BBM409 : Introduction to Machine Learning Lab - Assignment 4

Berra Nur SARI - 21727671
Melih SUNMAN - 21827809

Part 1: Multi Layer Neural Network

- In this part of the assignment, we implement multi layer neural network for classification. Created network consists of one input layer, n hidden layer(s) and one output layer. We implemented forward and backward propagations with the loss function and learning setting. Actually, we implemented a back-propagation algorithm to train a neural network.

Implementing Artificial Neural Network

Required libraries are imported

In [73]:

```
import numpy as np
import pandas as pd
import os

#For Preprocessing
import cv2
import itertools
from tqdm.notebook import tqdm
#from tqdm import tqdm_notebook as tqdm
from PIL import Image, ImageOps

#Additional imports for functionality
from sklearn.utils import class_weight, shuffle
from sklearn import metrics
from sklearn.metrics import confusion_matrix

#For Graphing and Plotting Images
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
%matplotlib inline

#For Create Tables
from prettytable import PrettyTable
```

Getting foldernames from dataset

In [2]:

```
foldernames = os.listdir('raw-img')
```

Creating Empty lists for categories and the files

In [3]:

```
categories = []
files = []
i = 0
```

Going over all the folders and their categories in the foldernames

In [4]:

```
for k, folder in enumerate(foldernames):
    print(k , folder)
    #Getting the filenames
    filenames = os.listdir("raw-img/" + folder);
    for file in filenames:
        #Appending all the image files into one list
        files.append("raw-img/" + folder + "/" + file)
        #Appending categories into one list
        categories.append(k)
```

```
0 cane
1 cavallo
2 elefante
3 farfalla
4 gallina
5 gatto
6 mucca
7 pecora
8 ragno
9 scoiattolo
```

Defining a DataFrame to store data

In [5]:

```
df = pd.DataFrame({
    'filename': files,
    'category': categories
})
df
```

Out[5]:

	filename	category
0	raw-img/cane/OIF-e2bexWrojgtQnAPPcUfOWQ.jpeg	0
1	raw-img/cane/OIP---A27bIBcUgX1qkbpZOPswHaFS.jpeg	0
2	raw-img/cane/OIP---cByAiEbIxIAleGo9AqOQAAAA.jpeg	0
3	raw-img/cane/OIP---ZIdwfUcJeVxnh47zppcQHaFj.jpeg	0
4	raw-img/cane/OIP---ZRsOF7zsMqhW30WeF8-AHaFj.jpeg	0
...	...	...
26174	raw-img/scoiattolo/OIP-_U7JiloYjbWPqmmmmdsvJwH...	9
26175	raw-img/scoiattolo/OIP-_VBkNQd_MZI4xoemUb-FtAH...	9
26176	raw-img/scoiattolo/OIP-_WyHKgREia-4VijlL6DNswH...	9
26177	raw-img/scoiattolo/OIP-_xFGMN0UbYduHdiXQ1maZAH...	9
26178	raw-img/scoiattolo/OIP-_XkUFCI2duAyKDD9utKQzgH...	9

26179 rows × 2 columns

Preprocessing on images:

- We want to train the network feeding by given training set as gray-level image values and size of images are 32x32. For this reason, we converted the pictures to black and white and changed their size.

- Since the data to be used during ANN training will be flatten, we converted our matrices to flatten and normalized them.

In [6]:

```
images = []

def process_image(img_path: str) -> np.array:
    img = Image.open(img_path)
    #to convert the image to grayscale
    img = ImageOps.grayscale(img)
    #to resizes our image (32 pixels wide and tall)
    img = img.resize(size=(32, 32))
    #to flatten the image and normalization
    img = np.ravel(img) / 255.0
    return img

#tqdm is used for visualing the progress of the image preprocessing as a progress bar
with tqdm(total=len(df)) as pbar:
    #Going over all the filenames in train_df
    for i, file_path in enumerate(df.filename.values):
        img = process_image(file_path)
        images.append(img)
        pbar.update(1)
```

```
  0%|          | 0/26179 [00:00<?, ?it/s]
```

In [26]:

```
images = np.array(images)
images.shape
```

Out[26]:

```
(26179, 1024)
```

This is an example to show how preprocessing works

In [27]:

```
example_path = df.filename.values[2]
test_image = process_image(example_path)
print(test_image)
src_img = Image.open(example_path)
display(src_img)

#Reverse the last step to represent the image visually with using array
Image.fromarray(np.uint8(test_image * 255).reshape((32, 32)))
```

```
[0.45490196 0.47058824 0.47843137 ... 0.57647059 0.57647059 0.60392157]
```

Out[27]:

Shuffle the data and convert to numpy

In [28]:

```
#Assigning x and y to be the values and their target labels respectively
x = df['filename']
y = df['category']

#Getting a list of the number of images used and a random index permutation from the data
#to randomly append images into x_shuffle along with their labels
data_num = len(y)
random_index = np.random.permutation(data_num)

#Shuffling the data
x, y = shuffle(x, y)

#Empty lists to store shuffled data
x_shuffle = []
y_shuffle = []
for i in range(data_num):
    x_shuffle.append(images[random_index[i]])
    y_shuffle.append(y[random_index[i]])

x = np.array(x_shuffle)
y = np.array(y_shuffle)
```

In [29]:

```
x.shape , y.shape
```

Out[29]:

```
((26179, 1024), (26179,))
```

Partitioning 20% of the dataset into test set and partitioning 20% of the dataset into validation set

In [30]:

```
split_num = int(round(0.2*len(y)))
split_num
```

Out[30]:

```
5236
```

60% of the data set was divided into training set, 20% validation set, and 20% test set

In [31]:

```
x_train = x[:len(y)-split_num*2]
y_train = y[:len(y)-split_num*2]
print("size of training sets   " , x_train.shape , " " , y_train.shape)

x_validation = x[len(y)-split_num*2:len(y)-split_num]
y_validation = y[len(y)-split_num*2:len(y)-split_num]
print("size of validation sets " ,x_validation.shape , "  " , y_validation.shape)

x_test = x[len(y)-split_num:]
y_test = y[len(y)-split_num:]
print("size of test sets       " ,x_test.shape , "  " , y_test.shape)
```

```
size of training sets    (15707, 1024)   (15707,)
size of validation sets  (5236, 1024)    (5236,)
size of test sets        (5236, 1024)    (5236,)
```

ANN class for create - train and test the model

In [105]:

```
class ANN(object):
    def __init__(self, train_set_samples, train_set_labels , number_of_hidden_layers , size
        self.X = train_set_samples
        self.y = train_set_labels
        self.number_of_hidden_layers = number_of_hidden_layers
        self.batch_size = batch_size
        self.epoch = epoch

        self.D = 32*32 #size of images
        self.K = 10 #number of classes
        self.h = size_of_hidden_layers   #size of hidden layer

        self.step_size = step_size
        self.reg = regularization_strength # regularization strength

        self.weights_list = []
        self.biases_list = []
        self.hidden_layer_score=[]

        self.loss = 0
        self.losses = [];

    def create_hidden_layer(self, n_inputs): #32*32 or 128
        weights = 0.01 * np.random.randn(n_inputs, self.h)
        self.weights_list.append(weights)
        biases = np.zeros((1, self.h))
        self.biases_list.append(biases)

    def create_final_layer(self):
        self.softmax_weights = 0.01 * np.random.randn(self.h, self.K)
        self.softmax_biases  = np.zeros((1,self.K))

    def softmax(self, scores):
        expX = np.exp(scores)
        return expX / np.sum(expX, axis=1, keepdims=True)# [N x K]

    def relu(self, data):
        return np.maximum(0, data)

    #average cross-entropy loss and regularization
    def compute_the_loss(self, labels , probs):
        correct_logprobs = -np.log(probs[range(len(labels)),labels])
        data_loss = np.sum(correct_logprobs)/len(labels)
        reg_loss = 0
        for i in range(len(self.weights_list)):
            reg_loss += 0.5*self.reg*np.sum(self.weights_list[i] ** 2)
        reg_loss += 0.5*self.reg*np.sum(self.softmax_weights ** 2)
        self.loss = data_loss + reg_loss

    def gradient(self, labels, probs):
        dscores = probs
        dscores[range(len(labels)),labels] -= 1
        dscores /= len(labels)
        return dscores

    def backpropate(self,dscores):
        d_hiddens_list = []
        d_weights_list = []
        d_biases_list = []
```

```
        j = self.number_of_hidden_layers

        d_softmax_weights = np.dot(self.hidden_layer_score[j].T, dscores)
        d_softmax_biases = np.sum(dscores, axis=0, keepdims=True)

        dhidden = np.dot(dscores, self.softmax_weights.T)

        dhidden[self.hidden_layer_score[j] <= 0] = 0


        for i in range(self.number_of_hidden_layers):
            j -= 1
            d_weights_list.insert(0, np.dot(self.hidden_layer_score[j].T, dhidden))
            d_biases_list.insert(0, np.sum(dhidden, axis=0, keepdims=True))
            dhidden = np.dot(dhidden, self.weights_list[j].T)
            dhidden[self.hidden_layer_score[j] <= 0] = 0


        # add regularization gradient contribution
        d_softmax_weights += self.reg * self.softmax_weights
        for i in range(len(d_weights_list)):
            d_weights_list[i] +=  self.reg * self.weights_list[i]

        for i in range(len(self.weights_list)):
            self.weights_list[i] += -self.step_size* d_weights_list[i]
            self.biases_list[i] += -self.step_size* d_biases_list[i]
        self.softmax_weights += -self.step_size* d_softmax_weights
        self.softmax_biases += -self.step_size* d_softmax_biases

    def plot_cost(self):
        plt.figure()
        plt.plot(np.arange(len(self.losses)), self.losses)
        plt.xlabel("epochs")
        plt.ylabel("cost")
        plt.show()

    def create_batch(self, batch_size=32):
        mini_batches=[]
        no_of_batches=self.X.shape[0]//batch_size
        temp = 0

        for i in range(no_of_batches):
            mini_batchX = self.X[i*batch_size:(i+1)*batch_size]
            mini_batchY = self.y[i*batch_size:(i+1)*batch_size]
            mini_batches.append((mini_batchX,mini_batchY))

        if self.X.shape[0] % batch_size != 0:
            mini_batchX = self.X[(i+1)*batch_size:]
            mini_batchY = self.y[(i+1)*batch_size:]
            mini_batches.append((mini_batchX,mini_batchY))

        return mini_batches

    def train(self):

        for i in range(self.number_of_hidden_layers):
            if(i == 0):
                self.create_hidden_layer(self.D)
            else:
                self.create_hidden_layer(self.h)
        self.create_final_layer()
```

```
        batches = self.create_batch(self.batch_size)

        for iteration in range(self.epoch):
            flag = True
            loss_flag = True
            for batch in batches:

                data_set = batch[0]
                batch_labels = batch[1]

                self.hidden_layer_score = []
                self.hidden_layer_score.append(data_set)

                for i in range(self.number_of_hidden_layers):
                    if(i == 0):
                        score = self.relu(np.dot(data_set, self.weights_list[i]) + self.bia
                        self.hidden_layer_score.append(score)
                    else:
                        score = self.relu(np.dot(score, self.weights_list[i]) + self.biases
                        self.hidden_layer_score.append(score)
                score = np.dot(score, self.softmax_weights) + self.softmax_biases
                #self.hidden_layer_score.append(score)
                probabilities = self.softmax(score)
                self.compute_the_loss(batch_labels, probabilities)
                if loss_flag:
                    self.losses.append(self.loss)
                    loss_flag = False
                if flag and iteration % 250 == 0 :
                    print("iteration %d: loss %f" % (iteration, self.loss))
                    predicted_class = np.argmax(probabilities, axis=1)
                    print("training accuracy: %f" % (np.mean(predicted_class == batch_label
                    flag = False

                dscores = self.gradient(batch_labels, probabilities)
                self.backpropate(dscores)



    def test(self, test_set, test_label_set):
        for i in range(self.number_of_hidden_layers):
            if(i == 0):
                score = self.relu(np.dot(test_set, self.weights_list[i]) + self.biases_list
            else:
                score = self.relu(np.dot(score, self.weights_list[i]) + self.biases_list[i]
        score = np.dot(score, self.softmax_weights) + self.softmax_biases
        probabilities = self.softmax(score)
        self.compute_the_loss(test_label_set,probabilities)

        print("loss %f" % (self.loss))
        predicted_class = np.argmax(probabilities, axis=1)
        print("test accuracy: %f" % (np.mean(predicted_class == test_label_set)))
```

- The data set of 15 thousand was too large for the validation processes we will do to get the best out of the model. It was taking too long. Therefore, it was necessary to use a smaller data set to determine the model parameters. For this reason, we used a small part of the train set.

In [49]:

```
split_num2 = int(round(0.1*len(y_train)))
split_num2
```

Out[49]:

```
1571
```

In [50]:

```
x_train_for_parameters = x_train[len(y_train)-split_num2:]
y_train_for_parameters = y_train[len(y_train)-split_num2:]
x_train_for_parameters.shape , y_train_for_parameters.shape
```

Out[50]:

```
((1571, 1024), (1571,))
```

In [51]:

```
x_validation_for_parameters = x_validation[len(y_validation)-split_num2:]
y_validation_for_parameters = y_validation[len(y_validation)-split_num2:]
x_validation_for_parameters.shape , y_validation_for_parameters.shape
```

Out[51]:

```
((1571, 1024), (1571,))
```

def **init**(self, train_set_samples, train_set_labels , number_of_hidden_layers , size_of_hidden_layers = 128 , batch_size = 32 , step_size = 5e-2 ,regularization_strength = 1e-3):

- In our first comparison, the effect of the number of hidden layers in our model on the success of the model is examined

In [52]:

```
model1 =  ANN(x_train_for_parameters,y_train_for_parameters,1)
```

In [53]:

```
model1.train()
```

```
iteration 0: loss 2.309643
training accuracy: 0.125000
iteration 1000: loss 0.742189
training accuracy: 0.937500
iteration 2000: loss 0.444153
training accuracy: 1.000000
```

In [54]:

```
model2 =  ANN(x_train_for_parameters,y_train_for_parameters,2)
```

In [55]:

```
model2.train()
```

```
iteration 0: loss 2.309908
training accuracy: 0.062500
iteration 1000: loss 0.311818
training accuracy: 1.000000
iteration 2000: loss 0.345530
training accuracy: 1.000000
```

In [56]:

```
model1.test(x_validation_for_parameters,y_validation_for_parameters)
```

```
loss 4.254196
test accuracy: 0.168682
```

In [57]:

```
model2.test(x_validation_for_parameters,y_validation_for_parameters)
```

```
loss 4.739755
test accuracy: 0.222788
```

In [77]:

```
table = PrettyTable(["Models", "Loss", "Accuracy"])
table.add_row(["Model1(1)", "4.254196", "0.168682"])
table.add_row(["Model2(2)", "4.739755", "0.222788"])
print(table)
```

```
+-----------+----------+----------+
|   Models  |   Loss   | Accuracy |
+-----------+----------+----------+
| Model1(1) | 4.254196 | 0.168682 |
| Model2(2) | 4.739755 | 0.222788 |
+-----------+----------+----------+
```

- Obviously, the model which has 2 hidden layers, has higher accuracy. For this reason, we will continue with the model that has 2 hidden layers.

- In our second comparison, the effect of the size of hidden layers in our model on the success of the model is examined

In [59]:

```
model3 =  ANN(x_train_for_parameters,y_train_for_parameters,2,64)
```

In [61]:

```
model3.train()
```

```
iteration 0: loss 2.306210
training accuracy: 0.000000
iteration 1000: loss 0.325267
training accuracy: 1.000000
iteration 2000: loss 1.146797
training accuracy: 0.718750
```

In [62]:

```
model4 =  ANN(x_train_for_parameters,y_train_for_parameters,2,128)
```

In [63]:

```
model4.train()
```

```
iteration 0: loss 2.310126
training accuracy: 0.000000
iteration 1000: loss 1.876717
training accuracy: 0.468750
iteration 2000: loss 0.269963
training accuracy: 1.000000
```

In [64]:

```
model5 =  ANN(x_train_for_parameters,y_train_for_parameters,2,256)
```

In [65]:

```
model5.train()
```

```
iteration 0: loss 2.318189
training accuracy: 0.125000
iteration 1000: loss 1.284069
training accuracy: 0.718750
iteration 2000: loss 0.769369
training accuracy: 0.812500
```

In [66]:

```
model3.test(x_validation_for_parameters,y_validation_for_parameters)
```

```
loss 4.909180
test accuracy: 0.208148
```

In [67]:

```
model4.test(x_validation_for_parameters,y_validation_for_parameters)
```

```
loss 5.529141
test accuracy: 0.210694
```

In [68]:

```
model5.test(x_validation_for_parameters,y_validation_for_parameters)
```

```
loss 5.247144
test accuracy: 0.243157
```

In [78]:

```
table = PrettyTable(["Models", "Loss", "Accuracy"])
table.add_row(["Model3(64)", "4.909180", "0.208148"])
table.add_row(["Model4(128)", "5.529141", "0.210694"])
table.add_row(["Model5(256)", "5.247144", "0.243157"])
print(table)
```

```
+-------------+----------+----------+
|    Models   |   Loss   | Accuracy |
+-------------+----------+----------+
|  Model3(64) | 4.909180 | 0.208148 |
| Model4(128) | 5.529141 | 0.210694 |
| Model5(256) | 5.247144 | 0.243157 |
+-------------+----------+----------+
```

- The model which size of hidden layers is 256, has higher accuracy. For this reason, we will continue with the model that size of hidden layers is 256

- In our third comparison, the effect of the batch size in our model on the success of the model is examined

In [79]:

```
model6 =  ANN(x_train_for_parameters,y_train_for_parameters,2,256,32)
```

In [80]:

```
model6.train()
```

```
iteration 0: loss 2.319158
training accuracy: 0.031250
iteration 1000: loss 0.761739
training accuracy: 0.875000
iteration 2000: loss 0.538830
training accuracy: 0.968750
```

In [81]:

```
model7 =  ANN(x_train_for_parameters,y_train_for_parameters,2,256,64)
```

In [82]:

```
model7.train()
```

```
iteration 0: loss 2.319486
training accuracy: 0.015625
iteration 1000: loss 0.546725
training accuracy: 0.937500
iteration 2000: loss 0.616635
training accuracy: 0.921875
```

In [83]:

```
model8 =  ANN(x_train_for_parameters,y_train_for_parameters,2,256,128)
```

In [84]:

```
model8.train()
```

```
iteration 0: loss 2.318739
training accuracy: 0.109375
iteration 1000: loss 0.355840
training accuracy: 1.000000
iteration 2000: loss 0.518063
training accuracy: 0.960938
```

In [85]:

```
model6.test(x_validation_for_parameters,y_validation_for_parameters)
```

```
loss 5.142517
test accuracy: 0.245703
```

In [86]:

```
model7.test(x_validation_for_parameters,y_validation_for_parameters)
```

```
loss 5.051058
test accuracy: 0.236792
```

In [87]:

```
model8.test(x_validation_for_parameters,y_validation_for_parameters)
```

```
loss 3.202471
test accuracy: 0.212603
```

In [102]:

```
table = PrettyTable(["Models", "Loss", "Accuracy"])
table.add_row(["Model6(32)", "5.142517", "0.245703"])
table.add_row(["Model7(64)", "5.051058", "0.236792"])
table.add_row(["Model8(128)", "3.202471", "0.212603"])
print(table)
```

```
+-------------+----------+----------+
|    Models   |   Loss   | Accuracy |
+-------------+----------+----------+
|  Model6(32) | 5.142517 | 0.245703 |
|  Model7(64) | 5.051058 | 0.236792 |
| Model8(128) | 3.202471 | 0.212603 |
+-------------+----------+----------+
```

- We create our final model by choosing the higher accuracy ones.
- We use the real train set to train this model.

In [99]:

```
split_num3 = int(round(0.5*len(y_train)))

x_train_for_parameters = x_train[len(y_train)-split_num3:]
y_train_for_parameters = y_train[len(y_train)-split_num3:]
x_train_for_parameters.shape , y_train_for_parameters.shape
```

Out[99]:

```
((7854, 1024), (7854,))
```

In [106]:

```
model9 =  ANN(x_train_for_parameters,y_train_for_parameters,2,256,32,5e-2 ,1e-3 , 1000)
```

In [107]:

```
model9.train()
```

```
iteration 0: loss 2.318631
training accuracy: 0.093750
iteration 250: loss 0.866025
training accuracy: 0.937500
iteration 500: loss 0.714254
training accuracy: 1.000000
iteration 750: loss 0.619105
training accuracy: 1.000000
```

In [108]:

```
model9.test(x_test,y_test)
```

```
loss 3.359079
test accuracy: 0.348930
```

- We could not train using the whole training dataset in the model we used, because the training takes really long and time-consuming, we had to wait for it to train again after making a small change in the model, but we did not have enough time for this. When we look at the training accuracy of the model, we can say that we have an overfitting problem. There were some things we could do to resolve this. For example, we could have used more smaples during the training of the model. Another option would be to change the size of the images we use. If we had worked with 128x128 images instead of 32x32, the overfitting problem might have been less.

Part 1 Multi Layer Neural Network		
	accuracy	loss
Model3	0.208148	4.909.180
Model4	0.210694	5.529.141
Model5	0.243157	5.247.144
Model6	0.245703	5.142.517
Model7	0.236792	5.051.058
Model8	0.212603	3.202.471
Model9	0.348930	3.359.079

Part 2: Convolutional Neural Network

A) All layers in the VGG-19

Required libraries are imported

In [1]:

```
from google.colab import drive
import os
from torch.utils.data import Dataset, DataLoader
import glob
import torch
import cv2
import torchvision
import torch.optim as optim

drive.mount('/content/drive/')
```

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

Getting foldernames from dataset

In [2]:

```
class CustomDataset(Dataset):
  def __init__(self):
    self.base_path = "/content/drive/MyDrive/raw-img/"
    file_list = glob.glob(self.base_path + "*")
    self.data = []
    for class_path in file_list:
      class_name = class_path.split("/")[-1]
      for img_path in glob.glob(class_path + "/*.jpeg"):
        self.data.append([img_path, class_name])
        self.class_map = {"cane": 0, "cavallo": 1, "elefante": 2, "farfalla": 3, "gallina":
                    "pecora": 7, "ragno": 8, "scoiattolo": 9}
        self.img_dim = (224, 224)
  def __len__(self):
    return len(self.data)
  def __getitem__(self, idx):
    img_path, class_name = self.data[idx]
    img = cv2.imread(img_path)
    img = cv2.resize(img, self.img_dim)
    label = self.class_map[class_name]
    img_tensor = torch.from_numpy(img)
    img_tensor = img_tensor.float()
    img_tensor = img_tensor.permute(2, 0, 1)
    label = torch.tensor(label)
    return img_tensor, label
```

In [3]:

```
dataset = CustomDataset()
```

In [4]:

```
print(len(dataset))
```

```
24209
```

In [5]:

```
train_set, test_set = torch.utils.data.random_split(dataset, [int(len(dataset)*8/10), len(d
```

In [6]:

```
train_data_loader = torch.utils.data.DataLoader(
       train_set, batch_size=32,
       shuffle=True
   )

test_data_loader = torch.utils.data.DataLoader(
       test_set, batch_size=32,
       shuffle=True
   )
```

In [7]:

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

```
cuda:0
```

Implementing VGG19 Algorithm

In [8]:

```
vgg_based = torchvision.models.vgg19(pretrained=True)

for param in vgg_based.parameters():
  param.requires_grad = False

# Modify the last layer
number_features = vgg_based.classifier[6].in_features
features = list(vgg_based.classifier.children())[:-1] # Remove last layer
features.extend([torch.nn.Linear(number_features, 10)])
vgg_based.classifier = torch.nn.Sequential(*features)

vgg_based = vgg_based.to(device)

criterion = torch.nn.CrossEntropyLoss()
optimizer_ft = optim.SGD(vgg_based.parameters(), lr=0.001, momentum=0.9)
```

```
Downloading: "https://download.pytorch.org/models/vgg19-dcbb9e9d.pth" to /ro
ot/.cache/torch/hub/checkpoints/vgg19-dcbb9e9d.pth

  0%|          | 0.00/548M [00:00<?, ?B/s]
```

Definition of Train function

In [9]:

```
def train_model(model, criterion, optimizer, num_epochs=10):
  for epoch in range(num_epochs):
    print('Epoch {}/{}'.format(epoch, num_epochs - 1))
    print('-' * 10)
    train_loss = 0
    for i, data in enumerate(train_data_loader):
      inputs , labels = data
      inputs = inputs.to(device)
      labels = labels.to(device)
      optimizer.zero_grad()
      with torch.set_grad_enabled(True):
        outputs  = model(inputs)
        loss = criterion(outputs, labels)
      loss.backward()
      optimizer.step()
      train_loss += loss.item() * inputs.size(0)
      print('{} Ara Loss: {:.4f}'.format('train', train_loss / len(train_set)))
    print('{} Loss: {:.4f}'.format(
              'train', train_loss / len(train_set)))
  return model
```

Training Model (VGG19)

In [10]:

```
vgg_based = train_model(vgg_based, criterion, optimizer_ft, num_epochs=10)
```

```
Görüntülenen çıkış son 5000 satıra kısaltıldı.
train Ara Loss: 54.1495
train Ara Loss: 54.2564
train Ara Loss: 54.3873
train Ara Loss: 54.5025
train Ara Loss: 54.5689
train Ara Loss: 54.7063
train Ara Loss: 54.7687
train Ara Loss: 54.8713
train Ara Loss: 54.9964
train Ara Loss: 55.0701
train Ara Loss: 55.1627
train Ara Loss: 55.2795
train Ara Loss: 55.3758
train Ara Loss: 55.4763
train Ara Loss: 55.5505
train Ara Loss: 55.6466
train Ara Loss: 55.7237
train Ara Loss: 55.8425
```

In [11]:

```
def test_model(model):
  print("this is test")
```

In [12]:

```
vgg_based
```

Out[12]:

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode
=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode
=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (15): ReLU(inplace=True)
    (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (17): ReLU(inplace=True)
    (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mod
e=False)
    (19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (22): ReLU(inplace=True)
    (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (24): ReLU(inplace=True)
    (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (26): ReLU(inplace=True)
    (27): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mod
e=False)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (29): ReLU(inplace=True)
    (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (31): ReLU(inplace=True)
    (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (33): ReLU(inplace=True)
    (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (35): ReLU(inplace=True)
    (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mod
```

```
e=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=10, bias=True)
  )
)
```

Implementation of the optimization algorithm

In [13]:

```
import torch.optim as optim
from torch import nn
# specify loss function (categorical cross-entropy)
criterion = nn.CrossEntropyLoss()

# specify optimizer (stochastic gradient descent) and learning rate = 0.001
optimizer = optim.SGD(vgg_based.parameters(), lr=0.001)
```

In [14]:

```
train_on_gpu = torch.cuda.is_available()
import numpy as np
```

Implemention of Test function

In [15]:

```
def seq (model, df, name ):
    train_loss = 0.0
    class_correct = list(0. for i in range(10))
    class_total = list(0. for i in range(10))
    for batch_i, (data, target) in enumerate(df):
        # move tensors to GPU if CUDA is available
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()
            model.cuda()
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the batch loss
        loss = criterion(output, target)
        # backward pass: compute gradient of the loss with respect to model parameters
        if name == 'train':
            loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()
        # update training loss
        train_loss += loss.item()
        _, pred = torch.max(output, 1)
        # compare predictions to true label
        correct_tensor = pred.eq(target.data.view_as(pred))
        correct = np.squeeze(correct_tensor.numpy()) if not train_on_gpu else np.squeeze(co
        for i in range(len(target.data)):
            label = target.data[i]
            class_correct[label] += correct[i].item()
            class_total[label] += 1

    return class_correct, class_total, train_loss
```

Definition of Labels

In [20]:

```
translate = {"cane": "dog", "cavallo": "horse", "elefante": "elephant", "farfalla": "butter
             "gallina": "chicken", "gatto": "cat", "mucca": "cow", "pecora": "sheep",
             "ragno": "spider", "scoiattolo": "squirrel" }
classes = ["cane", "cavallo", "elefante", "farfalla", "gallina", "gatto", "mucca", "pecora"
```

Definition of the function that prints the results

In [21]:

```
def printdata(class_correct, class_total, train_loss, epoch, name, df ):
    print(f'Epoch %d, loss: %.8f \t{name} Accuracy (Overall): %2d%% (%2d/%2d)' %(epoch,
        train_loss / len(df), 100. * np.sum(class_correct) / np.sum(class_total),
        np.sum(class_correct), np.sum(class_total)))
    if ((epoch+1) % 5 == 0 or epoch == 1):
        for i in range(10):
            if class_total[i] > 0:
                print(f'{name} Accuracy of %5s: %2d%% (%2d/%2d)' % (
                translate[classes[i]], 100 * class_correct[i] / class_total[i],
                np.sum(class_correct[i]), np.sum(class_total[i])))
```

In [22]:

```
 # track test loss
# over 10 animals classes
test_loss = 0.0
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
vgg_based.eval()
class_correct, class_total, train_loss= seq(vgg_based, test_data_loader, 'test')
printdata(class_correct, class_total, train_loss, 1, 'test', test_data_loader)
```

```
Epoch 1, loss: 32.72482463      test Accuracy (Overall): 54% (2624/4842)
test Accuracy of   dog: 68% (646/944)
test Accuracy of horse: 39% (214/541)
test Accuracy of elephant: 39% (89/223)
test Accuracy of butterfly: 67% (225/332)
test Accuracy of chicken: 67% (441/649)
test Accuracy of   cat: 22% (55/241)
test Accuracy of   cow: 60% (227/376)
test Accuracy of sheep: 35% (106/295)
test Accuracy of spider: 67% (593/882)
test Accuracy of squirrel:  7% (28/359)
```

We used the VGG19 artificial neural network in part A of the 2nd part of the project. Our dataset contains data from 10 different animals. We used the mini-batch technique for training, thus overcoming the disadvantages of the size of our dataset. Unlike part B, we trained all layers in the training and obtained the following accuracy values for different classes in the dataset.

Accuracy of dog: 68% (646/944)
Accuracy of horse: 39% (214/541)
Accuracy of elephant: 39% (89/223)
Accuracy of butterfly: 67% (225/332)
Accuracy of chicken: 67% (441/649)
Accuracy of cat: 22% (55/241)
Accuracy of cows: 60% (227/376)
Accuracy of sheep: 35% (106/295)
Accuracy of spider: 67% (593/882)
Accuracy of squirrel: 7% (28/359)

We achieved higher accuracy values in dog, butterfly, chicken classes compared to others, but especially in squirrel, the accuracy value was noticeably low. This is because squirrels are often treated as cat or bird. Trained Neural Network failed to understand this difference. One reason for this is the indiscriminate samples in the data set.

In [ ]:

```

```

Part 2: Convolutional Neural Network

B) FC1 and FC2 layers in the VGG-19

Required libraries are imported

In [ ]:

```
from __future__ import print_function, division
import os
import time
import torch
import torchvision
from torchvision import datasets, models, transforms
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt
```

In [1]:

```


import numpy as np
import pandas as pd

import os

import torch

import torchvision
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt
from torch import nn
for dirname, _, filenames in os.walk('/content/drive/MyDrive/ml_final/raw-img'):
    for filename in filenames:
        path, folder = os.path.split(dirname)
```

```
C:\ProgramData\Anaconda3\lib\site-packages\torchvision\io\image.py:11: UserW
arning: Failed to load image Python extension: Could not find module 'C:\Pro
gramData\Anaconda3\Lib\site-packages\torchvision\image.pyd' (or one of its d
ependencies). Try using the full path with constructor syntax.
  warn(f"Failed to load image Python extension: {e}")
```

In [ ]:

```
train_on_gpu = torch.cuda.is_available()
data_transform1 = transforms.Compose([transforms.RandomRotation(45),
                                     transforms.RandomRotation(30),
                                     transforms.RandomResizedCrop(1080),
                                     transforms.Resize(512),
                                     transforms.Resize(224),
                                     transforms.RandomRotation(45),
                                      transforms.ToTensor()])

data_transform2 = transforms.Compose<i style="color:green"> Getting foldernames from datase
                                     transforms.RandomResizedCrop(1080),
                                     transforms.Resize(224),
                                     transforms.RandomRotation(45),
                                      transforms.RandomRotation(35),
                                      transforms.ToTensor()])
```

Getting foldernames from dataset

In [ ]:

```
<i style="color:green"> Getting foldernames from dataset </i>from torch.utils.data import S
dataset1 = datasets.ImageFolder(path,transform=data_transform1)
dataset2 = datasets.ImageFolder(path,transform=data_transform2)
print(type(dataset1))
#master=datasets.ImageFolder(path,transform=data_transform1)
maxlen=750
for l, cls in enumerate(dataset1.classes):
    if l == 0 :
        idx = [i for i in range(len(dataset1) ) if dataset1.imgs[i][1] == dataset1.class_to
        subset = Subset(dataset1, idx)
        master= Subset(subset,idx [:maxlen])
        subset = Subset(dataset2, idx[:maxlen])
        master= ConcatDataset((master, subset))

        print(len(master))
    else :
        idx = [i for i in range(len(dataset1) ) if dataset1.imgs[i][1] == dataset1.class_to

        subset = Subset(dataset1, idx[:maxlen])
        master= ConcatDataset((master, subset))
        subset = Subset(dataset2, idx[:maxlen])
        master= ConcatDataset((master, subset))
        print(len(master))
        #print(len(master))
```

```
<class 'torchvision.datasets.folder.ImageFolder'>
1500
3000
4500
6000
7500
9000
10500
12000
13500
15000
```

In [ ]:

```
valid_size = 0.1
test_size = 0.1
num_train = len(master)
indices = list(range(num_train))
np.random.shuffle(indices)
valid_split = int(np.floor((valid_size) * num_train))
test_split = int(np.floor((valid_size+test_size) * num_train))
valid_idx, test_idx, train_idx = indices[:valid_split], indices[valid_split:test_split], in

num_workers = 6
batch_size= 60
disimage = 20
#data = torch.utils.data.DataLoader(master, batch_size=batch_size, num_workers=num_workers)

train_loader = Subset(master, train_idx)
valid_loader = Subset(master,valid_idx )
test_loader = Subset(master,test_idx )


train_loader =torch.utils.data.DataLoader(train_loader, batch_size=batch_size, num_workers=
valid_loader =torch.utils.data.DataLoader(valid_loader, batch_size=batch_size, num_workers=
test_loader =torch.utils.data.DataLoader(test_loader, batch_size=batch_size, num_workers=nu
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:481: U
serWarning: This DataLoader will create 6 worker processes in total. Our sug
gested max number of worker in current system is 2, which is smaller than wh
at this DataLoader is going to create. Please be aware that excessive worker
creation might get DataLoader running slow or even freeze, lower the worker
number to avoid potential slowness/freeze if necessary.
  cpuset_checked))
```

Definition of Labels

In [ ]:

```
### Definition of Labelsclasses = ["cane", "cavallo", "elefante", "farfalla", "gallina", "g

translate = {"cane": "dog", "cavallo": "horse", "elefante": "elephant", "farfalla": "butter
             "gallina": "chicken", "gatto": "cat", "mucca": "cow", "pecora": "sheep",
             "ragno": "spider", "scoiattolo": "squirrel" }

dataiter = iter(train_loader)
images, labels = dataiter.next()
images = images.numpy() # convert images to numpy for display
# plot the images in the batch, along with the corresponding labels
fig = plt.figure(figsize=(25, 4))
for idx in np.arange(disimage):
    ax = fig.add_subplot(2, disimage/2, idx+1, xticks=[], yticks=[])
    plt.imshow(np.transpose(images[idx], (1, 2, 0)))
    ax.set_title(translate[classes[labels[idx]]])
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:481: U
serWarning: This DataLoader will create 6 worker processes in total. Our sug
gested max number of worker in current system is 2, which is smaller than wh
at this DataLoader is going to create. Please be aware that excessive worker
creation might get DataLoader running slow or even freeze, lower the worker
number to avoid potential slowness/freeze if necessary.
  cpuset_checked))
```

In [ ]:

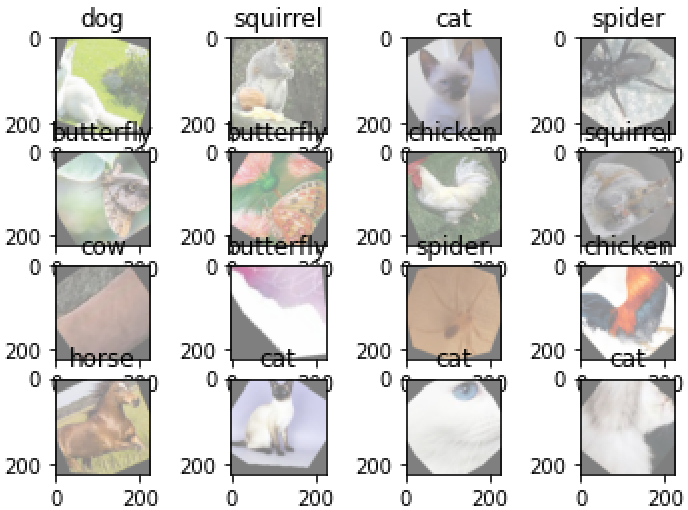
```
input_shape = 224
mean = [0.5, 0.5, 0.5]
std = [0.5, 0.5, 0.5]

#data transformation
data_transforms = {
   'train': transforms.Compose([
       transforms.CenterCrop(input_shape),
       transforms.ToTensor(),
       transforms.Normalize(mean, std)
   ]),
   'validation': transforms.Compose([
       transforms.CenterCrop(input_shape),
       transforms.ToTensor(),
       transforms.Normalize(mean, std)
   ]),
}
```

In [ ]:

```
dataiter = iter(train_loader)
images, labels = dataiter.next()
rows = 4
columns = 4
fig=plt.figure()
for i in range(16):
   fig.add_subplot(rows, columns, i+1)
   plt.title(translate[classes[labels[i]]])
   img = images[i].numpy().transpose((1, 2, 0))
   img = std * img + mean
   plt.imshow(img)
plt.show()
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:481: U
serWarning: This DataLoader will create 6 worker processes in total. Our sug
gested max number of worker in current system is 2, which is smaller than wh
at this DataLoader is going to create. Please be aware that excessive worker
creation might get DataLoader running slow or even freeze, lower the worker
number to avoid potential slowness/freeze if necessary.
  cpuset_checked))
```



In [ ]:

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

In [ ]:

```
len(train_loader)
```

Out[21]:

```
200
```

Implementing VGG19 Algorithm and Freezing Layers which we don't use

In [ ]:

```
## Load the model based on VGG19
vgg_based = torchvision.models.vgg19(pretrained=True)

## freeze the layers
for param in vgg_based.parameters():
   param.requires_grad = False

# Modify the last layer
number_features = vgg_based.classifier[6].in_features
features = list(vgg_based.classifier.children())[:-1] # Remove last layer
features.extend([torch.nn.Linear(number_features, len(classes))])
vgg_based.classifier = torch.nn.Sequential(*features)

vgg_based = vgg_based.to(device)

print(vgg_based)

criterion = torch.nn.CrossEntropyLoss()
optimizer_ft = optim.SGD(vgg_based.parameters(), lr=0.001, momentum=0.9)
```

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mo
de=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mo
de=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (15): ReLU(inplace=True)
    (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (17): ReLU(inplace=True)
    (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_m
ode=False)
    (19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (22): ReLU(inplace=True)
    (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (24): ReLU(inplace=True)
```

```
    (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 
1))
    (26): ReLU(inplace=True)
    (27): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_m
ode=False)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 
1))
    (29): ReLU(inplace=True)
    (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 
1))
    (31): ReLU(inplace=True)
    (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 
1))
    (33): ReLU(inplace=True)
    (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 
1))
    (35): ReLU(inplace=True)
    (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_m
ode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=10, bias=True)
  )
)
```

Definition of Train function

In [ ]:

```
### Definition of Train functiondef train_model(model, criterion, optimizer, num_epochs=25)
    since = time.time()

    for epoch in range(num_epochs):
        print('Epoch {}/{}'.format(epoch, num_epochs - 1))
        print('-' * 10)

        #set model to trainable
        # model.train()

        train_loss = 0

        # Iterate over data.
        for i, data in enumerate(train_loader):
            inputs , labels = data
            inputs = inputs.to(device)
            labels = labels.to(device)
            #print("labels : ",labels)

            optimizer.zero_grad()

            with torch.set_grad_enabled(True):
                outputs  = model(inputs)
                loss = criterion(outputs, labels)

            loss.backward()
            optimizer.step()

            train_loss += loss.item() * inputs.size(0)

            print('{} Loss: {:.4f}'.format(
                'train', train_loss / len(train_loader)))

    time_elapsed = time.time() - since
    print('Training complete in {:.0f}m {:.0f}s'.format(
        time_elapsed // 60, time_elapsed % 60))

    return model
def visualize_model(model, num_images=6):
    was_training = model.training
    model.eval()
    images_so_far = 0
    fig = plt.figure()
    global pred_false
    global pred_true
    pred_false =0
    pred_true = 0




    with torch.no_grad():
        for i, (inputs, labels) in enumerate(test_loader):
            inputs = inputs.to(device)
            labels = labels.to(device)

            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)
```

```
        for j in range(inputs.size()[0]):
            images_so_far += 1
            ax = plt.subplot(num_images//2, 2, images_so_far)
            ax.axis('off')
            ax.set_title('predicted: {} truth: {}'.format(translate[classes[preds[j]]],

            if (classes[preds[j]]== classes[labels[j]]):
              pred_true +=1

            else:
              pred_false+=1


            img = inputs.cpu().data[j].numpy().transpose((1, 2, 0))
            img = std * img + mean
            ax.imshow(img)

            if images_so_far == num_images:
                model.train(mode=was_training)
                return
    print("predictoins True : ",pred_true,"\nPredictions False :", pred_false)
    model.train(mode=was_training)
```

Training Model (VGG19)

In [ ]:

```
vgg_based = train_model(vgg_based, criterion, optimizer_ft, num_epochs=25)

visualize_model(vgg_based)

plt.show()
```

```
Epoch 0/24
----------

/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:481:
UserWarning: This DataLoader will create 6 worker processes in total. Our
suggested max number of worker in current system is 2, which is smaller th
an what this DataLoader is going to create. Please be aware that excessive
worker creation might get DataLoader running slow or even freeze, lower th
e worker number to avoid potential slowness/freeze if necessary.
  cpuset_checked))
```

In [ ]:

```
visualize_model(vgg_based)

plt.show()
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:481: U
serWarning: This DataLoader will create 6 worker processes in total. Our sug
gested max number of worker in current system is 2, which is smaller than wh
at this DataLoader is going to create. Please be aware that excessive worker
creation might get DataLoader running slow or even freeze, lower the worker
number to avoid potential slowness/freeze if necessary.
  cpuset_checked))
```

In [ ]:

```
pred_true
```

Out[47]:

```
2
```

In [ ]:

```
pred_false
```

Out[48]:

```
4
```

In [ ]:

```
import torch.optim as optim

# specify loss function (categorical cross-entropy)
criterion = nn.CrossEntropyLoss()

# specify optimizer (stochastic gradient descent) and learning rate = 0.001
optimizer = optim.SGD(vgg_based.parameters(), lr=0.001)
```

Implemention of Test function

In [ ]:

```
### Implemention of Test functiondef seq (model, df, name ):
    train_loss = 0.0
    class_correct = list(0. for i in range(10))
    class_total = list(0. for i in range(10))
    for batch_i, (data, target) in enumerate(df):
        # move tensors to GPU if CUDA is available
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()
            model.cuda()
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the batch loss
        loss = criterion(output, target)
        # backward pass: compute gradient of the loss with respect to model parameters
        if name == 'train':
            loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()
        # update training loss
        train_loss += loss.item()
        _, pred = torch.max(output, 1)
        # compare predictions to true label
        correct_tensor = pred.eq(target.data.view_as(pred))
        correct = np.squeeze(correct_tensor.numpy()) if not train_on_gpu else np.squeeze(co
        for i in range(len(target.data)):
            label = target.data[i]
            class_correct[label] += correct[i].item()
            class_total[label] += 1

    return class_correct, class_total, train_loss
```

In [ ]:

```
def printdata(class_correct, class_total, train_loss, epoch, name, df ):
    print(f' loss: %.8f \t{name} Accuracy (Overall): %2d%% (%2d/%2d)' %(
        train_loss / len(df), 100. * np.sum(class_correct) / np.sum(class_total),
        np.sum(class_correct), np.sum(class_total)))
    if ((epoch+1) % 5 == 0 or epoch == 1):
        for i in range(10):
            if class_total[i] > 0:
                print(f'{name} Accuracy of %5s: %2d%% (%2d/%2d)' % (
                translate[classes[i]], 100 * class_correct[i] / class_total[i],
                np.sum(class_correct[i]), np.sum(class_total[i])))
```

In [ ]:

```
test_loss = 0.0
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
vgg_based.eval()
class_correct, class_total, train_loss= seq(vgg_based, test_loader, 'test')
printdata(class_correct, class_total, train_loss, 1, 'test', test_loader)
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:481: U
serWarning: This DataLoader will create 6 worker processes in total. Our sug
gested max number of worker in current system is 2, which is smaller than wh
at this DataLoader is going to create. Please be aware that excessive worker
creation might get DataLoader running slow or even freeze, lower the worker
number to avoid potential slowness/freeze if necessary.
  cpuset_checked))

 loss: 1.22225365       test Accuracy (Overall): 57% (867/1500)
test Accuracy of   dog: 54% (77/141)
test Accuracy of horse: 42% (66/157)
test Accuracy of elephant: 51% (79/154)
test Accuracy of butterfly: 77% (121/157)
test Accuracy of chicken: 55% (76/136)
test Accuracy of   cat: 62% (93/149)
test Accuracy of   cow: 45% (67/148)
test Accuracy of sheep: 64% (107/165)
test Accuracy of spider: 78% (114/145)
test Accuracy of squirrel: 45% (67/148)
```

We used the VGG19 artificial neural network in part B of the 2nd part of the project. Unlike part A, we only trained certain layers and got a different test result. When we examined the results, we obtained a much more balanced score compared to the A part.

Accuracy of dog: 54% (77/141)
Accuracy of horses: 42% (66/157)
Accuracy of elephant: 51% (79/154)
Accuracy of butterfly: 77% (121/157)
Accuracy of chicken: 55% (76/136)
Accuracy of cat: 62% (93/149)
Accuracy of cows: 45% (67/148)
Accuracy of sheep: 64% (107/165)
Accuracy of spider: 78% (114/145)
Accuracy of squirrel: 45% (67/148)

We have achieved higher accuracy values in the butterfly and spider classes compared to the others, and there is no obvious decrease in the other classes. The reason for this is that we did not cause overfitting by working with more classes than necessary and we obtained a healthier test result.

In [ ]: