Algorithms and Applications in Computer Vision – 046746

Homework 1 Report

Bar Goldner [redacted]

Daria Hasin [redacted]

23/11/2022

Part 1 – Classic Classifier:

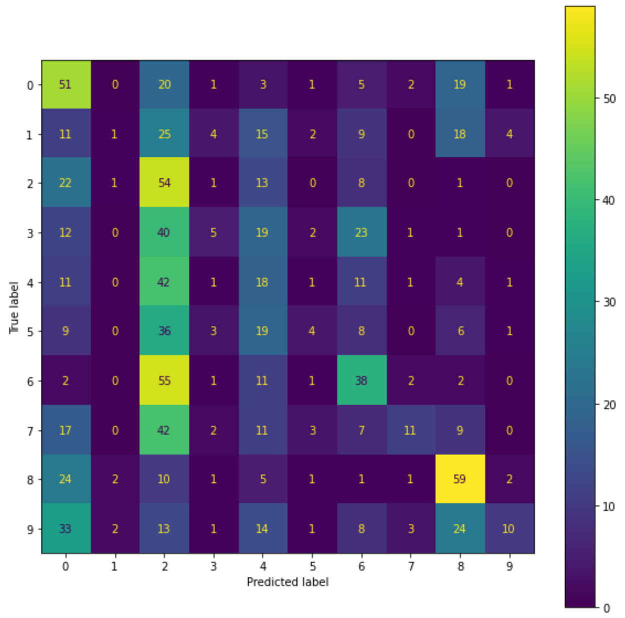
1. 5 images from the CIFAR-10 training data set with their labels:

   We made some pre-processing steps on the images - we jittered the brightness, the contrast and the saturation by 0.2. We also converted the image to pytorch tensor.

2. We built a K-Nearest Neighbors classifier for k=10 using sklearn library and flattened the image so it will fit the KNN input requests.

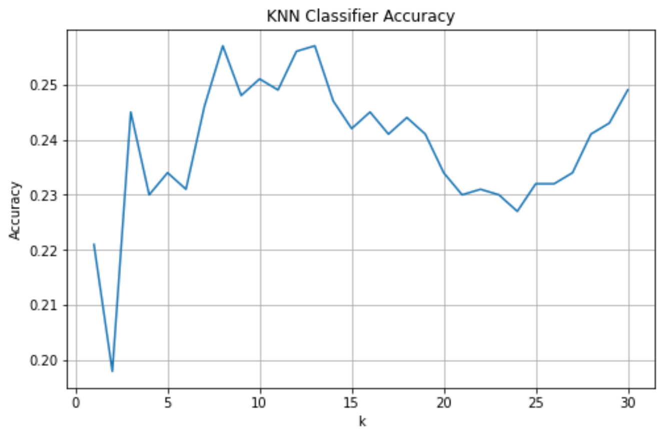
3. The accuracy we obtained for loading 1,000 samples from CIFAR-10 test set of the K-NN classifier for k=10 was **25.1%**.

   The model result via confusion matrix:



   We can see in the results that the model has the best results for labels 0, 2, 8. Yet, we also see that label 2 got a lot of false positives so we can’t assume that the true positive results are not as a result of biased model to this label.

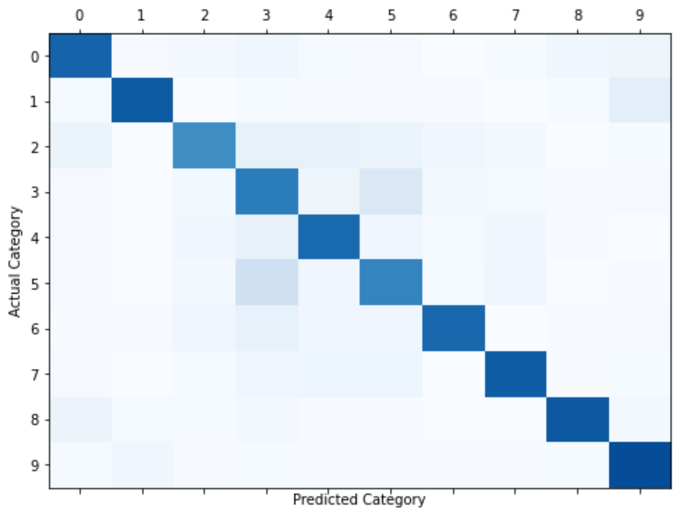
4. Comparison graph of the model's accuracy for a different number of neighbors (1≤K≤30):



We can see in the graph that we got low accuracy for all k's between 1-30 while k=1 is the lowest (the model is not overfitted) and the accuracy gets a little bit better for the middle k values because it gets more labels to compare the distance with, but the accuracy is still poor. We assume that we got these results because KNN is typically not a good classifier, like we learned in class.

Part 2 - Design and Build a CNN Classifier:

1. After using the CNN from tutorials 3-4 and training it on the CIFAR-10 dataset, the accuracy that was obtained is: **77.8%**.
   The model result via confusion matrix:



   From the confusion matrix, the most confusing classes for this model are 2, 3 and 5 because we get the least true positives within these labels.

2. We designed our CNN called Baria_CNN.
   Our chosen architecture was built from 17 layers: 5 Conv2d layers, 4 MaxPooling layers which is similar to convolution layer, but instead of doing convolution operation, the max values are selected in the receptive fields of the input, saving the indices and then producing a summarized output volume. 3 BatchNorm2d layers which applies Batch Normalization over a 4D input, and 1 Dropout2d and 1 Dropout layer which zero out random channels sample in the batched input.
   We also used 3 Fully Connected layers towards the end of the layers with the sizes:
   1x1080 which is the image dimensions multiply the output channels of the layer before (2*2*270).
   1x40 which was chosen in order to add a layer that gives a division of an integer to the number 1080.
   1x20 which is the number of the layer that its result is the dimension of the model classes, 10 in this model and this is the prediction layer.
   In a Fully Connected layer, all the neurons of the input are connected to every neuron of the output. The Fully Connected receives 1D array.
   The activation functions we chose are ReLU and Leaky ReLU.
   Leaky ReLU is an activation function that based on ReLU (Rectified Linear Unit) but has a slope for negative values that can be determined per Leaky ReLU function, in opposite to ReLU that this slope is 0. For positive values the slope is 1 so the output is equal to the input.
   The input dimension is 3x32x32 which is the image width and height with 3 channels of colors. The output dimension is 1x10 and it represents the prediction layer.

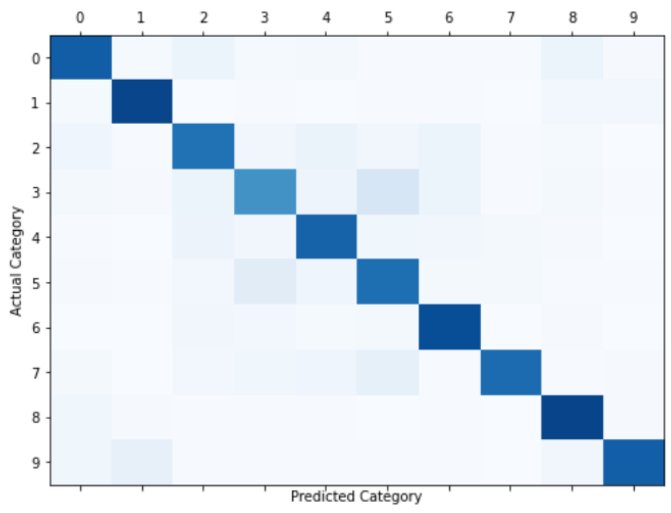
Calculations of the number of parameters (weights) in the network (without bias):

		Calculations	**Parameters**
CNN	Conv2d	3*60*3*3	1620
	BatchNorm2d	2*60	120
	ReLU		0
	MaxPool2d		0
	Conv2d	60*60*3*3	32400
	Leaky ReLU		0
	Conv2d	60*90*3*3	48600
	BatchNorm2d	2*90	180
	ReLU		0
	Dropout2d		0
	Conv2d	90*180*3*3	145800
	ReLU		0
	MaxPool2d		0
	Conv2d	180*270*3*3	437400
	BatchNorm2d	2*270	540
	ReLU		0
	MaxPool2d		0
	MaxPool2d		0
FC	Dropout		0
	Linear	1080*40	43200
	ReLU		0
	Linear	40*20	800
	ReLU		0
	Linear	20*10	200
Sum of parameters			**710860**

3. The hyper-parameters in this model are:
   Model architecture – we changed the architecture of the model in order to get better accuracy.
   Learning rate - controls how much to change the model in response to the estimated error when the model weights are updated.
   Batch size - defines the number of samples to work through before updating the internal model parameters.
   Epochs - defines the number of times that the learning algorithm will work through the entire training dataset.
   Optimizer - iterates at each model and the search space to optimize and then minimizes the objective function.

   We tuned the model by changing the architecture and by using cross-validation.
   We divided the train data randomly into two sets, train and validation, for k=3 iterations so we can evaluate the results as we go. We used cross-validation to protect the model from overfitting to the training data.

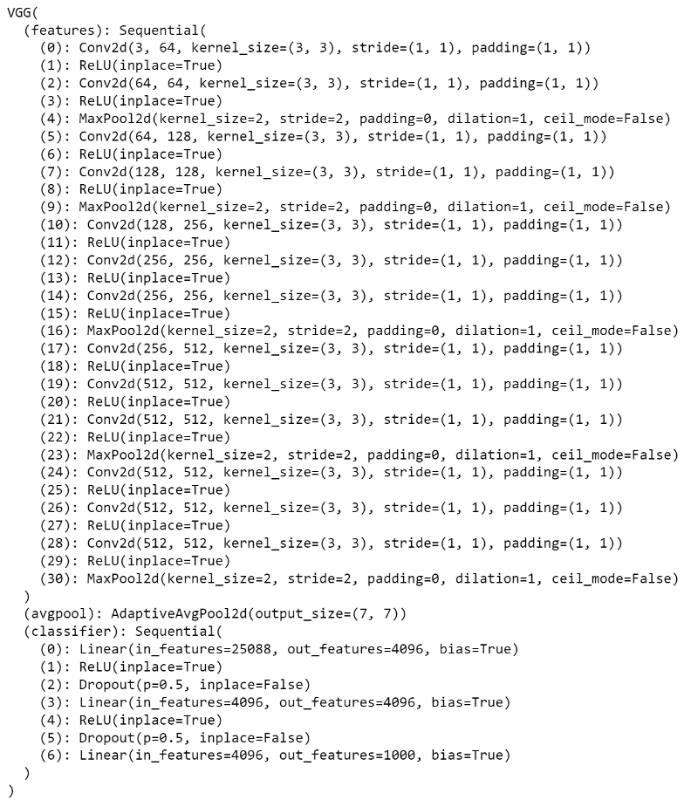
The final accuracy that was obtained is: **80.68%**.
The model result via confusion matrix:



From the confusion matrix, the most confusing classes for this model are also 2, 3 and 5 but we can see that labels 0, 1, 6 and 8 have better results because of their darker color compared to the confusion matrix of the tutorial model.

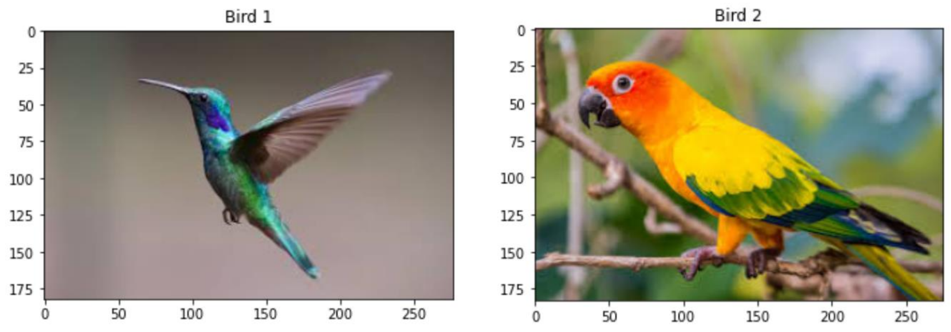
Part 3 - Analyzing a Pre-trained CNN:

1. We Loaded a pre-trained VGG16 with PyTorch and used the model in evaluation mode.
   Pre-trained 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)
  )
)
```

2. We loaded the two birds' images, and to display them, we converted them to RGB color format:



3. To get better model learning, we made pre-process steps to the images, these steps are required tasks for cleaning the data and making it suitable for the model, which increases the accuracy and efficiency.
   The Pre-process steps that we take over the images to fit VGG16's architecture:
   - ToTensor() – it converts the images to torch.FloatTensor type of shape (CxHxW) in range of [0.0, 1.0] instead of [0, 256].
   - Resize((224, 224)) – it resizes all the images to 224x224 pixels, we did that because the requirement that where H and W are expected to be at least 224 pixels.

- Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]) – it normalizes a tensor image with mean and standard deviation by this formula: output[channel] = (input[channel] - mean[channel]) / std[channel].

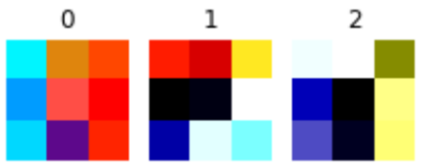
The birds' images after pre-processing:

4. We read the label data from the .txt file and converted it to labels by reading the file and evaluate it into a dictionary. We fed the two birds' images into the model and checked which label each image fitted.
   The outputs of the images are:
   Bird 1 – hummingbird.
   Bird 2 – lorikeet.

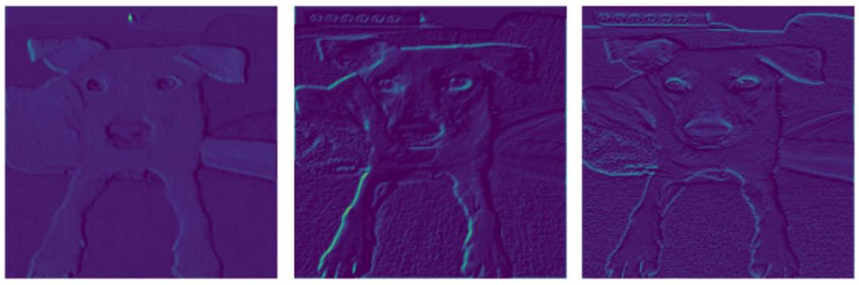
5. We took an image of Miley – Daria's dog, to display it, we converted them to RGB color format.

   We made the same pre-process steps from the section before and then fed it to the model.
   The output is: **Whippet**.
   (P.S. We tried to run an image of a Haifa wild boar that we took on the street and the model did recognize it as a wild boar!)

6. The first 3 filters of the first layer of VGG16 plots are:



Their responses for the image of Miley are:

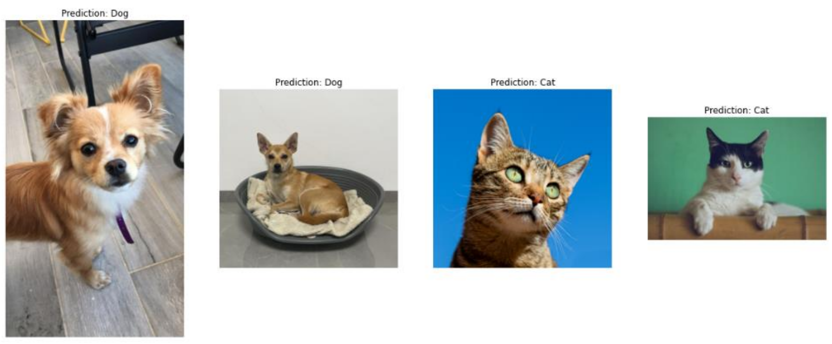


We can see that there are three different filters with the dimensions of 3x3 and when we applied each filter, we got three different image responses.
From the first image response, we got a general shape of the dog's structure, without any interference from the background.
From the second image response, we got the dog fur's texture, and the shape and number of the fingers can be clearly seen.
From the third image response, we got the nose, eyes and ears shape and size and their position in the image space.
Each filter is able to indicate to us a different part of the image that has repetition that can be seen in each of the images of the same class and that is a part of what helps the model to become good model.

7. For each image in the ‘dogs’ folder and ‘cats’ folder, we extracted and saved their feature vectors from a fully connected layer of the VGG16 model, we again, made the same pre-process steps from the section before.
The layer we picked was layer 36 which is the fully connected layer before the last one.
The size of the feature space is 1x4096
We chose this layer and not the last fully connected layer because the last one represents the decision layer and we wanted to represent the image date and not the prediction.

8. We trained an SVM model with the 20 images of dogs and cats and created a label vector in which 0 represents dog and 1 represents cat.
We took two images of Bar's dogs – Eevee and Bony, and two cats' images from the internet, made pre-process steps and made a list of the results that used as the testing set.

9. We fed the testing set to the model and displayed the results.
   The results were very good and all the 4 labels were correct! (Even though Bony kind of look like a cat)



<u>Part 4 - Dry Questions:</u>

1. The bag of words algorithm is a model that represents an image as a vector of occurrence counts of local image patches that occur in it.
   The use of this model for flags detection has some pros and cons:

   Pros:

   - <u>Fast training</u>: A lot of flags have similar parts in their design, like only 2 or 3 different colors in lines and it will make a small number of different patches of the image. The training time shouldn't take long because there is a small number of patches to go through.

   - <u>Fast Prediction</u>: Because of the simplicity of the colors and shapes in flags, the computations that will be needed to compare and eliminate possibilities of the vector of occurrence counts of local image patches, will be short and quick. For example, vector that has patches of green color, quickly will be eliminated when a flag without green color is tested.

   Cons:

   - <u>Low accuracy</u>: Due to colors and shape similarity between different flags, the accuracy will be low, that is because this model takes similar patches vector for different flags. For example, the flag of Monaco that look like the flag of Indonesia.

   - <u>Prediction consistency</u>: Due to the similarity between flags with the same shapes and colors, they will have very similar vectors of occurrence counts which will affect the consistency of the prediction. For example, the flags of Holland and France have similar colors, shapes, and colors order in their flags so if the model will test the flag of France twice, it can decide for the first time that this is the flag of France, and for the second time that this is the flag of Holland.
     Visual example:
     Patches from Syria's flag:
     Patches from Sudan's flag:
     Are practically the same.

2.

Step	Layer	Output dimensions	Number of parameters
1	INPUT	64x64x3	0
2	CONV7-16	62x62x16	3*16*7*7 = 2353
3	POOL2	31x31x16	0
4	CONV7-32	29x29x32	16*32*7*7 = 25088
5	POOL2	14x14x32	0
6	FC-3	1x3	32*14*14*3 = 18816

General convolutional layer output size formula:

$$W_{out} = \left\lfloor \frac{W_{in} - F + 2P}{S} + 1 \right\rfloor$$

General Max Pooling (2x2) layer output size formula:

$$W_{out} = \left\lfloor \frac{W_{in}}{2} \right\rfloor$$

Step 2: Conv2d(input: 3, output: 16, kernel_size=(7, 7), stride=(1), padding=(2))

⇨ $W_{out} = \frac{64-7+2\cdot2}{1} + 1 = 62$

Number of parametars: 3 input channels, 26 output channels with 7x7 kernel size.

Step 3: MaxPooling(2x2)

⇨ $W_{out} = \frac{62}{2}$ = 31

Number of parametars: 0 (no new data use)

Step 4: Conv2d(input: 16, output: 32, kernel_size=(7, 7), stride=(1), padding=(2))

⇨ $W_{out} = \frac{31-7+2\cdot2}{1} + 1 = 29$

Number of parametars: 16 input channels, 32 output channels with 7x7 kernel size.

Step 5: MaxPooling(2x2)

⇨ $W_{out} = \frac{29}{2} = 14.5 => 14\ (flooring)$

⇨ Number of parametars: 0 (no new data use)

Step 6: Fully Connected layer with 3 neurons will give 1 dimension tensor with the size of 3.

Number of parametars: 32 input channels, image dimensions 14x14, 3 classes.