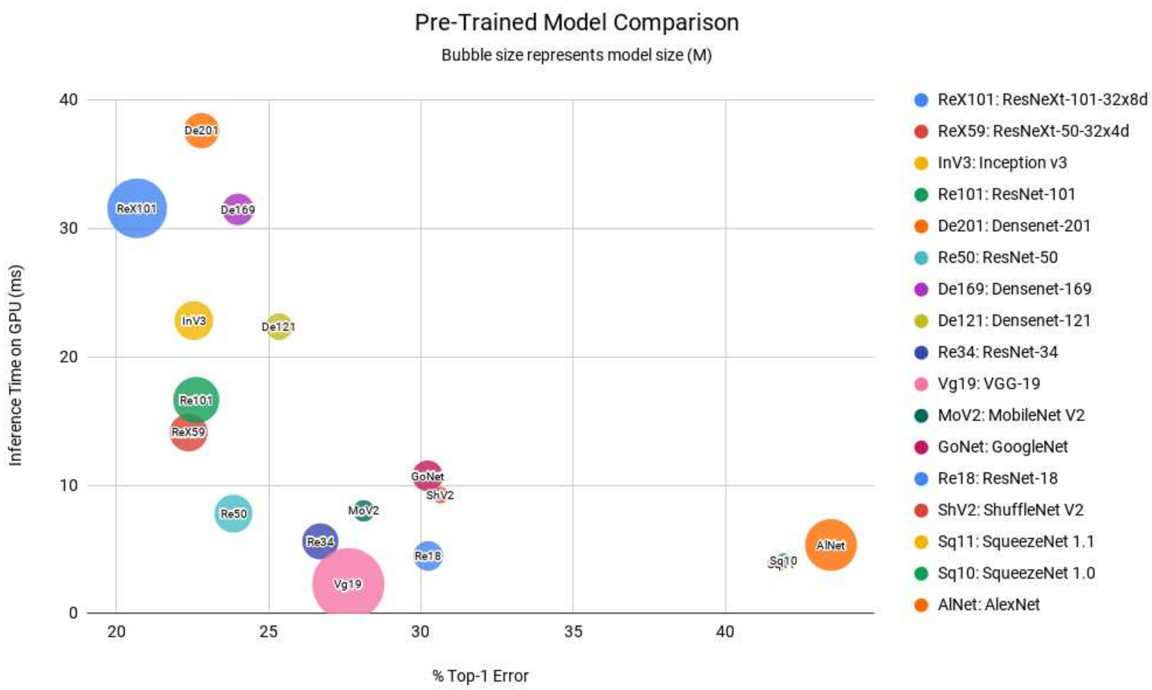



```
Correct label: standard_schnauzer
 Predicted label model 1: giant_schnauzer
 Predicted label model 2: affenpinscher
```

ResNet 50 - not pretrained

As our second model we decided to use **ResNet 50**. The reason for our choice is that between the pretrained PyTorch models, ResNet 50 has some of the best characteristics, as we can see in the following graph.

(Source of the picture: https://learnopencv.com/pytorch-for-beginners-image-classification-using-pre-trained-models/)



Here we can see that ResNet 50 is great because it has the following features:

- small inference time on GPU
- small top-1 error (A top-1 error occurs if the class predicted by a model with highest confidence is not the same as the true class).

AlexNet is one of the first model created, most recent model are improved version and have lower Top-1 Error. For this reason we expect greater results from ResNet 50 compared to AlexNet.

ResNet50 is a variant of ResNet model which has 48 Convolution layers along with 1 MaxPool and 1 Average Pool layer.

First we gave in input the images having height, width as 224x224 and 3 as channel width. The first layer we have is a convolution layer using

7x7 kernel size followed by a max pooling layer with 3x3 kernel size. Then we can divide the architecture of Resnet50 in 4 stages.

Stage 1 of the network starts and it has 3 Residual blocks containing 3 layers each. The size of kernels used to perform the convolution operation in all 3 layers of the block of stage 1 are 64, 64 and 128 respectively.

As we progress from one stage to another, the channel width is doubled and the size of the input is reduced to half. For each residual function, 3 layers are stacked one over the other. The three layers are 1×1, 3×3, 1×1 convolutions. The 1×1 convolution layers are responsible for reducing and then restoring the dimensions. The 3×3 layer is left as a bottleneck with smaller input/output dimensions.

Finally, the network has an Average Pooling layer followed by a fully connected layer having 120 neurons, the number of classes (ImageNet class output).

We import ResNet 50 from the Pytorch website.

In [48]:
```
ResNet_model = torch.hub.load('pytorch/vision:v0.10.0', 'resnet50', pretrained=False)

#Model description
ResNet_model.eval()
```

```
Downloading: "https://github.com/pytorch/vision/archive/v0.10.0.zip" to /root/.cache/torch/hub/v0.10.0.zi
p
```

Out[48]:
```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (layer2): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (3): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (layer3): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (3): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (4): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (5): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (layer4): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
        (relu): ReLU(inplace=True)
      )
      (2): Bottleneck(
        (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
      )
    )
    (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
    (fc): Linear(in_features=2048, out_features=1000, bias=True)
  )
```

In order to have a model for classification on 120 classes, we changed the last linear layer of ResNet50 to a layer with an output size equal to 120.

In [49]:
```
ResNet_model.fc = nn.Linear(2048, n_classes)
```

We now verify the device where the model is running and we move the input and ResNet_model to GPU for speed, if available.

In [50]:
```
import torch
torch.cuda.empty_cache()

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)

ResNet_model.to(device)
```

```
cuda:0
```

Out[50]:
```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (layer2): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (3): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (layer3): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (3): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (4): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (5): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (layer4): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
```

```
    )
    (1): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=2048, out_features=120, bias=True)
)
```

As loss function we use the Cross Entropy function, and as optimizer we use stochastic gradient descent.

In [51]:
```
#Loss
criterion = nn.CrossEntropyLoss()

#Optimizer(SGD)
optimizer = optim.SGD(ResNet_model.parameters(), lr=0.001, momentum=0.9)
```

Training the model

Let's train our model with the previously defined function. We set the number of epochs and a early stopping value to stop our model in case there is not great improvement in the accuracy.

In [52]:
```
%%time

Early_stopping_value = 0.00001
EPOCHS = 100
train_accu = []
train_loss = []

for epoch in range(1, EPOCHS + 1):     # loop over the dataset multiple times
  train(epoch, ResNet_model)
  if epoch > 2:
    if abs(train_accu[epoch-2] - train_accu[epoch-1]) < Early_stopping_value:
      print("\nEarly stopping. Epoch:", epoch)
      break

print('\nFinished Training of my model')
```

```
Epoch : 1
[batch: 1] train loss: 5.044, train accuracy: 0.00 %
[batch: 101] train loss: 5.029, train accuracy: 0.93 %
[batch: 201] train loss: 2.538, train accuracy: 0.87 %
[batch: 301] train loss: 1.688, train accuracy: 0.93 %
[batch: 401] train loss: 1.252, train accuracy: 0.94 %
[batch: 501] train loss: 0.995, train accuracy: 0.99 %
[batch: 601] train loss: 0.831, train accuracy: 1.02 %
[batch: 701] train loss: 0.699, train accuracy: 1.05 %
[batch: 801] train loss: 0.612, train accuracy: 1.18 %
[batch: 901] train loss: 0.541, train accuracy: 1.17 %
[batch: 1001] train loss: 0.480, train accuracy: 1.22 %

 [EPOCH 1] train loss: 0.132, train accuracy: 1.23 %

Epoch : 2
[batch: 1] train loss: 5.169, train accuracy: 6.25 %
[batch: 101] train loss: 4.708, train accuracy: 2.23 %
[batch: 201] train loss: 2.360, train accuracy: 2.05 %
[batch: 301] train loss: 1.563, train accuracy: 2.49 %
[batch: 401] train loss: 1.168, train accuracy: 2.34 %
[batch: 501] train loss: 0.932, train accuracy: 2.38 %
[batch: 601] train loss: 0.762, train accuracy: 2.27 %
[batch: 701] train loss: 0.658, train accuracy: 2.31 %
[batch: 801] train loss: 0.574, train accuracy: 2.36 %
[batch: 901] train loss: 0.511, train accuracy: 2.39 %
[batch: 1001] train loss: 0.456, train accuracy: 2.42 %

 [EPOCH 2] train loss: 0.123, train accuracy: 2.45 %

Epoch : 3
[batch: 1] train loss: 4.252, train accuracy: 0.00 %
[batch: 101] train loss: 4.467, train accuracy: 3.77 %
[batch: 201] train loss: 2.238, train accuracy: 3.61 %
[batch: 301] train loss: 1.487, train accuracy: 3.43 %
```

```
[batch: 401] train loss: 0.067, train accuracy: 92.14 %
[batch: 501] train loss: 0.045, train accuracy: 92.18 %
[batch: 601] train loss: 0.043, train accuracy: 92.26 %
[batch: 701] train loss: 0.037, train accuracy: 92.11 %
[batch: 801] train loss: 0.032, train accuracy: 91.99 %
[batch: 901] train loss: 0.030, train accuracy: 91.90 %
[batch: 1001] train loss: 0.026, train accuracy: 91.85 %

 [EPOCH 97] train loss: 0.007, train accuracy: 91.82 %

Epoch : 98
[batch: 1] train loss: 0.056, train accuracy: 100.00 %
[batch: 101] train loss: 0.229, train accuracy: 92.88 %
[batch: 201] train loss: 0.116, train accuracy: 92.72 %
[batch: 301] train loss: 0.081, train accuracy: 92.63 %
[batch: 401] train loss: 0.056, train accuracy: 92.69 %
[batch: 501] train loss: 0.049, train accuracy: 92.54 %
[batch: 601] train loss: 0.039, train accuracy: 92.65 %
[batch: 701] train loss: 0.031, train accuracy: 92.64 %
[batch: 801] train loss: 0.034, train accuracy: 92.52 %
[batch: 901] train loss: 0.027, train accuracy: 92.52 %
[batch: 1001] train loss: 0.022, train accuracy: 92.61 %

 [EPOCH 98] train loss: 0.006, train accuracy: 92.58 %

Epoch : 99
[batch: 1] train loss: 0.312, train accuracy: 93.75 %
[batch: 101] train loss: 0.198, train accuracy: 94.18 %
[batch: 201] train loss: 0.099, train accuracy: 94.00 %
[batch: 301] train loss: 0.068, train accuracy: 93.81 %
[batch: 401] train loss: 0.044, train accuracy: 94.01 %
[batch: 501] train loss: 0.042, train accuracy: 93.80 %
[batch: 601] train loss: 0.039, train accuracy: 93.63 %
[batch: 701] train loss: 0.032, train accuracy: 93.50 %
[batch: 801] train loss: 0.027, train accuracy: 93.45 %
[batch: 901] train loss: 0.026, train accuracy: 93.40 %
[batch: 1001] train loss: 0.025, train accuracy: 93.31 %

 [EPOCH 99] train loss: 0.006, train accuracy: 93.31 %

Epoch : 100
[batch: 1] train loss: 0.217, train accuracy: 93.75 %
[batch: 101] train loss: 0.205, train accuracy: 93.69 %
[batch: 201] train loss: 0.104, train accuracy: 93.10 %
[batch: 301] train loss: 0.080, train accuracy: 92.82 %
[batch: 401] train loss: 0.053, train accuracy: 92.99 %
[batch: 501] train loss: 0.040, train accuracy: 93.15 %
[batch: 601] train loss: 0.039, train accuracy: 93.02 %
[batch: 701] train loss: 0.038, train accuracy: 92.78 %
[batch: 801] train loss: 0.034, train accuracy: 92.63 %
[batch: 901] train loss: 0.026, train accuracy: 92.61 %
[batch: 1001] train loss: 0.024, train accuracy: 92.59 %

 [EPOCH 100] train loss: 0.007, train accuracy: 92.58 %
```

Let's save our model

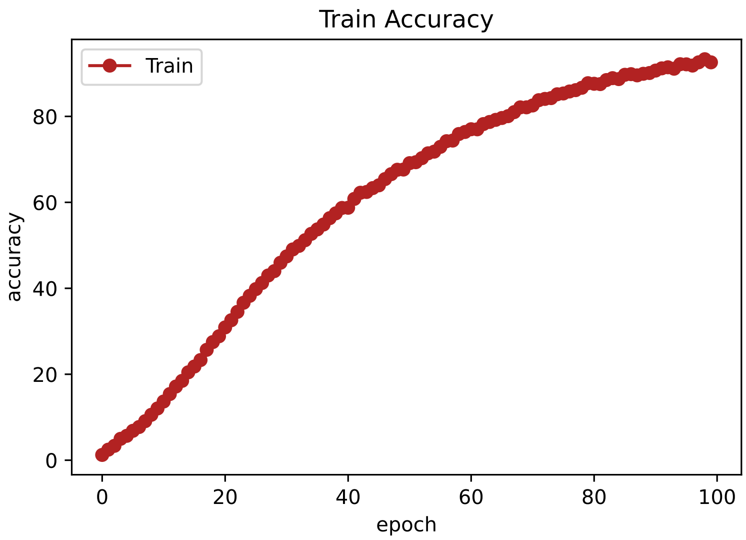
In [45]:
```
%%capture
import pickle
#with open('/content/drive/MyDrive/Advanced Programming/resnet_model1_final', 'wb') as files:
#  pickle.dump(ResNet_model, files)

with open('/content/drive/MyDrive/Advanced Programming/resnet_model1_final' , 'rb') as f:
  ResNet_model = pickle.load(f)

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
ResNet_model.to(device)
```

In [46]:
```
#with open('/content/drive/MyDrive/Advanced Programming/train_accu_resnet1_final', 'wb') as files:
#  pickle.dump(train_accu, files)

with open('/content/drive/MyDrive/Advanced Programming/train_accu_resnet1_final' , 'rb') as f:
  train_accu = pickle.load(f)
```

In [47]:
```
#with open('/content/drive/MyDrive/Advanced Programming/train_loss_resnet1_final', 'wb') as files:
#  pickle.dump(train_loss, files)

with open('/content/drive/MyDrive/Advanced Programming/train_loss_resnet1_final' , 'rb') as f:
  train_loss = pickle.load(f)
```

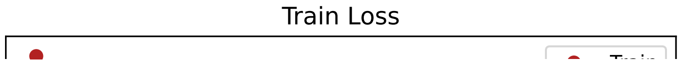
In [48]:

```
import matplotlib.pyplot as plt
from IPython.display import HTML, display, Image

display(HTML("""
<style>
#output-body {
    display: flex;
    align-items: center;
    justify-content: center;
}
</style>
"""))

plt.plot(train_accu,'-o', color = "firebrick")
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.legend(['Train'])
plt.title('Train Accuracy')

plt.show()
```



In [49]:

```
display(HTML("""
<style>
#output-body {
    display: flex;
    align-items: center;
    justify-content: center;
}
</style>
"""))

plt.plot(train_loss,'-o', color = "firebrick")
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend(['Train'])
plt.title("Train Loss")

plt.show()
```



Testing the model

Let's compute the total accuracy of the model and the accuracy for each class.

In [50]:
```
predictions = []
t = test(ResNet_model)
predictions, acc = t[0], t[1]
print('Accuracy of the network on the test images: %.2f %%' % acc)
```

```
Accuracy of the network on the test images: 87.83 %
```

In [51]:
```
#Testing classification accuracy for individual classes.

acc = test_accuracy_per_class(ResNet_model, name_classes)
for i in range(n_classes):
    print('Accuracy of %5s : %.2f %%' %(name_classes[i],acc[i]) )
```

```
Accuracy of Chihuahua : 92.86 %
Accuracy of Japanese_spaniel : 96.55 %
Accuracy of Maltese_dog : 86.00 %
Accuracy of Pekinese : 86.21 %
Accuracy of Shih_Tzu : 77.59 %
Accuracy of Blenheim_spaniel : 90.32 %
Accuracy of papillon : 86.67 %
Accuracy of toy_terrier : 81.82 %
Accuracy of Rhodesian_ridgeback : 85.71 %
Accuracy of Afghan_hound : 97.62 %
Accuracy of basset : 86.84 %
Accuracy of beagle : 94.12 %
Accuracy of bloodhound : 78.95 %
Accuracy of bluetick : 97.73 %
Accuracy of black_and_tan_coonhound : 89.29 %
Accuracy of Walker_hound : 83.33 %
Accuracy of English_foxhound : 90.32 %
Accuracy of redbone : 78.79 %
Accuracy of borzoi : 96.97 %
Accuracy of Irish_wolfhound : 82.35 %
Accuracy of Italian_greyhound : 94.44 %
Accuracy of whippet : 94.29 %
Accuracy of Ibizan_hound : 95.00 %
Accuracy of Norwegian_elkhound : 91.18 %
Accuracy of otterhound : 86.49 %
Accuracy of Saluki : 94.74 %
Accuracy of Scottish_deerhound : 96.00 %
Accuracy of Weimaraner : 96.43 %
Accuracy of Staffordshire_bullterrier : 74.07 %
Accuracy of American_Staffordshire_terrier : 85.00 %
Accuracy of Bedlington_terrier : 94.74 %
Accuracy of Border_terrier : 90.32 %
Accuracy of Kerry_blue_terrier : 94.87 %
Accuracy of Irish_terrier : 88.00 %
Accuracy of Norfolk_terrier : 94.74 %
Accuracy of Norwich_terrier : 89.47 %
Accuracy of Yorkshire_terrier : 84.38 %
Accuracy of wire_haired_fox_terrier : 85.19 %
Accuracy of Lakeland_terrier : 86.49 %
Accuracy of Sealyham_terrier : 95.83 %
Accuracy of Airedale : 92.68 %
Accuracy of cairn : 73.53 %
Accuracy of Australian_terrier : 87.18 %
Accuracy of Dandie_Dinmont : 85.71 %
Accuracy of Boston_bull : 85.71 %
Accuracy of miniature_schnauzer : 76.00 %
Accuracy of giant_schnauzer : 78.57 %
Accuracy of standard_schnauzer : 84.21 %
Accuracy of Scotch_terrier : 88.89 %
Accuracy of Tibetan_terrier : 90.00 %
Accuracy of silky_terrier : 87.88 %
Accuracy of soft_coated_wheaten_terrier : 91.67 %
Accuracy of West_Highland_white_terrier : 100.00 %
Accuracy of Lhasa : 87.50 %
Accuracy of flat_coated_retriever : 85.71 %
Accuracy of curly_coated_retriever : 88.89 %
Accuracy of golden_retriever : 88.57 %
Accuracy of Labrador_retriever : 75.00 %
Accuracy of Chesapeake_Bay_retriever : 88.46 %
Accuracy of German_short_haired_pointer : 80.77 %
Accuracy of vizsla : 91.67 %
Accuracy of English_setter : 80.00 %
Accuracy of Irish_setter : 87.10 %
Accuracy of Gordon_setter : 93.75 %
Accuracy of Brittany_spaniel : 87.88 %
Accuracy of clumber : 83.33 %
Accuracy of English_springer : 87.50 %
Accuracy of Welsh_springer_spaniel : 96.77 %
```

```
Accuracy of cocker_spaniel : 84.85 %
Accuracy of Sussex_spaniel : 87.50 %
Accuracy of Irish_water_spaniel : 100.00 %
Accuracy of kuvasz : 72.41 %
Accuracy of schipperke : 85.00 %
Accuracy of groenendael : 86.36 %
Accuracy of malinois : 89.66 %
Accuracy of briard : 93.55 %
Accuracy of kelpie : 85.71 %
Accuracy of komondor : 91.67 %
Accuracy of Old_English_sheepdog : 100.00 %
Accuracy of Shetland_sheepdog : 88.00 %
Accuracy of collie : 85.71 %
Accuracy of Border_collie : 100.00 %
Accuracy of Bouvier_des_Flandres : 85.71 %
Accuracy of Rottweiler : 79.31 %
Accuracy of German_shepherd : 96.43 %
Accuracy of Doberman : 71.43 %
Accuracy of miniature_pinscher : 74.19 %
Accuracy of Greater_Swiss_Mountain_dog : 93.02 %
Accuracy of Bernese_mountain_dog : 86.05 %
Accuracy of Appenzeller : 76.47 %
Accuracy of EntleBucher : 95.56 %
Accuracy of boxer : 75.76 %
Accuracy of bull_mastiff : 93.10 %
Accuracy of Tibetan_mastiff : 90.62 %
Accuracy of French_bulldog : 82.05 %
Accuracy of Great_Dane : 77.42 %
Accuracy of Saint_Bernard : 96.43 %
Accuracy of Eskimo_dog : 71.05 %
Accuracy of malamute : 80.65 %
Accuracy of Siberian_husky : 86.05 %
Accuracy of affenpinscher : 87.50 %
Accuracy of basenji : 97.44 %
Accuracy of   pug : 87.50 %
Accuracy of Leonberg : 93.75 %
Accuracy of Newfoundland : 88.89 %
Accuracy of Great_Pyrenees : 94.87 %
Accuracy of Samoyed : 90.48 %
Accuracy of Pomeranian : 90.91 %
Accuracy of  chow : 97.22 %
Accuracy of keeshond : 96.00 %
Accuracy of Brabancon_griffon : 100.00 %
Accuracy of Pembroke : 97.37 %
Accuracy of Cardigan : 82.76 %
Accuracy of toy_poodle : 83.87 %
Accuracy of miniature_poodle : 87.18 %
Accuracy of standard_poodle : 73.53 %
Accuracy of Mexican_hairless : 96.30 %
Accuracy of dingo : 81.58 %
Accuracy of dhole : 96.30 %
```

We save also the predictions

In [52]:
```
#with open('/content/drive/MyDrive/Advanced Programming/pred_resnet1_final', 'wb') as files:
#  pickle.dump(predictions, files)

with open('/content/drive/MyDrive/Advanced Programming/pred_resnet1' , 'rb') as f:
  predictions = pickle.load(f)
```

ResNet 50 pretrained - as fixed feature extractor

Now we try to improve the results with a fixed feature extractor from the pretrained model of ResNet50. For the same reasons stated for pretrained AlexNet as a fixed feature extractor, we expect greater accuracy.

In [ ]:
```
# Flag for feature extracting. When False, we finetune the whole model,
#   when True we only update the reshaped layer params
feature_extract = True

ResNet_model2 = torch.hub.load('pytorch/vision:v0.10.0', 'resnet50', pretrained=True)

#Model description
ResNet_model2.eval()
```

```
Using cache found in /root/.cache/torch/hub/pytorch_vision_v0.10.0
Downloading: "https://download.pytorch.org/models/resnet50-0676ba61.pth" to /root/.cache/torch/hub/checkp
oints/resnet50-0676ba61.pth
```

Out[ ]:
```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (layer2): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (3): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (layer3): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
```

```
      (2): Bottleneck(
        (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
      )
      (3): Bottleneck(
        (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
      )
      (4): Bottleneck(
        (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
      )
      (5): Bottleneck(
        (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
      )
    )
    (layer4): Sequential(
      (0): Bottleneck(
        (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (downsample): Sequential(
          (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
          (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
      )
      (1): Bottleneck(
        (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
      )
      (2): Bottleneck(
        (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
      )
    )
    (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
    (fc): Linear(in_features=2048, out_features=1000, bias=True)
  )
```

In [ ]:
```python
def set_parameter_requires_grad(model, feature_extracting):
    if feature_extracting:
        for param in model.parameters():
            param.requires_grad = False

set_parameter_requires_grad(ResNet_model2, feature_extract)
ResNet_model2.fc = nn.Linear(2048, n_classes)
```

In [ ]:
```python
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)

ResNet_model2.to(device)
```

```
cpu
```

Out[ ]:
```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
```

```
(maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
(layer1): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (downsample): Sequential(
      (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (2): Bottleneck(
    (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
)
(layer2): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (downsample): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (2): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (3): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
)
(layer3): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (downsample): Sequential(
      (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
```

```
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (3): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (4): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (5): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (layer4): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=2048, out_features=120, bias=True)
```

In [ ]:

```
#Loss
criterion = nn.CrossEntropyLoss()
```

In [ ]:
```
params_to_update = ResNet_model2.parameters()
print("Params to learn:")
if feature_extract:
    params_to_update = []
    for name,param in ResNet_model2.named_parameters():
        if param.requires_grad == True:
            params_to_update.append(param)
            print("\t",name)
else:
    for name,param in ResNet_model2.named_parameters():
        if param.requires_grad == True:
            print("\t",name)

# Observe that all parameters are being optimized
optimizer = optim.SGD(params_to_update, lr=0.001, momentum=0.9)
```

```
Params to learn:
	 fc.weight
	 fc.bias
```

Training the model

In [ ]:
```
%%time
Early_stopping_value = 0.00001
EPOCHS = 75
train_accu = []
train_loss = []

for epoch in range(1, EPOCHS+1):      # loop over the dataset multiple times
  train(epoch, ResNet_model2)

  if epoch > 2:
    if abs(train_accu[epoch-2] - train_accu[epoch-1]) < Early_stopping_value:
      print("\nEarly stopping. Epoch:", epoch)
      break

print('\nFinished Training of ResNet')
```

```
Epoch : 1
[batch: 1] train loss: 5.112, train accuracy: 0.00 %
[batch: 101] train loss: 4.587, train accuracy: 5.75 %
[batch: 201] train loss: 2.040, train accuracy: 13.31 %
[batch: 301] train loss: 1.214, train accuracy: 19.93 %
[batch: 401] train loss: 0.809, train accuracy: 25.53 %
[batch: 501] train loss: 0.578, train accuracy: 30.39 %
[batch: 601] train loss: 0.432, train accuracy: 34.46 %
[batch: 701] train loss: 0.340, train accuracy: 37.90 %
[batch: 801] train loss: 0.267, train accuracy: 41.25 %
[batch: 901] train loss: 0.227, train accuracy: 43.82 %
[batch: 1001] train loss: 0.189, train accuracy: 46.04 %

 [EPOCH 1] train loss: 0.050, train accuracy: 46.54 %

Epoch : 2
[batch: 1] train loss: 1.772, train accuracy: 75.00 %
[batch: 101] train loss: 1.671, train accuracy: 69.62 %
[batch: 201] train loss: 0.814, train accuracy: 69.53 %
[batch: 301] train loss: 0.511, train accuracy: 69.85 %
[batch: 401] train loss: 0.383, train accuracy: 69.75 %
[batch: 501] train loss: 0.288, train accuracy: 69.82 %
[batch: 601] train loss: 0.229, train accuracy: 70.21 %
[batch: 701] train loss: 0.187, train accuracy: 70.65 %
[batch: 801] train loss: 0.166, train accuracy: 70.92 %
[batch: 901] train loss: 0.141, train accuracy: 71.09 %
[batch: 1001] train loss: 0.122, train accuracy: 71.23 %

 [EPOCH 2] train loss: 0.036, train accuracy: 71.24 %

Epoch : 3
[batch: 1] train loss: 0.605, train accuracy: 100.00 %
[batch: 101] train loss: 1.152, train accuracy: 75.87 %
[batch: 201] train loss: 0.576, train accuracy: 75.62 %
[batch: 301] train loss: 0.380, train accuracy: 75.56 %
[batch: 401] train loss: 0.281, train accuracy: 75.36 %
[batch: 501] train loss: 0.220, train accuracy: 75.31 %
[batch: 601] train loss: 0.183, train accuracy: 75.09 %
[batch: 701] train loss: 0.155, train accuracy: 74.97 %
[batch: 801] train loss: 0.135, train accuracy: 75.05 %
[batch: 901] train loss: 0.113, train accuracy: 75.06 %
[batch: 1001] train loss: 0.105, train accuracy: 74.99 %

 [EPOCH 3] train loss: 0.028, train accuracy: 75.04 %

Epoch : 4
[batch: 1] train loss: 0.876, train accuracy: 75.00 %
[batch: 101] train loss: 1.003, train accuracy: 76.05 %
[batch: 201] train loss: 0.483, train accuracy: 76.74 %
[batch: 301] train loss: 0.327, train accuracy: 76.50 %
```

```
[batch: 301] train loss: 0.172, train accuracy: 84.39 %
[batch: 401] train loss: 0.118, train accuracy: 84.71 %
[batch: 501] train loss: 0.104, train accuracy: 84.42 %
[batch: 601] train loss: 0.082, train accuracy: 84.53 %
[batch: 701] train loss: 0.074, train accuracy: 84.55 %
[batch: 801] train loss: 0.064, train accuracy: 84.64 %
[batch: 901] train loss: 0.052, train accuracy: 84.71 %
[batch: 1001] train loss: 0.046, train accuracy: 84.85 %

 [EPOCH 73] train loss: 0.014, train accuracy: 84.88 %

Epoch : 74
[batch: 1] train loss: 0.197, train accuracy: 100.00 %
[batch: 101] train loss: 0.499, train accuracy: 83.60 %
[batch: 201] train loss: 0.244, train accuracy: 84.33 %
[batch: 301] train loss: 0.173, train accuracy: 84.14 %
[batch: 401] train loss: 0.125, train accuracy: 84.20 %
[batch: 501] train loss: 0.098, train accuracy: 84.22 %
[batch: 601] train loss: 0.086, train accuracy: 84.28 %
[batch: 701] train loss: 0.071, train accuracy: 84.28 %
[batch: 801] train loss: 0.071, train accuracy: 84.13 %
[batch: 901] train loss: 0.054, train accuracy: 84.29 %
[batch: 1001] train loss: 0.050, train accuracy: 84.23 %

 [EPOCH 74] train loss: 0.016, train accuracy: 84.13 %

Epoch : 75
[batch: 1] train loss: 1.028, train accuracy: 75.00 %
[batch: 101] train loss: 0.466, train accuracy: 85.71 %
[batch: 201] train loss: 0.253, train accuracy: 85.07 %
[batch: 301] train loss: 0.169, train accuracy: 85.09 %
[batch: 401] train loss: 0.129, train accuracy: 84.65 %
[batch: 501] train loss: 0.094, train accuracy: 84.73 %
[batch: 601] train loss: 0.091, train accuracy: 84.36 %
[batch: 701] train loss: 0.074, train accuracy: 84.36 %
[batch: 801] train loss: 0.061, train accuracy: 84.44 %
[batch: 901] train loss: 0.053, train accuracy: 84.57 %
[batch: 1001] train loss: 0.055, train accuracy: 84.41 %

 [EPOCH 75] train loss: 0.016, train accuracy: 84.35 %
```

In [ ]:

```
%%capture
import pickle
#with open('/content/drive/MyDrive/Advanced Programming/resnet_model2_final', 'wb') as files:
#  pickle.dump(ResNet_model2, files)

with open('/content/drive/MyDrive/Advanced Programming/resnet_model2_final' , 'rb') as f:
  ResNet_model2 = pickle.load(f)
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
ResNet_model2.to(device)
```

In [ ]:

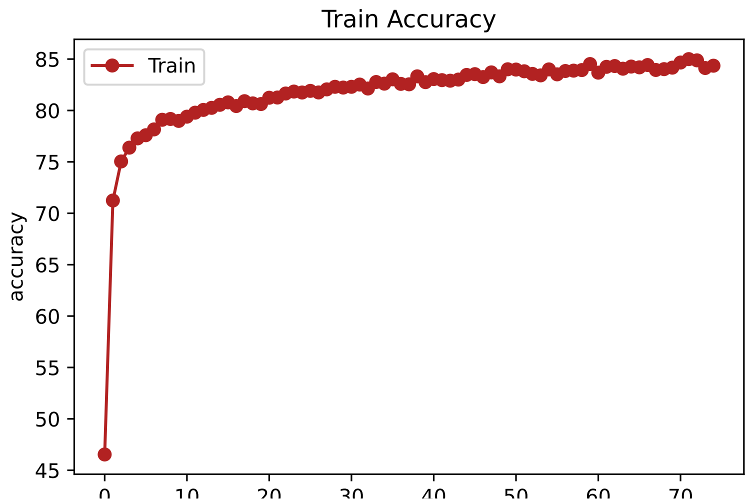
```
#with open('/content/drive/MyDrive/Advanced Programming/train_accu_resnet2_final', 'wb') as files:
#  pickle.dump(train_accu, files)

with open('/content/drive/MyDrive/Advanced Programming/train_accu_resnet2_final' , 'rb') as f:
  train_accu = pickle.load(f)
```

In [ ]:

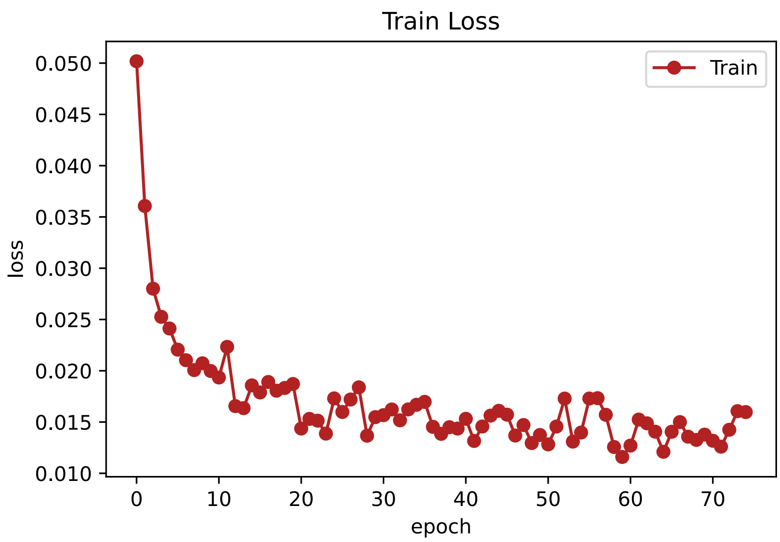
```
#with open('/content/drive/MyDrive/Advanced Programming/train_loss_resnet2_final', 'wb') as files:
#  pickle.dump(train_loss, files)

with open('/content/drive/MyDrive/Advanced Programming/train_loss_resnet2_final' , 'rb') as f:
  train_loss = pickle.load(f)
```

In [ ]:

```
import matplotlib.pyplot as plt
from IPython.display import HTML, display, Image

display(HTML("""
<style>
#output-body {
    display: flex;
    align-items: center;
    justify-content: center;
}
</style>
"""))

plt.plot(train_accu,'-o', color = "firebrick")
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.legend(['Train'])
plt.title('Train Accuracy')

plt.show()
```



In [ ]:
```
display(HTML("""
<style>
#output-body {
    display: flex;
    align-items: center;
    justify-content: center;
}
</style>
"""))

plt.plot(train_loss,'-o',color = "firebrick")
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend(['Train'])
plt.title("Train Loss")

plt.show()
```



Testing the model

In [ ]:
```
predictions2 = []
t = test(ResNet_model2)
predictions2, acc = t[0], t[1]
print('Accuracy of the network on the test images: %.2f %%' % acc)
```

```
Accuracy of the network on the test images: 87.27 %
```

In [ ]:
```
#Testing classification accuracy for individual classes.
acc = test_accuracy_per_class(ResNet_model2, name_classes)
for i in range(n_classes):
    print('Accuracy of %5s : %2d %%' %(name_classes[i],acc[i]) )
```

```
Accuracy of Chihuahua : 83 %
Accuracy of Japanese_spaniel : 86 %
Accuracy of Maltese_dog : 93 %
Accuracy of Pekinese : 100 %
Accuracy of Shih_Tzu : 80 %
```

```
Accuracy of Siberian_husky : 91 %
Accuracy of affenpinscher : 93 %
Accuracy of basenji : 93 %
Accuracy of   pug : 98 %
Accuracy of Leonberg : 97 %
Accuracy of Newfoundland : 88 %
Accuracy of Great_Pyrenees : 94 %
Accuracy of Samoyed : 95 %
Accuracy of Pomeranian : 94 %
Accuracy of  chow : 94 %
Accuracy of keeshond : 100 %
Accuracy of Brabancon_griffon : 90 %
Accuracy of Pembroke : 94 %
Accuracy of Cardigan : 88 %
Accuracy of toy_poodle : 67 %
Accuracy of miniature_poodle : 62 %
Accuracy of standard_poodle : 81 %
Accuracy of Mexican_hairless : 96 %
Accuracy of dingo : 95 %
Accuracy of dhole : 88 %
```

In [53]:
```
#with open('/content/drive/MyDrive/Advanced Programming/pred_resnet2_final', 'wb') as files:
#  pickle.dump(predictions2, files)

with open('/content/drive/MyDrive/Advanced Programming/pred_resnet2_final' , 'rb') as f:
  predictions2 = pickle.load(f)
```

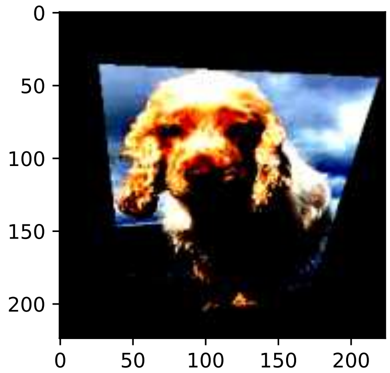
An example

Let's have a look at an example. We load two random images and see if our model is able to predict the breeds in the pictures.

In [59]:
```
imageA, labelA = test_dataset[1200][0], test_dataset[1200][1]
predA1 = predictions[1200]
predA2 = predictions2[1200]
imageB, labelB = test_dataset[800][0], test_dataset[800][1]
predB1 = predictions[800]
predB2 = predictions2[800]
```

In [55]:
```
imshow(imageA, single = True)
print('Correct label:', name_classes[labelA],"\n Predicted label model 1:", name_classes[predA1],
      "\n Predicted label model 2:", name_classes[predA2])
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for intege
rs).
```




```
Correct label: cocker_spaniel
 Predicted label model 1: cocker_spaniel
 Predicted label model 2: Appenzeller
```

In [60]:
```
 imshow(imageB, single=True)
print('Correct label:', name_classes[labelB],"\n Predicted label model 1:",  name_classes[predB1],
      "\n Predicted label model 2:",  name_classes[predB2])
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for intege
rs).
```

Our puppies

Let's see if our model can predict the breeds of our puppies.

In [ ]:
```
margot0 = cv2.imread('/content/drive/MyDrive/Advanced Programming/Dati dogs/margot.jpg')
margot1 = cv2.resize(margot0, dsize=(224,224),interpolation=cv2.INTER_CUBIC)
margot = torch.from_numpy(cv2.cvtColor(margot1, cv2.COLOR_BGR2RGB)).permute(2,0,1)
F.to_pil_image(margot)
```

Out[ ]:

In [ ]:
```
kobe0 = cv2.imread('/content/drive/MyDrive/Advanced Programming/Dati dogs/kobe.jpg')
kobe1 = cv2.resize(kobe0, dsize=(224,224),interpolation=cv2.INTER_CUBIC)
kobe = torch.from_numpy(cv2.cvtColor(kobe1, cv2.COLOR_BGR2RGB)).permute(2,0,1)
F.to_pil_image(kobe)
```

Out[ ]:

In [ ]:
```
from torchvision.transforms.functional import convert_image_dtype

batch_int = torch.stack([margot.to(device), kobe.to(device)])
batch = convert_image_dtype(batch_int, dtype=torch.float)

outputs = AlexNet_model(batch)
_, predicted = torch.max(outputs.data, 1)

print("Prediction with AlexNet (not pretrained): \n")
print('Correct breed of Margot: Bernese Mountain Dog. Predicted breed:', name_classes[predicted[0]])
print('Correct breed of Kobe: unknown. Predicted breed:', name_classes[predicted[1]])
```

```
Prediction with AlexNet (not pretrained):

Correct breed of Margot: Bernese Mountain Dog. Predicted breed: Afghan_hound
Correct breed of Kobe: unknown. Predicted breed: dhole
```

In [ ]:
```
outputs = AlexNet_model2(batch)
_, predicted = torch.max(outputs.data, 1)

print("Prediction with AlexNet (pretrained as fixed features exctractor): \n")
print('Correct breed of Margot: Bernese Mountain Dog. Predicted breed:', name_classes[predicted[0]])
print('Correct breed of Kobe: unknown. Predicted breed:', name_classes[predicted[1]])
```

```
Prediction with AlexNet (pretrained as fixed features exctractor):

Correct breed of Margot: Bernese Mountain Dog. Predicted breed: Gordon_setter
Correct breed of Kobe: unknown. Predicted breed: dhole
```

In [ ]:
```
outputs = ResNet_model(batch)
_, predicted = torch.max(outputs.data, 1)

print("Prediction with ResNet (not pretrained): \n")
print('Correct breed of Margot: Bernese Mountain Dog. Predicted breed:', name_classes[predicted[0]])
print('Correct breed of Kobe: unknown. Predicted breed:', name_classes[predicted[1]])
```

```
Prediction with ResNet (not pretrained):

Correct breed of Margot: Bernese Mountain Dog. Predicted breed: Great_Pyrenees
Correct breed of Kobe: unknown. Predicted breed: dhole
```

In [ ]:
```
outputs = ResNet_model2(batch)
_, predicted = torch.max(outputs.data, 1)

print("Prediction with ResNet50 (pretrained as fixed features exctractor): \n")
print('Correct breed of Margot: Bernese Mountain Dog. Predicted breed:', name_classes[predicted[0]])
print('Correct breed of Kobe: unknown. Predicted breed:', name_classes[predicted[1]])
```

```
Prediction with ResNet50 (pretrained as fixed features exctractor):

Correct breed of Margot: Bernese Mountain Dog. Predicted breed: Bernese_mountain_dog
Correct breed of Kobe: unknown. Predicted breed: kelpie
```

Conclusion

It is important to notice that our dataset has a lot of classes and very few data for each class. When training a model, only 80% of the original data is considered, so we are training on even less data. If we wanted to guess a dog breed by randomly picking one of the 120 breeds, the probability of success would be 0,83 %, very low!

When we tried to implement a model from scratch, training the model took a very long time and a lot of epochs to achieve good accuracy. We trained the models using 100 epochs. The final accuracies on the test dataset are:

- AlexNet not pretrained: 85.59% accuracy in 100 epochs
- ResNet50 not pretrained: 87.83% accuracy in 100 epochs

Our solution to the long time that it took to train the model was to use pretrained models as fixed features exctractor. We used 2 models pretrained on a larger version of our dataset (the full ImageNet dataset) and we only updated the weights of the final fully connected layer. With this solution, we were able to reach a great accuracy on the test set with fewer epochs and in a shorter time. In just 3 epochs we were able to reach around 65% of accuracy in the train, while with the not pretrained models we reached that accuracy in 50 epochs. We can notice that ResNet50 performed better than AlexNet since is deeper model. The final accuracies on the test dataset are:

- AlexNet pretrained as a fixed feature exctractor: 82.09% accuracy in 75 epochs
- ResNet50 pretrained as a fixed feature exctractor: 87.27% accuracy in 75 epochs

We are very pleased with the results achieved with all the models considering the large amount of classes.