Animal Classification
Group 49 - Nogay Evirgen

1. Introduction

This project aims to create a tool that can recognize an animal' category from an image. Image of an animal will be given as input to this tool and it will make a prediction about this animal' category. Animals-10 dataset [1] will be used. This project contains two phases. The first phase is preprocessing. In this phase, we need to make sure that data is distributed equally to each class. Train and test sets should be constructed from the dataset. These sets should protect the ratio these classes have relative to their sets. Meaning, If the "dog" class forms 15% of the train set, the "dog" class should also form 15% of the test set. We can standardize our dataset after calculating the mean and standard deviation of the dataset. Standardization can be achieved by subtracting the mean from the dataset array and diving it with the standard deviation. After standardization, features can be extracted from VGG16, features of each image will be stored. In the second phase, machine learning models will be implemented and they will perform classification. The models that are going to be implemented are Logistic Regression, Linear SVM and Multilayer Perceptron (MLP). Advantages, disadvantages and reasons for using these models will be mentioned in their sections. Results will be evaluated with performance metrics.

2. Dataset Description and Preprocess

Animals-10 dataset consists of 10 classes distributed over 26179 images. Images don't have a fixed size and these images aren't distributed equally to classes. If we use the whole dataset to construct train and test sets with the ratios of 80% and 20% while keeping the same ratio of classes relative to the set they are in, the distribution of images in the train and test set would look like the following.

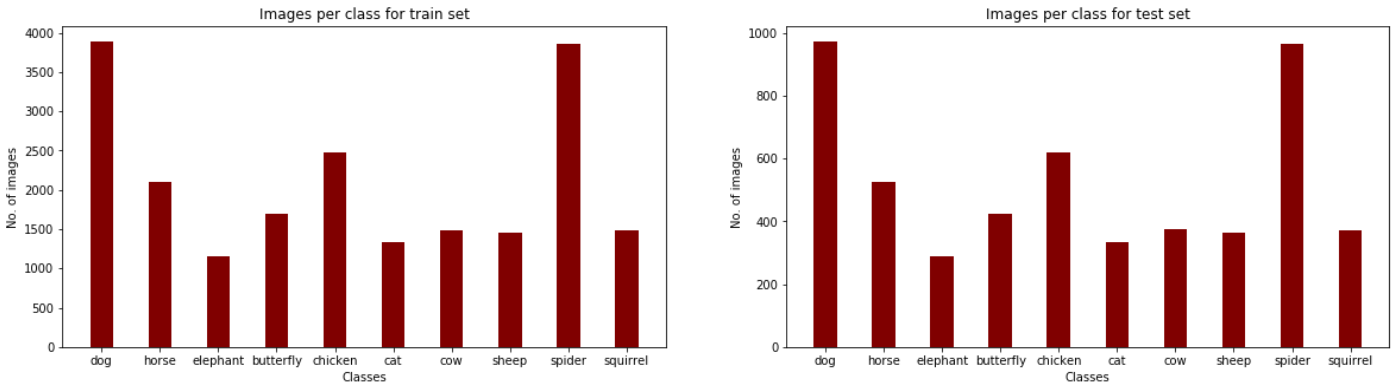


Figure 1: Distribution of images in the train and test sets.

From the figure, it is visible that the ratio of classes relative to the set they are in is protected. These graphs have the same distribution. However, the problem is that data is not distributed equally. In the train set, the class with the most images is the dog class with 3890 images and the class with the least images is the elephant class with 1156 images. We can distribute the data equally if each of the classes only has 1156 images. However, if we were to do that, the number of images in the train set would be less and it could be hard for our machine learning models to learn with a small train set. In order to suppress the classes with many images, I put a threshold of value 2500. Meaning, the classes with a higher number of the image than 2500 will be limited to 2500 images, other classes won't be affected. The new distribution of images in the train and the test looks like the following. Train set

used to contain 20938 images and a test set used to contain 5241 images. New train set contains 18192 images and the new test set contains 4551 images.

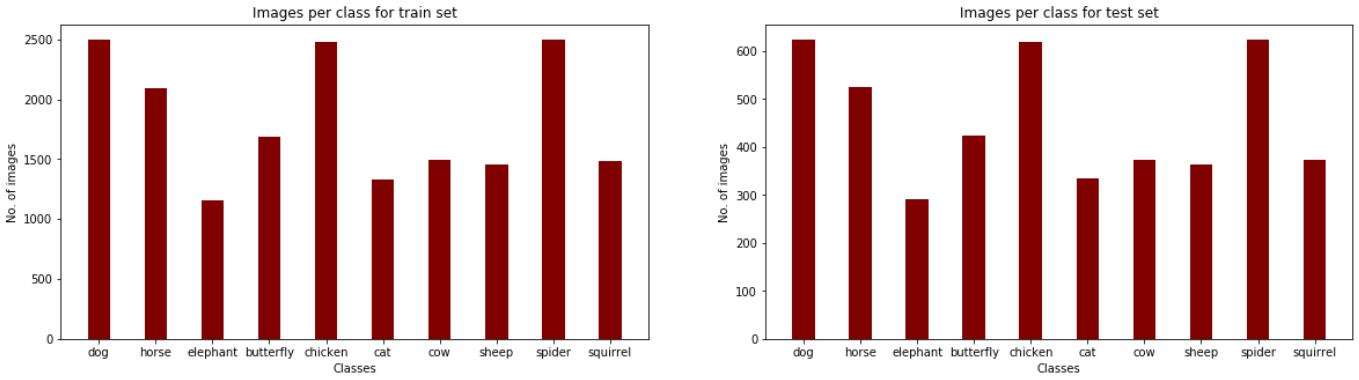


Figure 2: Distribution of images in the new train and new test sets.

Using each pixel of an image as a feature might work for models such as Convolution Neural Networks (CNN) or other neural network architectures. However, for models such as Logistic Regression and Linear SVM, they won't perform well. In those cases, feature extraction can be applied to have more meaningful features. VGG16 is a model trained with ImageNet which contains 1000 classes [2]. This is the architecture of VGG16.

```
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): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 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): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=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=1000, bias=True)
  )
)
```

Figure 3: Structure of VGG16

Currently, this model takes images with the size of 224px x 224px x 3 (3 channels, RGB) and performs classification [3]. If we remove the last 3 layers (ReLU, Dropout and Linear), this model can be used to extract features. The 4th layer in the classifier has 4096 outputs which mean we'll extract 4096 features per sample. Meaning this model takes images with the size 224px x 224px x 3 and returns a vector with size 4096. This process will be done for every image in the train and test sets. Each image will be resized to 224px x 224px x 3, then it will be forwarded to the model. This process takes around 150 minutes (around 0.33 second per image).

3. Completed Algorithms and Findings

3.1. Logistic Regression

Logistic Regression model is chosen for this problem since it is easy to implement, easy to interpret and it's very fast at making predictions. One disadvantage of this model is that it's not able to capture non-linearities [4]. It's a linear classifier. There are multiple classes in this dataset, that's why binary logistic regression won't work in this problem. In multiclass classification problems, it's popular to use softmax classifier in models when there are multiple classes. The benefit of using a softmax classifier is that it gives a probability for each class while making a prediction [5]. Thus, models confidence can easily be seen when it makes a prediction. As the optimizer, model will use batch gradient descent where the batch size is equal to the size of the train set [6]. As the loss function, cross-entropy will be used. Weights of the model initialized randomly.

```
Starting to train.
-----------------
epoch: 1/100 , loss: 2.35, training accuracy: 0.16
epoch: 10/100 , loss: 0.83, training accuracy: 0.90
epoch: 20/100 , loss: 0.56, training accuracy: 0.92
epoch: 30/100 , loss: 0.45, training accuracy: 0.93
epoch: 40/100 , loss: 0.39, training accuracy: 0.93
epoch: 50/100 , loss: 0.35, training accuracy: 0.93
epoch: 60/100 , loss: 0.33, training accuracy: 0.93
epoch: 70/100 , loss: 0.31, training accuracy: 0.93
epoch: 80/100 , loss: 0.29, training accuracy: 0.93
epoch: 90/100 , loss: 0.28, training accuracy: 0.94
epoch: 100/100 , loss: 0.27, training accuracy: 0.94
-----------------
time elapsed: 53.75 seconds
```

Figure 4: Training the logistic regression model

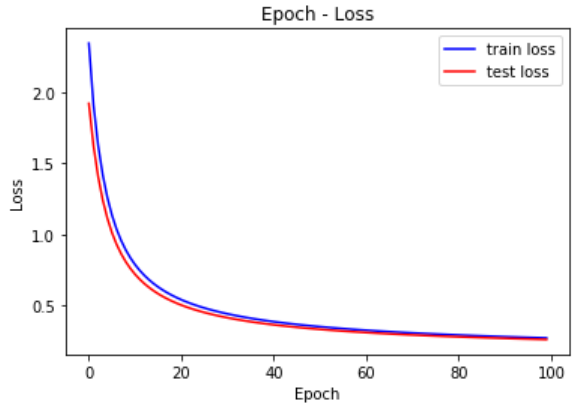


Figure 5:Loss history

Figure 4 and figure 5 shows training details when the learning rate is 0.001. Training this model for 100 epochs took 53.75 seconds As you can see from figure 5, choosing 100 as epoch number is not necessary. Since, after epoch 50, training loss rarely decreases. An epoch number of 40 would be more efficient and it would take around 27 seconds to train the model. Here are the results when the model is tested on the test set.

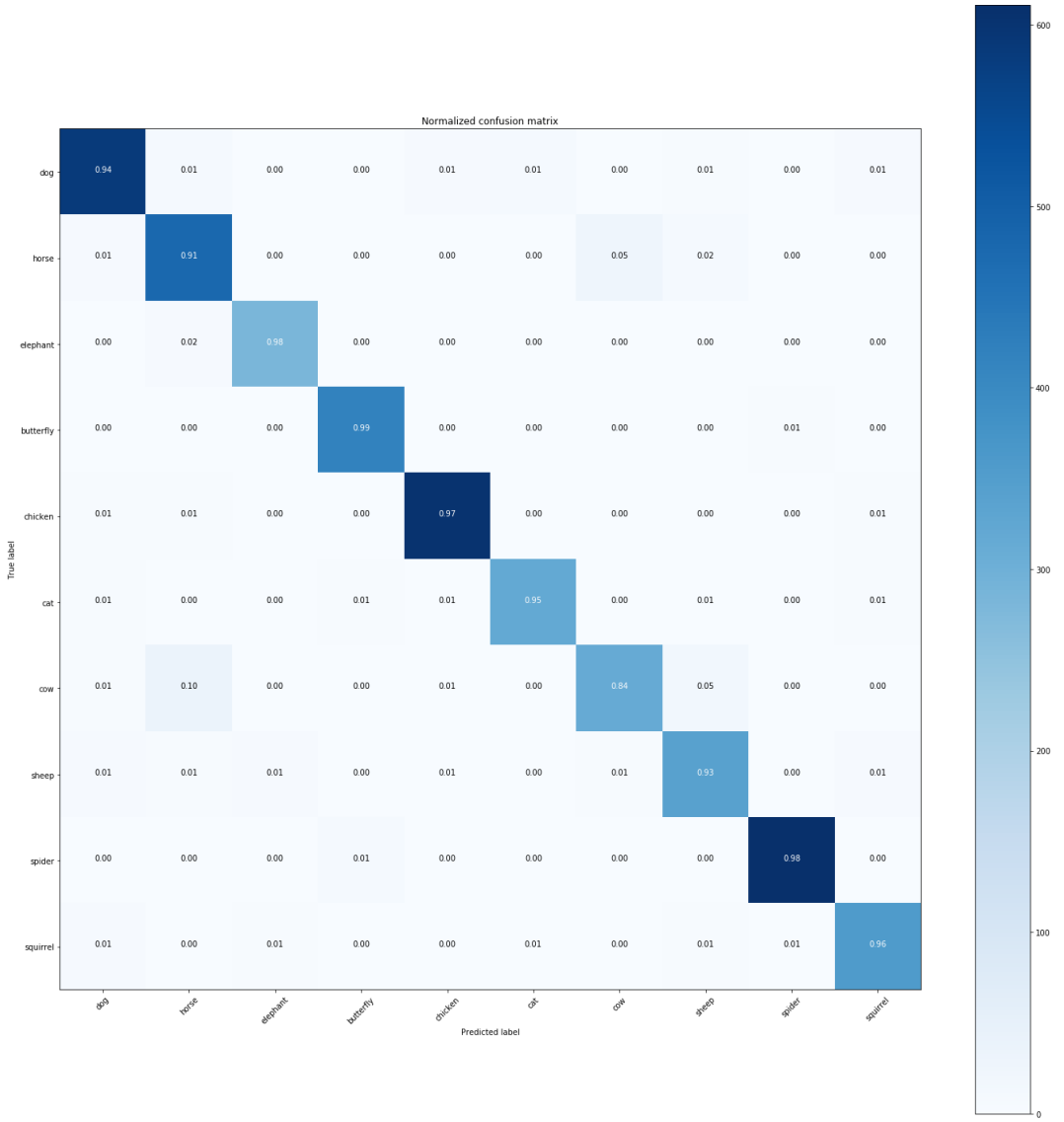


Figure 6: Confusion matrix

Overall accuracy of this model is : 93%
Overall precision of this model is: 93%
Overall recall of this model is: 93%

Model has high overall accuracy, precision and recall. Also, performance on each class is good too. Model is working successfully.

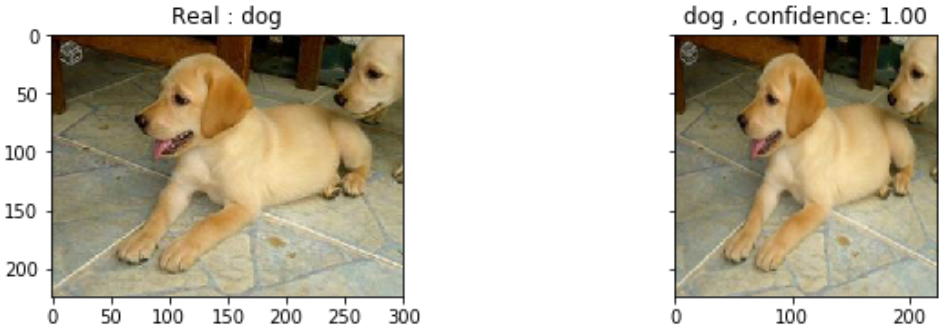


Figure 7: Normal image, preprocessed image and models prediction with its confidence

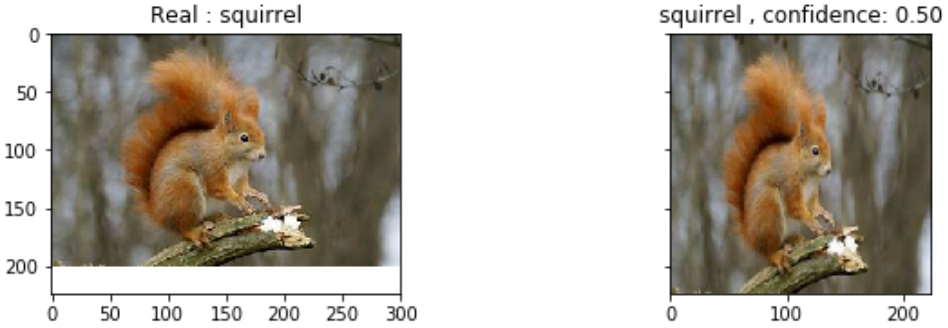


Figure 8: Normal image, preprocessed image and models prediction with its confidence

The images on the left are the original images from the dataset. The images on the right are the resized version of these images. Their sizes are 224px x 224 px x 3. Right image's title shows the model's prediction and its confidence. In figure 7, the model is 100% sure that the input picture is a dog. In figure 8, the model is only 50% sure that the input picture is a squirrel.

3.2. Linear SVM

It is popular to use feature descriptors to obtain features and use them as inputs in a SVM model. That is why I chose to use SVM. SVM's are also useful if kernel trick is used, with the kernel trick, SVM can capture non-linearities. However, I only implemented a Linear SVM. As the loss function, this model uses Hinge loss with L2 regularization [7]. Weights of the model is initialized randomly. As optimizer, model will use batch gradient descent where the batch size is equal to the size of the train set. I decided to use regularization to have a more robust model. The difference between Logistic Regression and Linear SVM is that Linear SVM wants to make a correct prediction with a margin. The downside of using Linear SVM and hinge loss with L2 regularization is that model can't give a confidence level of its prediction.

```
Starting to train.
-----------------
epoch: 1/30 , loss: 8.69 training accuracy: 0.16
epoch: 10/30 , loss: 0.41 training accuracy: 0.90
epoch: 20/30 , loss: 0.30 training accuracy: 0.93
epoch: 30/30 , loss: 0.28 training accuracy: 0.93
-----------------
time elapsed: 18.93 seconds
```

Figure 9: Training the linear SVM model

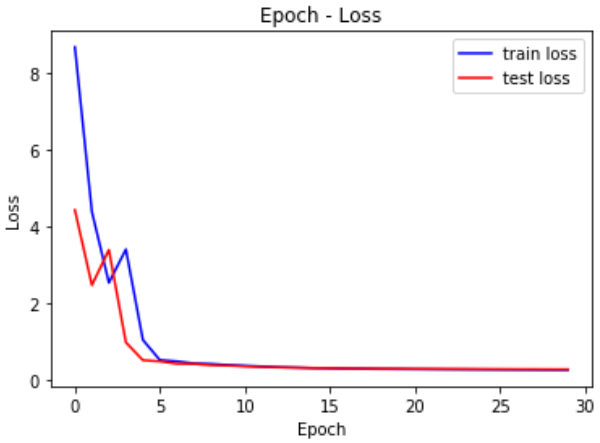


Figure 10: Loss history

Figure 9 and figure 10 shows training details when the learning rate is 0.001 and regularization coefficient is 0.00005. Training this model for 30 epochs took 18.93 seconds As you can see from figure 10, choosing 30 as an epoch number is not necessary. Since, after epoch 10, training loss rarely decreases. An epoch number of 10 would be more efficient and it would take around 6 seconds to train the model. Here are the results when the model is tested on the test set.

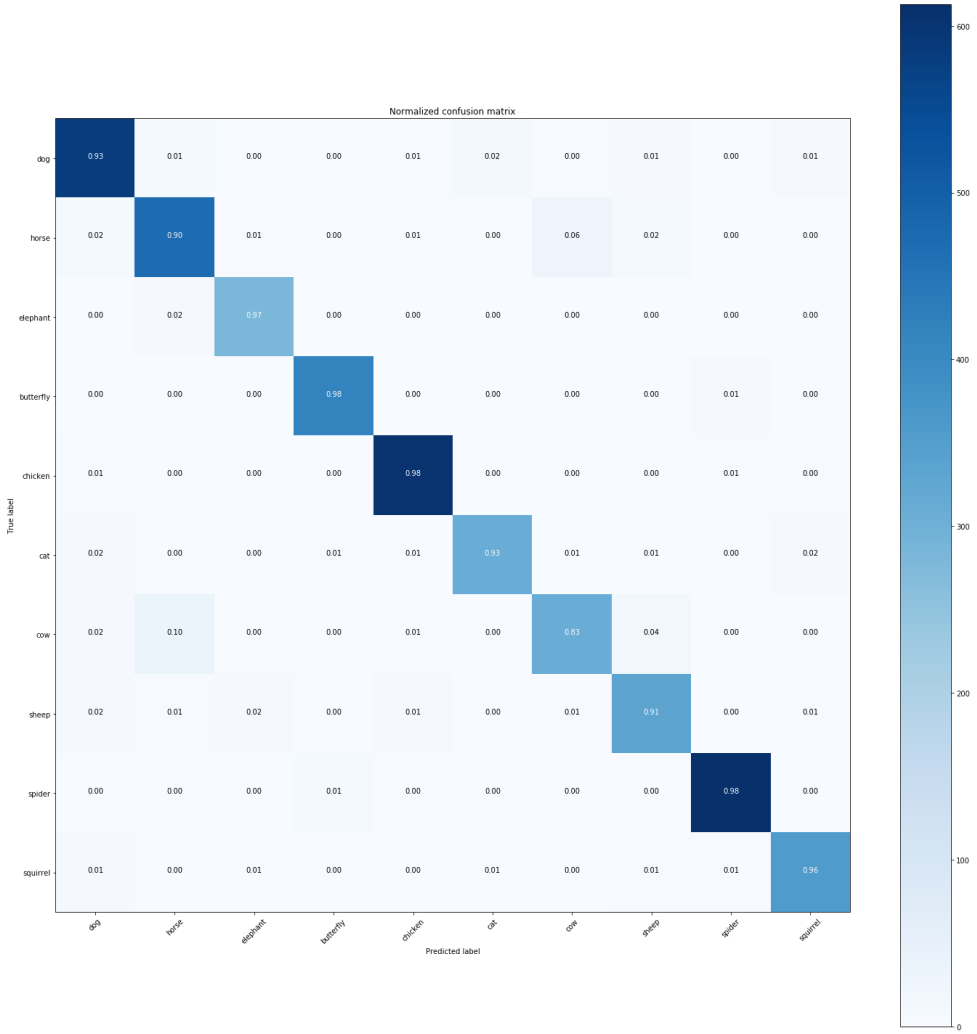


Figure 11: Confusion matrix

Overall accuracy of this model is : 94%
Overall precision of this model is: 93%
Overall recall of this model is: 93%

Model has high overall accuracy, precision and recall. Also, performance on each class is good too. Model is working successfully.

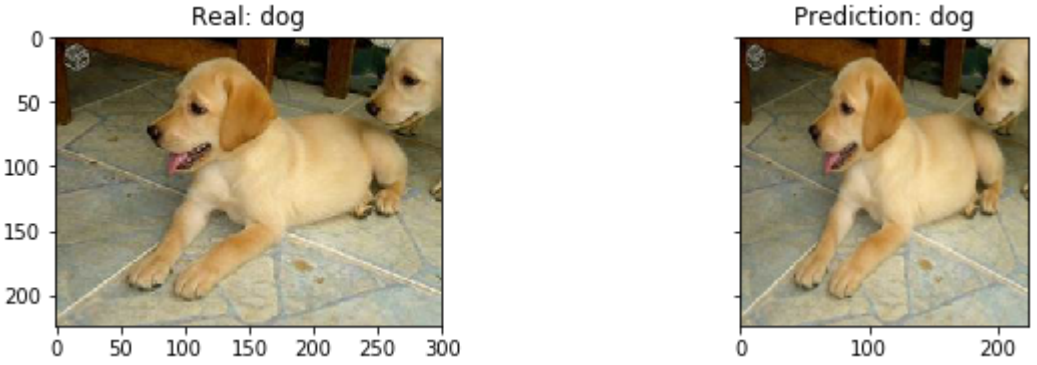


Figure 12: Normal image, preprocessed image and models prediction

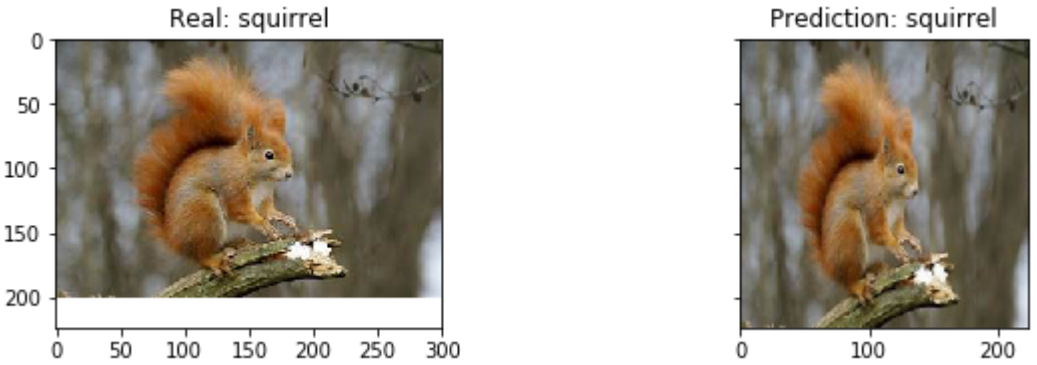


Figure 13: Normal image, preprocessed image and models prediction

The images on the left are the original images from the dataset. The images on the right are the resized version of these images. Their sizes are 224px x 224 px x 3. Right image's title shows the model's prediction. In figure 12, the model predicts that the input picture is a dog. In figure 13, the model predicts that the input picture is a squirrel.

3.3. Multilayer Perceptron

MLP's consist of input layers, hidden layers and output layers. MLP's also utilize activation functions. With non-linear activation functions, the model can capture nonlinearities [8]. In my MLP, the input layer has 4096 neurons since we got 4096 features from the VGG16 and the output layer consist of 10 neurons since we have 10 animal classes in our dataset. I used Stochastic Gradient Descent (SGD) as my optimizer [9]. At first, I used 2 hidden layers, random weight initialization with learning rate of 0.001 for 20 epochs. However, my model didn't converge. Loss didn't decrease at all. After that, I used grid-search to try multiple combinations of hidden layer and learning rate. None of those models converged.

```
Starting to train.
-----------------
epoch: 1/20, Loss: 2.28, training accuracy: 0.14
epoch: 2/20, Loss: 2.27, training accuracy: 0.14
epoch: 3/20, Loss: 2.27, training accuracy: 0.14
epoch: 4/20, Loss: 2.27, training accuracy: 0.14
epoch: 5/20, Loss: 2.27, training accuracy: 0.14
epoch: 6/20, Loss: 2.27, training accuracy: 0.14
epoch: 7/20, Loss: 2.27, training accuracy: 0.14
epoch: 8/20, Loss: 2.27, training accuracy: 0.14
epoch: 9/20, Loss: 2.27, training accuracy: 0.14
epoch: 10/20, Loss: 2.27, training accuracy: 0.14
epoch: 11/20, Loss: 2.27, training accuracy: 0.14
epoch: 12/20, Loss: 2.27, training accuracy: 0.14
epoch: 13/20, Loss: 2.27, training accuracy: 0.14
epoch: 14/20, Loss: 2.27, training accuracy: 0.14
epoch: 15/20, Loss: 2.27, training accuracy: 0.14
epoch: 16/20, Loss: 2.27, training accuracy: 0.14
epoch: 17/20, Loss: 2.27, training accuracy: 0.14
epoch: 18/20, Loss: 2.27, training accuracy: 0.14
epoch: 19/20, Loss: 2.27, training accuracy: 0.14
epoch: 20/20, Loss: 2.27, training accuracy: 0.14
-----------------
time elapsed: 2650.54 seconds
```

Figure 14: Training MLP (with random weight initialization)

Afterwards, I decided to try “He” weight initialization instead of random weight initialization in my model and it worked [10]. My model converged. Then, I tried multiple combinations of hidden layers and learning rates to find relatively better hidden layers and learning rate values. 3 hidden layers with 128, 64 and 32 neurons gave a good result.

```
Starting to train.
-----------------
epoch: 1/20, Loss: 1.92, training accuracy: 0.46
epoch: 2/20, Loss: 1.31, training accuracy: 0.51
epoch: 3/20, Loss: 0.87, training accuracy: 0.78
epoch: 4/20, Loss: 0.58, training accuracy: 0.81
epoch: 5/20, Loss: 0.43, training accuracy: 0.87
epoch: 6/20, Loss: 0.35, training accuracy: 0.88
epoch: 7/20, Loss: 0.30, training accuracy: 0.90
epoch: 8/20, Loss: 0.25, training accuracy: 0.93
epoch: 9/20, Loss: 0.19, training accuracy: 0.94
epoch: 10/20, Loss: 0.16, training accuracy: 0.94
epoch: 11/20, Loss: 0.13, training accuracy: 0.95
epoch: 12/20, Loss: 0.11, training accuracy: 0.96
epoch: 13/20, Loss: 0.10, training accuracy: 0.96
epoch: 14/20, Loss: 0.09, training accuracy: 0.97
epoch: 15/20, Loss: 0.08, training accuracy: 0.97
epoch: 16/20, Loss: 0.07, training accuracy: 0.97
epoch: 17/20, Loss: 0.07, training accuracy: 0.97
epoch: 18/20, Loss: 0.06, training accuracy: 0.98
epoch: 19/20, Loss: 0.05, training accuracy: 0.98
epoch: 20/20, Loss: 0.05, training accuracy: 0.98
-----------------
time elapsed: 2671.06 seconds
```

Figure 15: Training MLP (with He weight initialization)

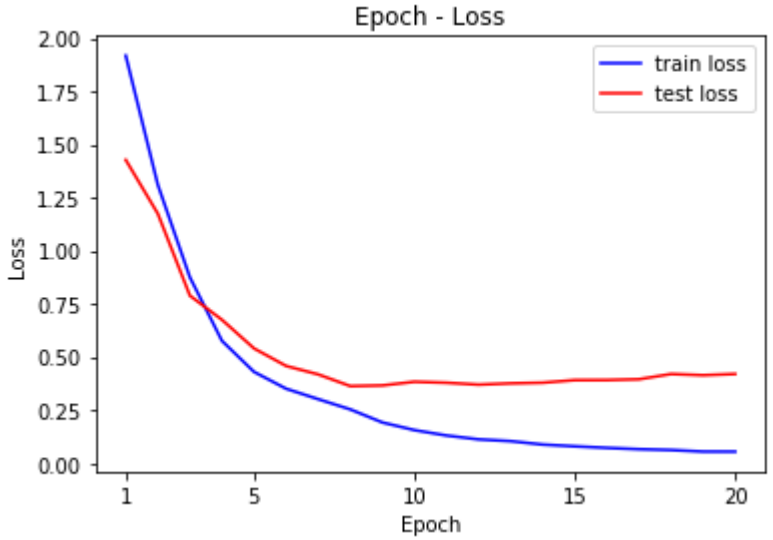


Figure 16: Loss History

Figure 15 and figure 16 shows training details when the model uses He weight initialization with the learning rate of 0.001 and there are 3 hidden layers with 128. 64 and 32 neurons. Training this model for 20 epochs took 2671.06 seconds. As you can see from figure 16, choosing 20 as an epoch number is not necessary. Since, after epoch 8, test loss doesn't decrease. An epoch number of 8 would be more efficient and it would take around 1068 seconds to train the model. Here are the results when the model is tested on the test set.

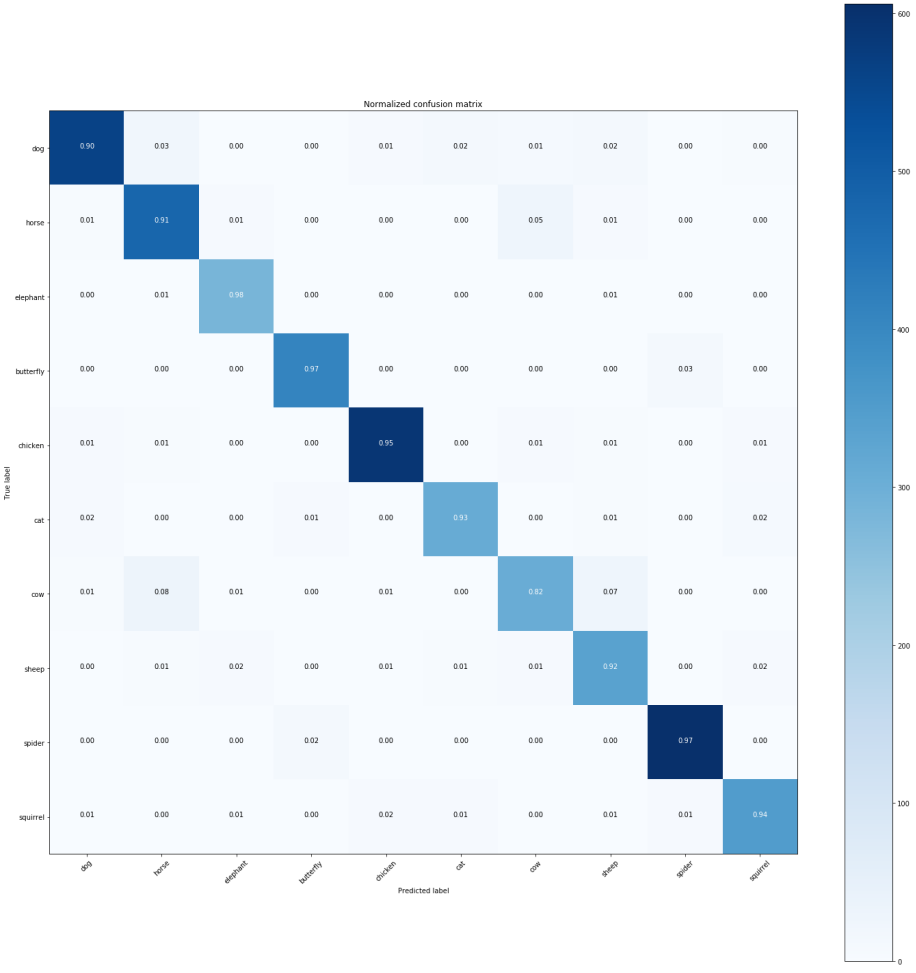


Figure 17: Confusion Matrix

Overall accuracy of this model is : 93%
Overall precision of this model is: 93%
Overall recall of this model is: 93%

Model has high overall accuracy, precision and recall. Also, performance on each class is good too. Model is working successfully.

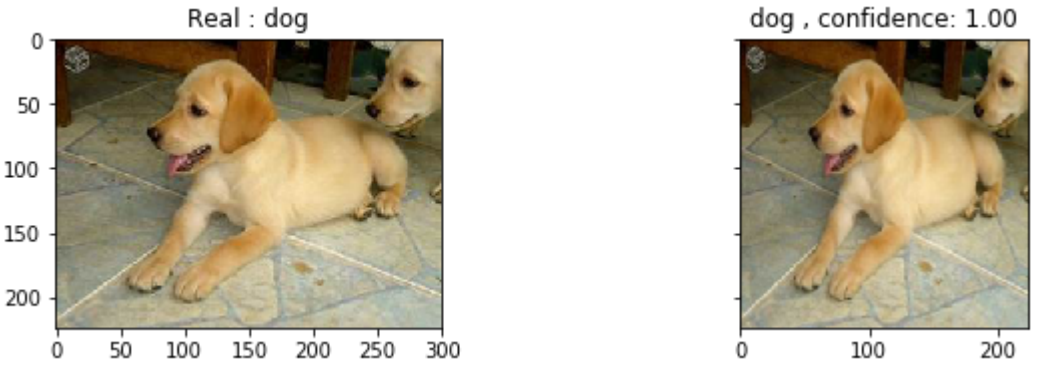


Figure 18: Normal image, preprocessed image and models prediction with its confidence

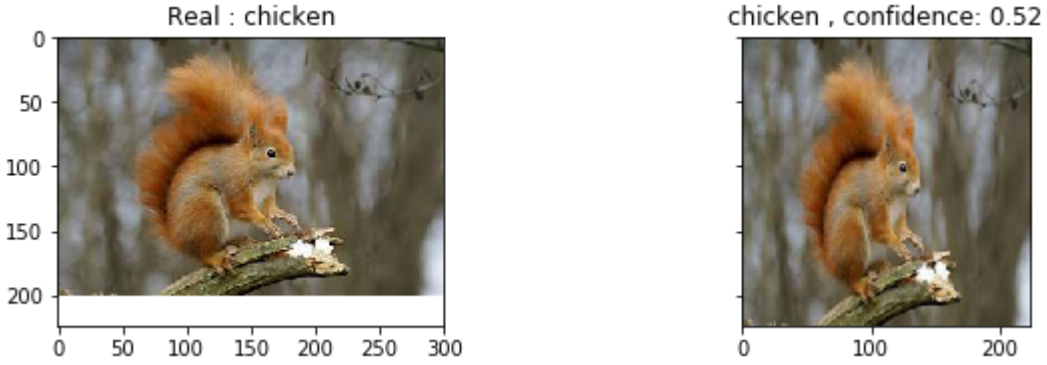


Figure 19: Normal image, preprocessed image and models prediction with its confidence

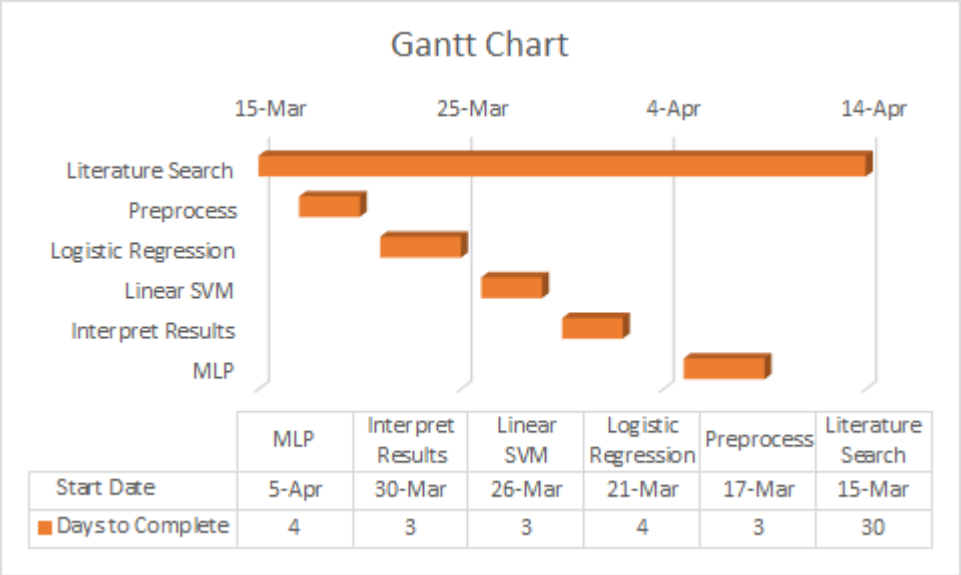
The images on the left are the original images from the dataset. The images on the right are the resized version of these images. Their sizes are 224px x 224 px x 3. Right image's title shows the model's prediction and its confidence. In figure 18, the model is 100% sure that the input picture is a dog. In figure 8, the model is only 52% sure that the input picture is a squirrel.

4. Conclusion

All of the models had similar results. Although, training the MLP model took around 2000 seconds while training the logistic model took around 50 seconds and training the linear SVM model took around 18 seconds. Normally, one would expect MLP model to have better results compared to logistic regression and linear SVM models. This wasn't the case for me. The reason for that is, I applied feature extraction to my images in the preprocessing part. If I haven't done that, the results of logistic regression and linear SVM models would be very poor and the result of MLP model would only be mediocre. After doing feature extraction, using linear SVM is the best choice since all of the models give similar results and linear SVM converges the fastest.

5. Division of Work

Nogay: Literature search, preprocessing data, implementing logistic regression, linear SVM, MLP and interpreting results.



	MLP	Interpret Results	Linear SVM	Logistic Regression	Preprocess	Literature Search
Start Date	5-Apr	30-Mar	26-Mar	21-Mar	17-Mar	15-Mar
Days to Complete	4	3	3	4	3	30

Figure 14: Gantt chart of the project

6. References

[1] C. Alessio, "Animals-10," *Kaggle*, 12-Dec-2019. [Online]. Available: https://www.kaggle.com/alessiocorrado99/animals10. [Accessed: 05-May-2021].

[2] *ImageNet*. [Online]. Available: http://image-net.org/download#:~:text=This%20dataset%20spans%201000%20object,subset%20is%20available%20on%20Kaggle. [Accessed: 05-May-2021].

[3] "Finetuning Torchvision Models," *Finetuning Torchvision Models - PyTorch Tutorials 1.2.0 documentation*. [Online]. Available: https://pytorch.org/tutorials/beginner/finetuning_torchvision_models_tutorial.html. [Accessed: 05-May-2021].

[4] "Advantages and Disadvantages of Logistic Regression," *GeeksforGeeks*, 02-Sep-2020. [Online]. Available: https://www.geeksforgeeks.org/advantages-and-disadvantages-of-logistic-regression/. [Accessed: 05-May-2021].

[5] *Softmax classification with cross-entropy*. [Online]. Available: https://peterroelants.github.io/posts/cross-entropy-softmax/. [Accessed: 05-May-2021].

[6] S. Patrikar, "Batch, Mini Batch & Stochastic Gradient Descent," *Medium*, 01-Oct-2019. [Online]. Available: https://towardsdatascience.com/batch-mini-batch-stochastic-gradient-descent-7a62ecba642a. [Accessed: 05-May-2021].

[7] *CS231n Convolutional Neural Networks for Visual Recognition*. [Online]. Available: https://cs231n.github.io/linear-classify/. [Accessed: 05-May-2021].

[8] DeepAI, “Multilayer Perceptron,” *DeepAI*, 17-May-2019. [Online]. Available: https://deepai.org/machine-learning-glossary-and-terms/multilayer-perceptron. [Accessed: 05-May-2021].
[9] A. V. Srinivasan, “Stochastic Gradient Descent - Clearly Explained !!,” *Medium*, 07-Sep-2019. [Online]. Available: https://towardsdatascience.com/stochastic-gradient-descent-clearly-explained-53d239905d31. [Accessed: 05-May-2021].
[10] V. Kakaraparthi, “Xavier and He Normal (He-et-al) Initialization,” *Medium*, 29-Sep-2018. [Online]. Available: https://prateekvishnu.medium.com/xavier-and-he-normal-he-et-al-initialization-8e3d7a087528. [Accessed: 05-May-2021].

7. Appendix

There are 4 files.

preprocessing.py (preprocessing.ipynb)

```
# coding: utf-8

# In[1]:


import os
import torch
import torch.nn as nn
import torchvision
from torchvision import transforms
import torchvision.models as models
from torchvision.datasets import ImageFolder
from torch.utils.data import Dataset, DataLoader
import matplotlib.pyplot as plt
import numpy as np
from collections import Counter
from tqdm.notebook import tqdm
from time import time


# In[2]:


device = torch.device("cpu")


# In[3]:


root_dir = os.getcwd()
dataset_dir = os.path.join(root_dir, "raw-img")


# In[4]:


fileToClass = {"cane": "dog",
               "cavallo": "horse",
               "elefante": "elephant",
               "farfalla": "butterfly",
               "gallina": "chicken",
               "gatto": "cat",
               "mucca": "cow",
```

```
                "pecora": "sheep",
                "ragno": "spider",
                "scoiattolo": "squirrel"}

classToNumber = {"cane": 0,
                 "cavallo": 1,
                 "elefante": 2,
                 "farfalla": 3,
                 "gallina": 4,
                 "gatto": 5,
                 "mucca": 6,
                 "pecora": 7,
                 "ragno": 8,
                 "scoiattolo": 9}

numberToClass = {
                 0: "dog",
                 1: "horse",
                 2: "elephant",
                 3: "butterfly",
                 4: "chicken",
                 5: "cat",
                 6: "cow",
                 7: "sheep",
                 8: "spider",
                 9: "squirrel"
}


# In[5]:


num_classes = len(fileToClass)
print(f"number of classes is : {num_classes}")


# In[6]:


model = models.vgg16(pretrained=True)


# In[7]:


model


# In[8]:


model.classifier = nn.Sequential(*list(model.classifier.children())[:-3])


# In[9]:


model


# In[10]:


input_size = 224
tensor_transform = transforms.Compose([
            transforms.Resize((input_size,input_size)),
            transforms.ToTensor()])

dataset = ImageFolder(dataset_dir, tensor_transform)

imgPaths = dataset.imgs
```

```
# In[11]:


print("sanity check")
counterDict = dict(Counter(dataset.targets))
print(counterDict)
print(f"number of class is : {len(counterDict)}")

noOfData = 0
for key in counterDict.keys():
    noOfData += counterDict[key]

print(f"total number of image is : {noOfData}")


# In[12]:


loader = DataLoader(dataset=dataset, shuffle=True, num_workers=os.cpu_count())

N_CHANNELS = 3

before = time()
mean = torch.zeros(3)
std = torch.zeros(3)
print('==> Computing mean and std..')
for inputs, _labels in tqdm(loader):
    for i in range(N_CHANNELS):
        mean[i] += inputs[:,i,:,:].mean()
        std[i] += inputs[:,i,:,:].std()
mean.div_(len(dataset))
std.div_(len(dataset))
print(mean, std)

print("time elapsed: ", time()-before)


# In[13]:


trainPercentage = 0.8
testPercentage = 1 - trainPercentage

trainCounts = []
testCounts = []
for key in counterDict.keys():
    number = int(trainPercentage * counterDict[key])
    trainCounts.append(number)
    testCounts.append(counterDict[key] - number)

print(trainCounts)
print(testCounts)


# In[14]:


classes = [numberToClass[i] for i in range(num_classes)]

fig = plt.figure(figsize = (20, 5))

plt.subplot(1, 2, 1)
# creating the bar plot
plt.bar(classes, trainCounts, color ='maroon',
        width = 0.4)

plt.xlabel("Classes")
plt.ylabel("No. of images")
plt.title("Images per class for train set")

plt.subplot(1, 2, 2)
```

```
plt.bar(classes, testCounts, color ='maroon',
        width = 0.4)

plt.xlabel("Classes")
plt.ylabel("No. of images")
plt.title("Images per class for test set")

plt.show()


# In[15]:


# trim from 2500 to make it more balanced.

newTrainCounts = []
newTestCounts = []
trimValue = 2500
for key in counterDict.keys():
    number = int(trainPercentage * counterDict[key])
    if number > trimValue:
        newTrainCounts.append(trimValue)
        newTestCounts.append(int(trimValue * (testPercentage / trainPercentage)))

    else:
        newTrainCounts.append(number)
        newTestCounts.append(counterDict[key] - number)

print(newTrainCounts)
print(newTestCounts)


# In[16]:


mean = [0.5177, 0.5003, 0.4126]
std = [0.2135, 0.2130, 0.2151]

tensor_transform = transforms.Compose([
            transforms.Resize((input_size,input_size)),
            transforms.ToTensor(),
            transforms.Normalize(torch.Tensor(mean),
                                 torch.Tensor(std))])

dataset = ImageFolder(dataset_dir, tensor_transform)

loader = DataLoader(dataset=dataset, batch_size = 1, shuffle=False, num_workers=os.cpu_count())


# In[17]:


data = np.empty(shape=[noOfData, 4096])
labels = np.empty(shape=[noOfData, 1])

with torch.no_grad():
    for i,(image, label) in enumerate(tqdm(loader)):
        output = model(image)
        data[i] = output.detach().numpy()
        labels[i] = label


print(data.shape)
print(labels.shape)


# In[18]:


with open('data.npy', 'wb') as f:
    np.save(f, data)
```

```
with open('labels.npy', 'wb') as f:
    np.save(f, labels)


# In[19]:


trainX = np.empty(shape=[sum(newTrainCounts), 4096])
trainY = np.empty(shape=[sum(newTrainCounts), 1])
testX = np.empty(shape=[sum(newTestCounts), 4096])
testY = np.empty(shape=[sum(newTestCounts), 1])

trainPaths = []
testPaths = []
pathIndex = 0

trainIndex = 0
testIndex = 0
numberOfTrainSamples = [0 for i in range(num_classes)]
numberOfTestSamples = [0 for i in range(num_classes)]

for i in range(len(data)):
    label = int(labels[i])
    if numberOfTrainSamples[label] == newTrainCounts[label]:
        if numberOfTestSamples[label] != newTestCounts[label]:
            testX[testIndex] = data[i]
            testY[testIndex] = labels[i]

            testIndex += 1

            numberOfTestSamples[label] += 1

            testPaths.append(imgPaths[pathIndex][0])

    else:
        trainX[trainIndex] = data[i]
        trainY[trainIndex] = labels[i]

        trainIndex += 1

        numberOfTrainSamples[label] += 1

        trainPaths.append(imgPaths[pathIndex][0])


    pathIndex += 1


# In[20]:


(uniqueTrain, countsTrain) = np.unique(trainY, return_counts=True)
print(f"frequency count of trainY {countsTrain}")
print(f"frequency count of wanted trainY {newTrainCounts}")
print("--------------")
(uniqueTest, countsTest) = np.unique(testY, return_counts=True)
print(f"frequency count of testy {countsTest}")
print(f"frequency count of wanted testY {newTestCounts}")


# In[21]:


for i in range(len(countsTrain)):
    allData = countsTrain[i] + countsTest[i]
    print(f"distribution of class {i} - train : {countsTrain[i]/allData:.2f} , test :
{countsTest[i]/allData:.2f}")


# In[22]:
```

```
classes = [numberToClass[i] for i in range(num_classes)]

fig = plt.figure(figsize = (20, 5))

plt.subplot(1, 2, 1)
# creating the bar plot
plt.bar(classes, countsTrain, color ='maroon',
        width = 0.4)

plt.xlabel("Classes")
plt.ylabel("No. of images")
plt.title("Images per class for train set")

plt.subplot(1, 2, 2)
plt.bar(classes, countsTest, color ='maroon',
        width = 0.4)

plt.xlabel("Classes")
plt.ylabel("No. of images")
plt.title("Images per class for test set")

plt.show()

print(f"number of images in train set: {trainX.shape[0]}")
print(f"number of images in test set: {testX.shape[0]}")


# In[38]:


with open('trainX.npy', 'wb') as f:
    np.save(f, trainX)

with open('trainY.npy', 'wb') as f:
    np.save(f, trainY)

with open('testX.npy', 'wb') as f:
    np.save(f, testX)

with open('testY.npy', 'wb') as f:
    np.save(f, testY)

with open("testPaths.txt", "w") as f:
    for i in range(len(testPaths)):
        f.write(testPaths[i] + "\n")

with open("trainPaths.txt", "w") as f:
    for i in range(len(trainPaths)):
        f.write(trainPaths[i] + "\n")
```

logistic-regression.py (logistic-regression.ipynb)

```
# coding: utf-8

# In[1]:


import numpy as np
import matplotlib.pyplot as plt
from time import time
import itertools
import os
import random
from PIL import Image
```

```
# to see the resized version of images
import torch
import torch.nn as nn
import torchvision
from torchvision import transforms
import torchvision.models as models
from torchvision.datasets import ImageFolder
from torch.utils.data import Dataset, DataLoader


# In[2]:


trainX = np.load('trainX.npy')
trainY = np.load('trainY.npy')
testX = np.load('testX.npy')
testY = np.load('testY.npy')

trainY = trainY.astype(int)
testY = testY.astype(int)

with open("testPaths.txt", "r") as f:
    testPaths = f.read().strip().split("\n")

with open("trainPaths.txt", "r") as f:
    trainPaths = f.read().strip().split("\n")

print(f"shape of trainX is : {trainX.shape}")
print(f"shape of trainY is : {trainY.shape}")
print(f"shape of testX is : {testX.shape}")
print(f"shape of testY is : {testY.shape}")
print(f"length of trainPaths is : {len(trainPaths)}")
print(f"length of testPaths is : {len(testPaths)}")


# In[3]:


class LogisticRegression:

    def __init__(self, lr, epochs):
        self.lr = lr
        self.epochs = epochs
        self.weights = None
        self.bias = None

    def fit(self, features, labels, test_features, test_labels):

        numberOfSamples , numberOfFeatures = features.shape
        numberOfClasses = len(np.unique(labels))


        self.weights = 0.01 * np.random.rand(numberOfFeatures, numberOfClasses)
        self.bias = 0.01 * np.random.rand(numberOfClasses) * 0.01

        targets = np.zeros(shape = [numberOfSamples, numberOfClasses])
        targets[np.arange(numberOfSamples), labels.T] = 1

        test_targets = np.zeros(shape = [len(test_features), numberOfClasses])
        test_targets[np.arange(len(test_features)), test_labels.T] = 1

        print("Starting to train.")
        print("------------------")
        loss_history_training = []
        loss_history_test = []

        for i in range(self.epochs):

            scores = np.dot(features, self.weights) + self.bias

            outputs = self.softmax(scores)
```

```
            d_error = outputs - targets

            dw = (1 / numberOfSamples) * np.dot(features.T, d_error)
            db = (1 / numberOfSamples) * np.sum(d_error, axis = 0)

            self.weights -= self.lr * dw
            self.bias -= self.lr * db

            loss = np.sum(self.categorical_cross_entropy_loss(targets, outputs))


            test_scores = np.dot(test_features, self.weights) + self.bias
            test_outputs = self.softmax(test_scores)
            test_loss = np.sum(self.categorical_cross_entropy_loss(test_targets, test_outputs))

            loss_history_training.append(loss/numberOfSamples)
            loss_history_test.append(test_loss/len(test_features))

            if i == 0 or (i - 9) % 10 == 0:
                print(f"epoch: {i+1}/{self.epochs} , loss: {loss/numberOfSamples:.2f}, training accuracy:
{self.evaluate(targets, outputs):.2f}")

        return loss_history_training, loss_history_test

    # predict 1 sample
    def predict(self, features):
        scores = np.dot(features, self.weights) + self.bias
        probabilities = self.softmax(scores)
        label = np.argmax(probabilities)
        confidence = probabilities[0][label]
        return label, confidence

    def softmax(self, x):
        e_x = np.exp(x - np.max(x))
        return e_x / e_x.sum(axis=1, keepdims=True)

    def categorical_cross_entropy_loss(self, targets, outputs):
        return -np.sum(targets * np.log(outputs), axis = 1)

    def evaluate(self, targets, outputs):
        return np.mean(np.argmax(targets, axis=1) == np.argmax(outputs, axis=1))



# In[4]:


learning_rate = 0.001
epochs = 100

model = LogisticRegression(learning_rate, epochs)
before = time()
history_train, history_test = model.fit(trainX, trainY, testX, testY)

print("-----------------")
print(f"time elapsed: {time() - before:.2f} seconds ")


# In[5]:


plt.plot(history_train, "-b", label="train loss")
plt.plot(history_test, "-r", label="test loss")
plt.title("Epoch - Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend(loc="upper right")


# In[6]:
```

```
# test
classes = np.unique(trainY)
conf_matrix = np.zeros(shape=(len(classes),len(classes)))
for i, (features, label) in enumerate(zip(testX, testY)):
    rowMatrix = np.reshape(features, (1, len(features)))
    label = label[0]

    prediction, _ = model.predict(rowMatrix)
    conf_matrix[label][prediction] += 1


# In[7]:


acc = correct / len(testX)
recall = np.mean(np.diag(conf_matrix) / np.sum(conf_matrix, axis = 1))
precision = np.mean(np.diag(conf_matrix) / np.sum(conf_matrix, axis = 0))

print(f"overall accuracy is {acc:.2f}")
print(f"overall precision is {precision:.2f}")
print(f"overall recall is {recall:.2f}")


# In[8]:


# from scikitlearn 0.18 - old version
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')


    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, "{:0.2f}".format(cm[i, j]),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')


# In[9]:


class_names = ["dog", "horse", "elephant", "butterfly", "chicken", "cat", "cow", "sheep", "spider","squirrel"]

plt.figure()
plt.figure(figsize=(20,20))

plot_confusion_matrix(conf_matrix, classes=class_names, normalize=True,
                      title='Normalized confusion matrix')

plt.show()
```

```
# In[10]:


# to free up memory
del trainX
del trainY


# In[11]:


vggModel = models.vgg16(pretrained=True)
vggModel.classifier = nn.Sequential(*list(vggModel.classifier.children())[:-3])
input_size = 224
mean = [0.5177, 0.5003, 0.4126]
std = [0.2135, 0.2130, 0.2151]
tensor_transform = transforms.Compose([
            transforms.Resize((input_size,input_size)),
            transforms.ToTensor()])


# In[12]:


numberToClass = {
                 0: "dog",
                 1: "horse",
                 2: "elephant",
                 3: "butterfly",
                 4: "chicken",
                 5: "cat",
                 6: "cow",
                 7: "sheep",
                 8: "spider",
                 9: "squirrel"
}


# In[13]:


testNumber = 20
for i in range(int(testNumber/2)):
    randIndex1 = random.randint(0, len(testX) -1)
    randIndex2 = random.randint(0, len(testX) -1)


    im1 = Image.open(testPaths[randIndex1]).convert("RGB")
    im2 = Image.open(testPaths[randIndex2]).convert("RGB")


    features1 = testX[randIndex1]
    features1 = np.reshape(features1, (1, len(features1)))

    features2 = testX[randIndex2]
    features2 = np.reshape(features2, (1, len(features2)))

    label1 = testY[randIndex1][0]
    label2 = testY[randIndex2][0]



    prediction1, confidence1 = model.predict(features1)
    prediction2, confidence2 = model.predict(features2)


    f, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
    plt.subplots_adjust(left=0.0, bottom=0, right=1.8, top=0.5,wspace=0.3, hspace=0.3)

    ax1.imshow(im1)
```

```
    ax2.imshow(im2)
    ax1.set_title(f"Real-{numberToClass[label1]}, Prediction-{numberToClass[prediction1]}, confidence:
{confidence1:.2f}")
    ax2.set_title(f"Real-{numberToClass[label2]}, Prediction-{numberToClass[prediction2]} , confidence:
{confidence2:.2f}")

    plt.show()


# In[14]:


os.chdir("D:\\python\\animal classifier")


root_dir = os.getcwd()
testDir = os.path.join(root_dir, "plotImgs")
os.chdir(testDir)

file = open("labels.txt", "r")
labels = file.read().strip().split("-")

with torch.no_grad():
    for i in range(len(os.listdir())): # 1 file is txt file
        if f"{i+1}.jpg" in os.listdir():
            path = f"{i+1}.jpg"

        elif f"{i+1}.jpeg" in os.listdir():
            path = f"{i+1}.jpeg"

        else:
            continue

        imgPIL = Image.open(path) #PIL image
        imgTensor = tensor_transform(imgPIL) #torch.FloatTensor 3,224,224
        features = vggModel(imgTensor[None, ...])
        features = features.detach().numpy()

        prediction, confidence = model.predict(features)

        f, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
        plt.subplots_adjust(left=0.0, bottom=0, right=1.3, top=0.5,wspace=0.3, hspace=0.3)

        ax1.imshow(imgPIL)
        ax2.imshow( imgTensor.permute(1, 2, 0) )
        ax1.set_title(f"Real : {numberToClass[prediction]}")
        ax2.set_title(f"{numberToClass[prediction]} , confidence: {confidence:.2f}")

        plt.show()
```

linear-SVM.py (linear-SVM.ipynb)

```
# coding: utf-8

# In[1]:


import numpy as np
import matplotlib.pyplot as plt
from time import time
import itertools
import os
import random

# to plot some images
```

```
import torch
import torch.nn as nn
import torchvision
from torchvision import transforms
import torchvision.models as models
from torchvision.datasets import ImageFolder
from torch.utils.data import Dataset, DataLoader
from PIL import Image


# In[2]:


trainX = np.load('trainX.npy')
trainY = np.load('trainY.npy')
trainY = trainY.flatten()
testX = np.load('testX.npy')
testY = np.load('testY.npy')
testY = testY.flatten()

trainY = trainY.astype(int)
testY = testY.astype(int)

with open("testPaths.txt", "r") as f:
    testPaths = f.read().strip().split("\n")

with open("trainPaths.txt", "r") as f:
    trainPaths = f.read().strip().split("\n")


print(f"shape of trainX is : {trainX.shape}")
print(f"shape of trainY is : {trainY.shape}")
print(f"shape of testX is : {testX.shape}")
print(f"shape of testY is : {testY.shape}")
print(f"length of trainPaths is : {len(trainPaths)}")
print(f"length of testPaths is : {len(testPaths)}")


# In[3]:


trainX = np.hstack([trainX, np.ones((trainX.shape[0], 1))])
testX = np.hstack([testX, np.ones((testX.shape[0], 1))])

print(f"shape of trainX is : {trainX.shape}")
print(f"shape of testX is : {testX.shape}")


# In[4]:


class LinearSVM(object):

    def __init__(self, lr, epochs, delta=1, reg=1e-5):
        self.lr = lr
        self.epochs = epochs
        self.weights = None
        self.delta = 1
        self.reg = reg

        self.numberOfSamples = None
        self.numberOfFeatures = None
        self.numberOfClasses = None


    def fit(self, features, labels, test_features, test_labels):
        self.numberOfSamples = features.shape[0]
        self.numberOfFeatures = features.shape[1]
        self.numberOfClasses = len(np.unique(labels))

        self.weights = 0.01 * np.random.rand(self.numberOfFeatures, self.numberOfClasses)
```

```
        print("Starting to train.")
        print("-----------------")
        loss_history_training = []
        loss_history_test = []

        for i in range(self.epochs):

            loss, dW, outputs = self.loss(features, labels, self.reg)

            self.weights = self.weights - self.lr * dW

            loss_history_training.append(loss)

            test_loss, _ , _ = self.loss(test_features, test_labels, self.reg)
            loss_history_test.append(test_loss)

            if i == 0 or (i - 9) % 10 == 0:
                print(f"epoch: {i+1}/{self.epochs} , loss: {loss:.2f} training accuracy: {self.evaluate(labels,
outputs):.2f}")

        return loss_history_training, loss_history_test


    def loss(self, features, labels, reg):
        loss = 0.0
        dW = np.zeros(self.weights.shape)

        scores = np.dot(features,self.weights)

        correctClassScores = scores[ np.arange(len(features)), labels].reshape(len(features),1)

        margin = np.maximum(0, scores - correctClassScores + self.delta)

        margin[np.arange(len(features)), labels] = 0

        loss = margin.sum() / len(features)

        # l2 regularization
        loss += self.reg * np.sum(self.weights * self.weights)

        # now, calculate the gradients.
        margin[margin > 0] = 1

        validMarginCount = margin.sum(axis=1)

        margin[np.arange(len(features)),labels ] -= validMarginCount

        dW = np.dot(features.T, margin) / len(features)

        # regularization gradient
        dW = dW + self.reg * 2 * self.weights

        return loss, dW, np.argmax(scores, axis=1)

    # predict 1 sample
    def predict(self, features):
        scores = np.dot(features, self.weights)
        label = np.argmax(scores)
        return label

    def evaluate(self, targets, outputs):
        return np.mean(targets == outputs)


# In[5]:


learning_rate = 0.001
epochs = 30

model = LinearSVM(learning_rate, epochs, delta=1, reg=5*1e-5)
before = time()
```

```
history_train, history_test = model.fit(trainX, trainY, testX, testY)

print("-----------------")
print(f"time elapsed: {time()-before:.2f} seconds ")


# In[6]:


plt.plot(history_train, "-b", label="train loss")
plt.plot(history_test, "-r", label="test loss")
plt.title("Epoch - Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend(loc="upper right")


# In[7]:


# test
classes = np.unique(trainY)
conf_matrix = np.zeros(shape=(len(classes),len(classes)))
for i, (features, label) in enumerate(zip(testX, testY)):
    rowMatrix = np.reshape(features, (1, len(features)))

    prediction = model.predict(rowMatrix)
    conf_matrix[label][prediction] += 1


# In[8]:


acc = correct / len(testX)
recall = np.mean(np.diag(conf_matrix) / np.sum(conf_matrix, axis = 1))
precision = np.mean(np.diag(conf_matrix) / np.sum(conf_matrix, axis = 0))

print(f"overall accuracy is {acc:.2f}")
print(f"overall precision is {precision:.2f}")
print(f"overall recall is {recall:.2f}")


# In[9]:


# from scikitlearn 0.18 - old version
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')


    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, "{:0.2f}".format(cm[i, j]),
                 horizontalalignment="center",
```

```
                color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')


# In[10]:


class_names = ["dog", "horse", "elephant", "butterfly", "chicken", "cat", "cow", "sheep", "spider","squirrel"]

plt.figure()
plt.figure(figsize=(20,20))

plot_confusion_matrix(conf_matrix, classes=class_names, normalize=True,
                      title='Normalized confusion matrix')

plt.show()


# In[11]:


# to free up memory
del trainX
del trainY


# In[12]:


numberToClass = {
                0: "dog",
                1: "horse",
                2: "elephant",
                3: "butterfly",
                4: "chicken",
                5: "cat",
                6: "cow",
                7: "sheep",
                8: "spider",
                9: "squirrel"
}


# In[14]:


testNumber = 20
for i in range(int(testNumber/2)):
    randIndex1 = random.randint(0, len(testX) -1)
    randIndex2 = random.randint(0, len(testX) -1)


    im1 = Image.open(testPaths[randIndex1]).convert("RGB")
    im2 = Image.open(testPaths[randIndex2]).convert("RGB")


    features1 = testX[randIndex1]

    features2 = testX[randIndex2]

    label1 = testY[randIndex1]
    label2 = testY[randIndex2]



    prediction1 = model.predict(features1)
    prediction2 = model.predict(features2)
```

```
    f, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
    plt.subplots_adjust(left=0.0, bottom=0, right=1.8, top=0.5,wspace=0.3, hspace=0.3)

    ax1.imshow(im1)
    ax2.imshow(im2)
    ax1.set_title(f"Real: {numberToClass[label1]}, Prediction: {numberToClass[prediction1]}")
    ax2.set_title(f"Real: {numberToClass[label2]}, Prediction: {numberToClass[prediction2]}")

    plt.show()


# In[ ]:


vggModel = models.vgg16(pretrained=True)
vggModel.classifier = nn.Sequential(*list(vggModel.classifier.children())[:-3])
input_size = 224
mean = [0.5177, 0.5003, 0.4126]
std = [0.2135, 0.2130, 0.2151]
tensor_transform = transforms.Compose([
            transforms.Resize((input_size,input_size)),
            transforms.ToTensor()])


# In[ ]:


os.chdir("D:\\python\\animal classifier")



root_dir = os.getcwd()
testDir = os.path.join(root_dir, "plotImgs")
os.chdir(testDir)

file = open("labels.txt", "r")
labels = file.read().strip().split("-")

with torch.no_grad():
    for i in range(len(os.listdir())): # 1 file is txt file
        if f"{i+1}.jpg" in os.listdir():
            path = f"{i+1}.jpg"

        elif f"{i+1}.jpeg" in os.listdir():
            path = f"{i+1}.jpeg"

        else:
            continue

        imgPIL = Image.open(path) #PIL image
        imgTensor = tensor_transform(imgPIL) #torch.FloatTensor 3,224,224
        features = vggModel(imgTensor[None, ...])
        features = features.detach().numpy()

        features = np.hstack([features, np.ones((features.shape[0], 1))])

        prediction = model.predict(features)

        f, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
        plt.subplots_adjust(left=0.0, bottom=0, right=1.3, top=0.5,wspace=0.3, hspace=0.3)

        ax1.imshow(imgPIL)
        ax2.imshow( imgTensor.permute(1, 2, 0) )
        ax1.set_title(f"Real: {numberToClass[prediction]}")
        ax2.set_title(f"Prediction: {numberToClass[prediction]}")

        plt.show()
```

MLP.py (MLP.ipynb)

```
# coding: utf-8

# In[1]:


import numpy as np
import matplotlib.pyplot as plt
from time import time
import itertools
import os
import random
from math import sqrt
from numpy.random import randn

# to plot some images
import torch
import torch.nn as nn
import torchvision
from torchvision import transforms
import torchvision.models as models
from torchvision.datasets import ImageFolder
from torch.utils.data import Dataset, DataLoader
from PIL import Image
from sklearn.utils import shuffle


# In[2]:


trainX = np.load('trainX.npy')
trainY = np.load('trainY.npy')
trainY = trainY.flatten()
testX = np.load('testX.npy')
testY = np.load('testY.npy')
testY = testY.flatten()

trainY = trainY.astype(int)
testY = testY.astype(int)

trainX, trainY = shuffle(trainX, trainY)
with open("testPaths.txt", "r") as f:
    testPaths = f.read().strip().split("\n")

with open("trainPaths.txt", "r") as f:
    trainPaths = f.read().strip().split("\n")



print(f"shape of trainX is : {trainX.shape}")
print(f"shape of trainY is : {trainY.shape}")
print(f"shape of testX is : {testX.shape}")
print(f"shape of testY is : {testY.shape}")
print(f"length of trainPaths is : {len(trainPaths)}")
print(f"length of testPaths is : {len(testPaths)}")


# In[3]:


class MLP(object):
    def __init__(self, num_inputs, num_hidden, num_outputs, weight_init):
        self.num_inputs = num_inputs
        self.num_hidden = num_hidden
        self.num_outputs = num_outputs
        self.weight_init = weight_init

        self.layers = [self.num_inputs] + self.num_hidden + [self.num_outputs]
```

```
        self.activations = []
        self.weights = []
        self.biases = []
        self.d_weights = []
        self.d_biases = []

        # initialize random weights
        # create derivatives for weights
        # initialize random biases
        # create derivatives for biases

        for i in range(len(self.layers) - 1):

            if self.weight_init == "he":
                weights = np.random.randn(self.layers[i], self.layers[i + 1]) * np.sqrt(2./self.layers[i + 1])
                self.weights.append(weights)

                biases = np.zeros(self.layers[i + 1])
                self.biases.append(biases)

                self.d_weights.append(np.zeros((self.layers[i] , self.layers[i + 1])))
                self.d_biases.append(np.zeros(self.layers[i + 1]))

            elif "random":
                weights =  0.01 *  np.random.rand(self.layers[i], self.layers[i + 1])
                self.weights.append(weights)

                self.d_weights.append(np.zeros((self.layers[i] , self.layers[i + 1])))

                biases = 0.01 * np.random.rand(self.layers[i + 1])
                self.biases.append(biases)

                self.d_biases.append(np.zeros(self.layers[i + 1]))


        # create activations per layer
        for i in range(len(self.layers)):
            activation = np.zeros(self.layers[i])
            self.activations.append(activation)


    def forward(self, inputs):
        self.activations[0] = inputs
        activations = inputs

        for i, weights in enumerate(self.weights):
            z = np.dot(activations, weights) + self.biases[i]

            if i != len(self.weights) - 1:
                activations = self.relu(z)
            else:
                activations = self.softmax(z)

            self.activations[i + 1] = activations

        return activations # output


    def train(self, inputs, labels, learningRate, epochs, test_inputs, test_labels):
        loss_history_training = []
        loss_history_test = []

        print("Starting to train.")
        print("------------------")
        for i in range(epochs):
            sumError_train = 0
            sumError_test = 0
            for j,data in enumerate(inputs):
                label = labels[j]


                target = np.zeros(self.num_outputs)
```

```
                target[label] = 1

                output = self.forward(data)
                d_error = output - target

                self.backpropagate(d_error)

                self.gradient_descent(learningRate)

                sumError_train += self.categorical_cross_entropy_loss(target, output)

            for j,data in enumerate(test_inputs):
                test_label = test_labels[j]

                test_target = np.zeros(self.num_outputs)
                test_target[test_label] = 1

                test_output = self.forward(data)

                sumError_test += self.categorical_cross_entropy_loss(test_target, test_output)


            loss_history_training.append(sumError_train / len(inputs))
            loss_history_test.append(sumError_test / len(test_inputs))

            print(f"epoch: {i+1}/{epochs}, Loss: {sumError_train/len(inputs):.2f}, training accuracy:
{self.evaluate(inputs, labels):.2f} ")


        return loss_history_training, loss_history_test

    def backpropagate(self, d_error):
        # dC / dZ = output(k) - target(k)  categorical cross entropy loss
        # dC/dW(l) = dC/dA * dA/dZ * dZ/dW(l-1)
        # dC/db(l) = dC/dA * dA/dZ * 1
        # dC/dW(l-1) = dC/dA * dA/dZ * dZ(l)/dA(l-1) * dA(l-1)/dZ(l-1) * dZ(l-1)/dW(l-1)
        for i in reversed(range(len(self.d_weights))):

            activations = self.activations[i + 1]

            if i == len(self.d_weights) - 1:
                commonDerivative = d_error
            else:
                commonDerivative = d_error * self.d_relu(activations)

            re_commonDerivative = commonDerivative.reshape(commonDerivative.shape[0], -1).T # 2d, 1 row

            # get activations for current layer
            current_activations = self.activations[i]
            # reshape activations as to have them as a 2d column matrix
            current_activations = current_activations.reshape(current_activations.shape[0],-1) #2d, 1 col

            self.d_weights[i] = np.dot(current_activations, re_commonDerivative)
            self.d_biases[i] = commonDerivative

            # backpropogate the error - the common part
            d_error = np.dot(commonDerivative, self.weights[i].T)

    def gradient_descent(self, learningRate):
        for i in range(len(self.d_weights)):
            d_weights = self.d_weights[i]
            self.weights[i] = self.weights[i] - d_weights * learningRate

            d_biases = self.d_biases[i]
            self.biases[i] = self.biases[i] - d_biases * learningRate

    def relu(self, x):
        return x * (x > 0)

    def d_relu(self, x):
        return 1 * (x > 0)
```

```
    def tanh(self, x):
        t=(np.exp(x)-np.exp(-x))/(np.exp(x)+np.exp(-x))
        return t

    def d_tanh(self, x):
        t=(np.exp(x)-np.exp(-x))/(np.exp(x)+np.exp(-x))
        dt=1-t**2
        return t
    def softmax(self, x):
        values = np.exp(x - np.amax(x))
        values = values / np.sum(values)
        values_clipped = np.clip(values, 1e-7, 1 - 1e-7)
        return values_clipped

    def categorical_cross_entropy_loss(self, targets, outputs):
        return -np.sum(targets * np.log(outputs))

    def predict(self, features):
        probabilities = self.forward(features)
        label = np.argmax(probabilities)
#         confidence = probabilities[0][label]
        confidence = probabilities[label]

        return label, confidence

    def evaluate(self, inputs, outputs):
        correct = 0
        classes = np.unique(outputs)
        conf_matrix = np.zeros((len(classes), len(classes)))

        for i, (data,label) in enumerate(zip(inputs,outputs)):
            prediction, _ = self.predict(data)
            if label == prediction:
                correct += 1

            conf_matrix[label][prediction] += 1

        acc = correct / len(inputs)
        return acc


# In[4]:


# create a Multilayer Perceptron with one hidden layer
# model = MLP(4096, [64, 32], 10, "random")
model = MLP(4096, [128 ,64, 32], 10, "random")

# train network
before = time()
history_train, history_test = model.train(trainX, trainY, 0.001, 20, testX, testY)
print("-----------------")
print(f"time elapsed: {time()-before:.2f} seconds")


# In[5]:


plt.plot(history_train, "-b", label="train loss")
plt.plot(history_test, "-r", label="test loss")
plt.title("Epoch - Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend(loc="upper right")


# In[6]:


correct = 0
classes = np.unique(trainY)
conf_matrix = np.zeros((len(classes), len(classes)))
```

```
for i, (data,label) in enumerate(zip(testX,testY)):
    predict = np.argmax(model.forward(data))
    if label == predict:
        correct += 1

    conf_matrix[label][predict] += 1

acc = correct / len(testX)
recall = np.mean(np.diag(conf_matrix) / np.sum(conf_matrix, axis = 1))
precision = np.mean(np.diag(conf_matrix) / np.sum(conf_matrix, axis = 0))

print(f"overall accuracy is {acc:.2f}")
print(f"overall precision is {precision:.2f}")
print(f"overall recall is {recall:.2f}")


# In[7]:


# create a Multilayer Perceptron with one hidden layer
# model = MLP(4096, [64, 32], 10, "he")
model = MLP(4096, [128 ,64, 32], 10, "he")

# train network
before = time()
history_train, history_test = model.train(trainX, trainY, 0.001, 20, testX, testY)
print("-----------------")
print(f"time elapsed: {time()-before:.2f} seconds")


# In[22]:


ticks = np.arange(20) + 1
plt.plot(ticks, history_train, "-b", label="train loss")
plt.plot(ticks, history_test, "-r", label="test loss")
plt.title("Epoch - Loss")
plt.xticks([1,5,10,15,20])
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend(loc="upper right")


# In[9]:


correct = 0
classes = np.unique(trainY)
conf_matrix = np.zeros((len(classes), len(classes)))
for i, (data,label) in enumerate(zip(testX,testY)):
    predict = np.argmax(model.forward(data))
    if label == predict:
        correct += 1

    conf_matrix[label][predict] += 1


# In[10]:


acc = correct / len(testX)
recall = np.mean(np.diag(conf_matrix) / np.sum(conf_matrix, axis = 1))
precision = np.mean(np.diag(conf_matrix) / np.sum(conf_matrix, axis = 0))

print(f"overall accuracy is {acc:.2f}")
print(f"overall precision is {precision:.2f}")
print(f"overall recall is {recall:.2f}")


# In[11]:
```

```
# from scikitlearn 0.18 - old version
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')


    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, "{:0.2f}".format(cm[i, j]),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')


# In[12]:


class_names = ["dog", "horse", "elephant", "butterfly", "chicken", "cat", "cow", "sheep", "spider","squirrel"]

plt.figure()
plt.figure(figsize=(20,20))

plot_confusion_matrix(conf_matrix, classes=class_names, normalize=True,
                      title='Normalized confusion matrix')

plt.show()


# In[13]:


# to free up memory
del trainX
del trainY


# In[14]:


vggModel = models.vgg16(pretrained=True)
vggModel.classifier = nn.Sequential(*list(vggModel.classifier.children())[:-3])
input_size = 224
mean = [0.5177, 0.5003, 0.4126]
std = [0.2135, 0.2130, 0.2151]
tensor_transform = transforms.Compose([
            transforms.Resize((input_size,input_size)),
            transforms.ToTensor()])


# In[15]:
```

```
numberToClass = {
                 0: "dog",
                 1: "horse",
                 2: "elephant",
                 3: "butterfly",
                 4: "chicken",
                 5: "cat",
                 6: "cow",
                 7: "sheep",
                 8: "spider",
                 9: "squirrel"
}


# In[16]:


testNumber = 20
for i in range(int(testNumber/2)):
    randIndex1 = random.randint(0, len(testX) -1)
    randIndex2 = random.randint(0, len(testX) -1)


    im1 = Image.open(testPaths[randIndex1]).convert("RGB")
    im2 = Image.open(testPaths[randIndex2]).convert("RGB")


    features1 = testX[randIndex1]
    features2 = testX[randIndex2]

    label1 = testY[randIndex1]
    label2 = testY[randIndex2]



    prediction1, confidence1 = model.predict(features1)
    prediction2, confidence2 = model.predict(features2)


    f, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
    plt.subplots_adjust(left=0.0, bottom=0, right=1.8, top=0.5,wspace=0.3, hspace=0.3)

    ax1.imshow(im1)
    ax2.imshow(im2)
    ax1.set_title(f"Real-{numberToClass[label1]}, Prediction-{numberToClass[prediction1]}, confidence:
{confidence1:.2f}")
    ax2.set_title(f"Real-{numberToClass[label2]}, Prediction-{numberToClass[prediction2]} , confidence:
{confidence2:.2f}")

    plt.show()


# In[17]:


os.chdir("D:\\python\\animal classifier")


root_dir = os.getcwd()
testDir = os.path.join(root_dir, "plotImgs")
os.chdir(testDir)

file = open("labels.txt", "r")
labels = file.read().strip().split("-")

with torch.no_grad():
    for i in range(len(os.listdir())): # 1 file is txt file
        if f"{i+1}.jpg" in os.listdir():
            path = f"{i+1}.jpg"

        elif f"{i+1}.jpeg" in os.listdir():
            path = f"{i+1}.jpeg"
```

```
else:
    continue

imgPIL = Image.open(path) #PIL image
imgTensor = tensor_transform(imgPIL) #torch.FloatTensor 3,224,224
features = vggModel(imgTensor[None, ...])
features = features.detach().numpy()

prediction, confidence = model.predict(features[0])

f, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
plt.subplots_adjust(left=0.0, bottom=0, right=1.3, top=0.5,wspace=0.3, hspace=0.3)
ax1.imshow(imgPIL)
ax2.imshow( imgTensor.permute(1, 2, 0) )
ax1.set_title(f"Real : {numberToClass[prediction]}")
ax2.set_title(f"{numberToClass[prediction]} , confidence: {confidence:.2f}")

plt.show()
```