CSE480

Machine Vision

Major Task (2)

Group Members:

Name	*ID*
Mostafa Abdelnasser Hassan	1801145
Mahmoud Gamal Mohamed	1802166
Ahmed Hossam El Din Ramadan	1807260

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Multilayer Neural Networks trained with the backpropagation algorithm constitute the best example of a successful Gradient-Based Learning technique. Given an appropriate network architecture, Gradient-Based Learning algorithms can be used to synthesize a complex decision surface that can classify high-dimensional patterns such as handwritten characters, with minimal pre-processing. Convolutional Neural Networks, that are specifically designed to deal with the variability of 2D shapes, are shown to outperform all other techniques.

INTRODUCTION

Over the last several years, machine learning techniques, particularly when applied to neural networks, have played an increasingly important role in the design of pattern recognition systems. In fact, it could be argued that the availability of learning techniques has been a crucial factor in the recent success of pattern recognition applications such as continuous speech recognition and handwriting recognition.

Convolutional neural network (CNN), a type of artificial neural network that has been dominant in several computer vision tasks, is gaining popularity in a range of fields, including radiology. CNN is meant to learn spatial hierarchies of data automatically and adaptively by backpropagation utilizing several building blocks such as convolution layers, pooling layers, and fully connected layers. This review article provides an overview of the fundamental ideas of CNN and its application to various radiological tasks, as well as a discussion of its problems and future directions in the area of radiology. We will also discuss two obstacles in applying CNN to radiological tasks: short datasets and overfitting, as well as approaches for overcoming them. Understanding the ideas, benefits, and limits of CNN is critical for maximizing its potential in diagnostic radiology, with the objective of increasing radiologists' performance and patient care.

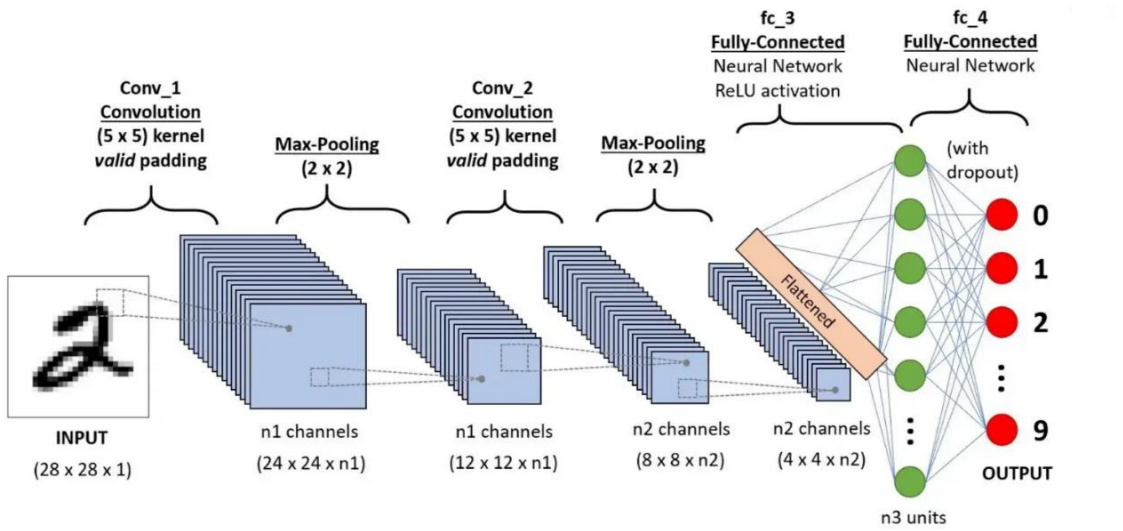


Figure 1: Convolution Neural Network Architecture

CONCEPTS OF CONVOLUTIONAL NEURAL NETWORK

Convolutional Neural Network (CNN), also known as ConvNet, is a type of Artificial Neural Network (ANN) with deep feed-forward architecture and incredible generalising ability when compared to other networks with FC layers. It can learn highly abstracted features of objects, particularly spatial data, and identify them more efficiently. A deep CNN model is made up of a finite number of processing layers that may learn various aspects of input data (such as images) at various levels of abstraction. The deeper layers learn and extract low level features whereas the initiatory layers learn and extract high level features (with lesser abstraction) (with higher abstraction). Figure 2 depicts the fundamental conceptual model of CNN, with several types of layers explained in later sections.

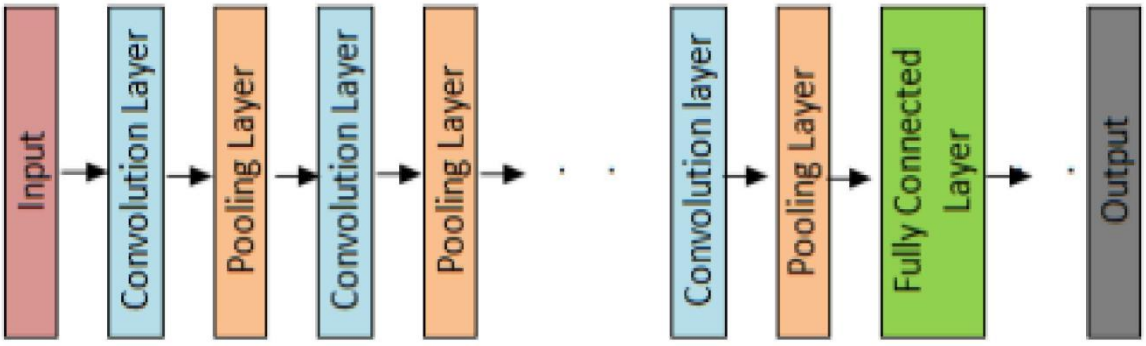


Figure 2: Conceptual Model of CNN

Why Convolutional Neural Networks is more considerable over other classical neural networks in the context of computer vision?

- One of the main reasons for considering CNN in such case is the weight sharing feature of CNN, that reduce the number of trainable parameters in the network, which helped the model to avoid overfitting and as well as to improved generalization.
- In CNN, the classification layer and the feature extraction layers learn together, that makes the output of the model more organized and makes the output more dependent to the extracted features.

- The implementation of a large network is more difficult by using other types of neural networks rather than using Convolutional Neural Networks.

Now description of different components or basic building blocks of CNN briefly as follows.

2.1. Network Layers

As we mentioned earlier, that a CNN is composed of multiple building blocks (known as layers of the architecture), in this subsection, we described some of these building blocks in detail with their role in the CNN architecture.

2.1.1. Convolutional Layer

Convolutional layer is the most important component of any CNN architecture. It contains a set of convolutional kernels (also called filters), which gets convolved with the input image (N-dimensional metrics) to generate an output feature map.

2.1.2. Pooling Layer

The pooling layers are used to sub-sample the feature maps (created after convolution operations), i.e., it takes bigger feature maps and compresses them to smaller feature maps. When decreasing the feature maps, the most prominent features (or information) are always preserved in each pool phase. Similar to the convolution operation, the pooling operation is conducted by defining the pooled region size and the stride of the operation. The biggest disadvantage of the pooling layer is that it occasionally reduces the overall performance of CNN. The reason for this is that the pooling layer assists CNN in determining whether or not a certain feature is present in the given input picture without regard for the correct location of that feature.

METHODS AND ALGORITHMS

In this section we try to discuss the training or learning process of a CNN model with certain guidelines in order to reduce the required training time and to improve model accuracy. The training process mainly includes the following steps:

- Data pre-processing and Data augmentation.
- Parameter initialization.
- Regularization of CNN.
- Optimizer selection.

3.1. Data pre-processing and Data augmentation

Data pre-processing is the process of applying artificial adjustments to the raw dataset (including the training, validation, and testing datasets) to improve its uniformity, cleanliness, and feature content. Before supplying the input to the CNN model, the data is pre-processed. It is a known fact that the performance of a convolutional neural network (CNN) is directly proportional to the volume of data used to train it, i.e., effective pre-processing always improves model accuracy. On the other hand, a poor pre-processing might also make the model perform worse.

3.2. Parameter Initialization

A deep CNN consists of millions or billions number of parameters. So, it must be well initialized at the begin of the training process, because weight initialization directly determines how fast the CNN model would converge and how accurately it might end up. The easiest way to doing it is by initializing all the weights with zero. However, this turns out to be a mistake, because if we initialize weights of all layer to zero, the output as well as the gradients (during backpropagation) calculated by every neuron in the network will be the same. Hence the update to all the weights would also be the same. As a result, there is no disparity between neurons and the network will not learn any useful features. To break this disparity between neurons,

we do not initialize all weights with the same value, rather than, we use different techniques to initialize the weights randomly as follows:

3.2.1. Random Initialization

As the name suggests, here we initialize the weights (belonging from both convolutional and FC layers) randomly using random matrices, where the elements of that matrices are sampled from some distribution with small standard deviation (e.g., 0.1 and 0.01) and with zero mean. But the key problem of random initialization is that it may potentially lead to vanishing gradients or exploding gradients problems.

3.2.2. Xavier Initialization

This technique is proposed by Xavier Glorot and Yoshua Bengio in 2010, it tries to make the variance of output connections and input connections to be equal for each layer in the network. The main idea is to balancing of the variance the activation functions. It does not perform well with ReLU (mostly used in CNN architecture nowadays) activation function. Later on, He initialization technique proposed by Kaiming He at al is used to work with ReLU activation on the same idea.

3.3. Regularization to CNN

The primary difficulty for deep learning algorithms is to generalise effectively to new or previously unobserved input, derived from the same distribution as training data. Over-fitting is the primary obstacle to a CNN model achieving adequate generalisation. A model is said to be over-fitted if it performs remarkably well on training data but poorly on test data (unseen data). The opposite is an under-fitted model, which occurs when the model has not learnt enough from the training data. Just-fitted models are those that perform well on both train and test data. The examples of over-fitted, under-fitted, and just-fitted models are attempted to be displayed in Fig. 3.

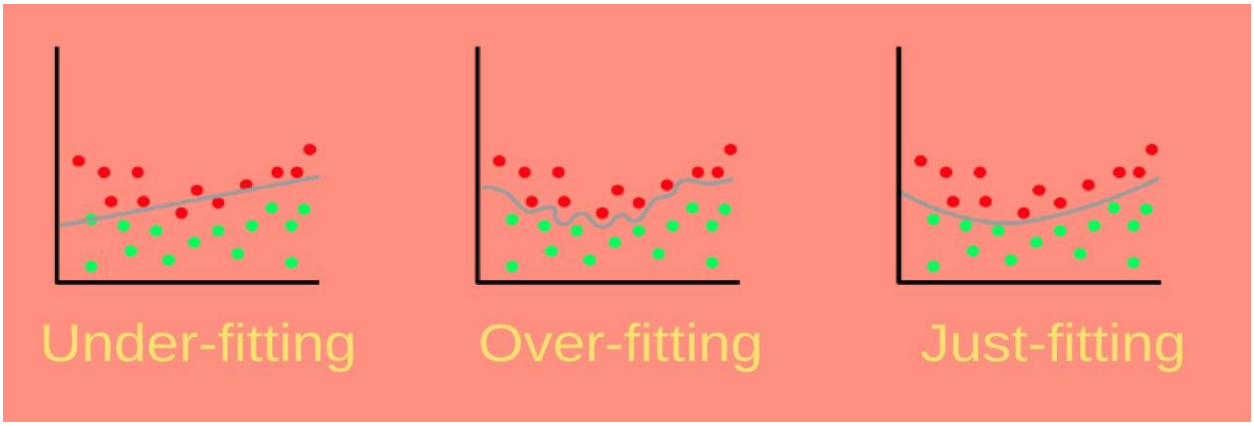


Figure 3: Over-fitted, Under-fitted and Just-fitted Models

Regularization helps to avoid over-fitting by using several intuitive ideas, some of which are:

1. Dropout
2. Drop-Weights
3. The ι^2 Regularization
4. The ι^1 Regularization
5. Data Augmentation
6. Early Stopping
7. Batch Normalization

3.4. Optimizer Selection

The learning process includes two major things, first one is the selection of the learning algorithm (Optimizer) and the next one is to use several improvements to that learning algorithm in order to improve the result. The main objective of any supervised learning algorithm is to minimize the Error or we can say the loss functions, based on several learnable parameters like weights, biases, etc. In case of learning to a CNN model the gradient-based learning methods come as a natural choice. To reduces the error the model parameters are being continuously updated during each training epoch and the model iteratively search for the locally optimal solution in each training epoch.

3.5. Pseudocode

#import important libraries

- cnn related
- data sets
- load data sets in train & test variables

Normalize the variables

#Make our CNN model

- Conv2d
- Activation”relu”
- Batch normalize
- Activation”relu”
- Maxpool2d
- Dropout
- Next phase
- Until end then
- Flatten
- SoftMax layer
- Get parameters

#train our model

- Using RMSprop optimizer with predetermined learning rate
- Compute loss
- Update parameter

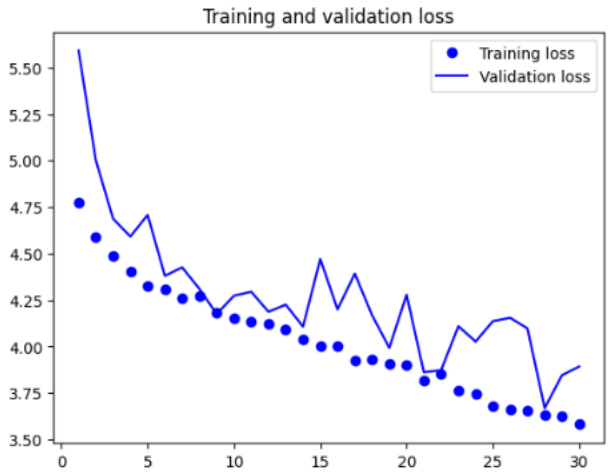
RESULTS AND OUTPUTS

4.1. 1st Trial: Model Training over 30 epochs

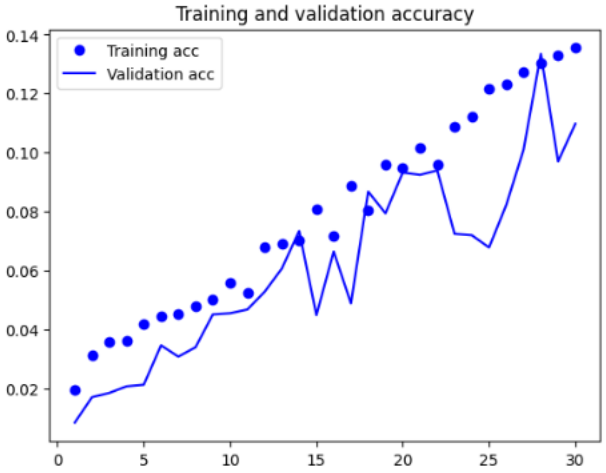
```
# Training model
import time
training_start = time.time()
history = model.fit(train_datagen.flow(X_train, Y_train, batch_size=70), steps_per_epoch=50,epochs=30,
          validation_data=(X_validation, Y_validation),
          verbose=1)
training_stop = time.time()
training_time = training_stop - training_start
print( f"Training time: {training_time}")
```

```
Epoch 1/30
50/50 [==============================] - 380s 8s/step - loss: 4.7768 - acc: 0.0194 - val_loss: 5.5914 - val_acc: 0.0083
Epoch 2/30
50/50 [==============================] - 378s 8s/step - loss: 4.5860 - acc: 0.0311 - val_loss: 5.0011 - val_acc: 0.0170
Epoch 3/30
50/50 [==============================] - 375s 8s/step - loss: 4.4877 - acc: 0.0357 - val_loss: 4.6873 - val_acc: 0.0184
Epoch 4/30
50/50 [==============================] - 378s 8s/step - loss: 4.4050 - acc: 0.0363 - val_loss: 4.5915 - val_acc: 0.0206
Epoch 5/30
50/50 [==============================] - 376s 8s/step - loss: 4.3228 - acc: 0.0420 - val_loss: 4.7077 - val_acc: 0.0212
Epoch 6/30
50/50 [==============================] - 378s 8s/step - loss: 4.3088 - acc: 0.0443 - val_loss: 4.3802 - val_acc: 0.0346
Epoch 7/30
50/50 [==============================] - 377s 8s/step - loss: 4.2585 - acc: 0.0451 - val_loss: 4.4256 - val_acc: 0.0307
50/50 [==============================] - 377s 8s/step - loss: 4.2585 - acc: 0.0451 - val_loss: 4.4256 - val_acc: 0.0307
Epoch 8/30
50/50 [==============================] - 375s 8s/step - loss: 4.2745 - acc: 0.0480 - val_loss: 4.3097 - val_acc: 0.0339
Epoch 9/30
50/50 [==============================] - 376s 8s/step - loss: 4.1808 - acc: 0.0500 - val_loss: 4.1746 - val_acc: 0.0450
Epoch 10/30
50/50 [==============================] - 375s 8s/step - loss: 4.1547 - acc: 0.0557 - val_loss: 4.2732 - val_acc: 0.0454
Epoch 11/30
50/50 [==============================] - 378s 8s/step - loss: 4.1329 - acc: 0.0526 - val_loss: 4.2945 - val_acc: 0.0467
Epoch 12/30
50/50 [==============================] - 380s 8s/step - loss: 4.1233 - acc: 0.0680 - val_loss: 4.1869 - val_acc: 0.0527
Epoch 13/30
50/50 [==============================] - 376s 8s/step - loss: 4.0933 - acc: 0.0689 - val_loss: 4.2251 - val_acc: 0.0606
Epoch 14/30
50/50 [==============================] - 376s 8s/step - loss: 4.0415 - acc: 0.0700 - val_loss: 4.1072 - val_acc: 0.0734
Epoch 15/30
50/50 [==============================] - 376s 8s/step - loss: 4.0043 - acc: 0.0809 - val_loss: 4.4706 - val_acc: 0.0448
Epoch 16/30
50/50 [==============================] - 377s 8s/step - loss: 4.0029 - acc: 0.0717 - val_loss: 4.1999 - val_acc: 0.0664
Epoch 17/30
50/50 [==============================] - 377s 8s/step - loss: 3.9246 - acc: 0.0889 - val_loss: 4.3918 - val_acc: 0.0488
Epoch 18/30
50/50 [==============================] - 379s 8s/step - loss: 3.9279 - acc: 0.0803 - val_loss: 4.1684 - val_acc: 0.0867
Epoch 26/30
50/50 [==============================] - 378s 8s/step - loss: 3.6642 - acc: 0.1231 - val_loss: 4.1542 - val_acc: 0.0822
Epoch 27/30
50/50 [==============================] - 377s 8s/step - loss: 3.6564 - acc: 0.1274 - val_loss: 4.0975 - val_acc: 0.1010
Epoch 28/30
50/50 [==============================] - 377s 8s/step - loss: 3.6306 - acc: 0.1303 - val_loss: 3.6686 - val_acc: 0.1334
Epoch 29/30
50/50 [==============================] - 377s 8s/step - loss: 3.6262 - acc: 0.1331 - val_loss: 3.8456 - val_acc: 0.0970
Epoch 30/30
50/50 [==============================] - 381s 8s/step - loss: 3.5809 - acc: 0.1354 - val_loss: 3.8926 - val_acc: 0.1098
Training time: 11336.094021320343
```

4.2. Training and Validation Accuracies



4.3. Training and Validation Loss



4.4. Model Performance and Accuracy

In [123]:
```
#test models on testing data
scores = model.evaluate (test_images, test_labels)
print (f'accuracy on test set: {model.metrics_names [1]} of {scores [1]*100}')
```

```
313/313 [==============================] - 139s 445ms/step - loss: 3.9126 - acc: 0.1043
accuracy on test set: acc of 10.429999977350235
```

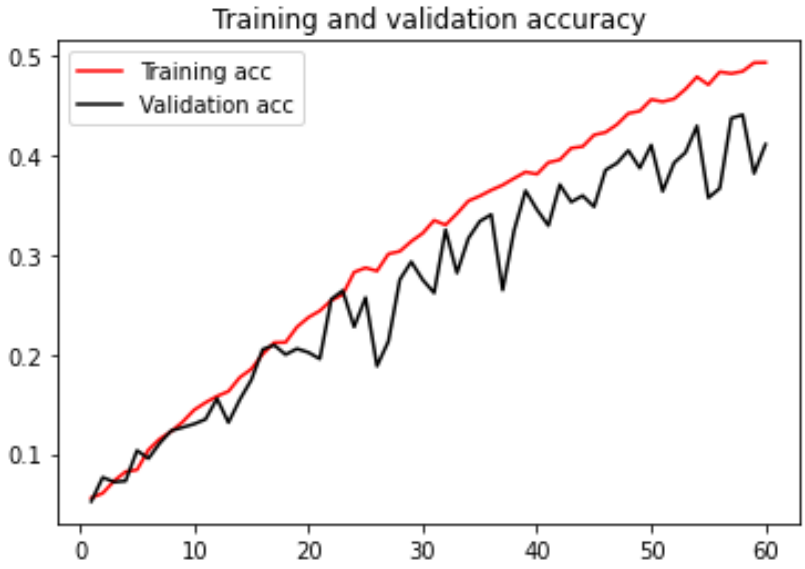
4.5. 2nd Trial: Model Training over 60 epochs

```
[ ]  Epoch 1/60
     100/100 [==============================] - 29s 247ms/step - loss: 4.1240 - acc: 0.0562 - val_loss: 4.1999 - val_acc: 0.0527
     Epoch 2/60
     100/100 [==============================] - 24s 240ms/step - loss: 4.0630 - acc: 0.0609 - val_loss: 3.9263 - val_acc: 0.0767
     Epoch 3/60
     100/100 [==============================] - 24s 242ms/step - loss: 3.9874 - acc: 0.0733 - val_loss: 4.1014 - val_acc: 0.0721
     Epoch 4/60
     100/100 [==============================] - 24s 240ms/step - loss: 3.9304 - acc: 0.0824 - val_loss: 3.9277 - val_acc: 0.0731
     Epoch 5/60
     100/100 [==============================] - 24s 240ms/step - loss: 3.9052 - acc: 0.0843 - val_loss: 3.7897 - val_acc: 0.1037
     Epoch 6/60
     100/100 [==============================] - 24s 244ms/step - loss: 3.7877 - acc: 0.1043 - val_loss: 3.9402 - val_acc: 0.0956
     Epoch 7/60
     100/100 [==============================] - 24s 245ms/step - loss: 3.7032 - acc: 0.1153 - val_loss: 3.7253 - val_acc: 0.1112
     Epoch 8/60
     100/100 [==============================] - 24s 240ms/step - loss: 3.6481 - acc: 0.1229 - val_loss: 3.6063 - val_acc: 0.1235
     Epoch 9/60
     100/100 [==============================] - 25s 247ms/step - loss: 3.5769 - acc: 0.1324 - val_loss: 3.7616 - val_acc: 0.1268
     Epoch 10/60
     100/100 [==============================] - 24s 241ms/step - loss: 3.5294 - acc: 0.1443 - val_loss: 3.6177 - val_acc: 0.1302
     Epoch 11/60
     100/100 [==============================] - 25s 247ms/step - loss: 3.4901 - acc: 0.1517 - val_loss: 3.6979 - val_acc: 0.1350
     Epoch 12/60
     100/100 [==============================] - 24s 242ms/step - loss: 3.4283 - acc: 0.1580 - val_loss: 3.4049 - val_acc: 0.1557
     Epoch 13/60
     100/100 [==============================] - 24s 240ms/step - loss: 3.4050 - acc: 0.1629 - val_loss: 3.6739 - val_acc: 0.1317
     Epoch 14/60
     100/100 [==============================] - 24s 242ms/step - loss: 3.3280 - acc: 0.1769 - val_loss: 3.4349 - val_acc: 0.1548
     Epoch 15/60
     100/100 [==============================] - 24s 244ms/step - loss: 3.2583 - acc: 0.1852 - val_loss: 3.3579 - val_acc: 0.1743
     Epoch 16/60
     100/100 [==============================] - 25s 246ms/step - loss: 3.2052 - acc: 0.2002 - val_loss: 3.1613 - val_acc: 0.2050
     Epoch 17/60
     100/100 [==============================] - 24s 240ms/step - loss: 3.1658 - acc: 0.2117 - val_loss: 3.1587 - val_acc: 0.2093
     Epoch 18/60
     100/100 [==============================] - 25s 247ms/step - loss: 3.1155 - acc: 0.2120 - val_loss: 3.1981 - val_acc: 0.1998
     Epoch 19/60
     100/100 [==============================] - 24s 241ms/step - loss: 3.0492 - acc: 0.2274 - val_loss: 3.1934 - val_acc: 0.2056
```

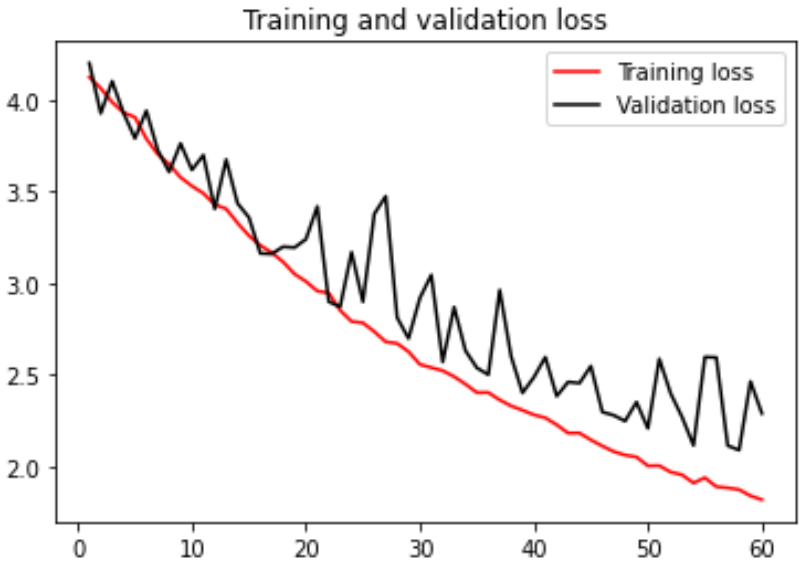
```
[ ]  Epoch 20/60
     100/100 [==============================] - 25s 245ms/step - loss: 3.0086 - acc: 0.2370 - val_loss: 3.2390 - val_acc: 0.2019
     Epoch 21/60
     100/100 [==============================] - 27s 270ms/step - loss: 2.9562 - acc: 0.2436 - val_loss: 3.4176 - val_acc: 0.1954
     Epoch 22/60
     100/100 [==============================] - 24s 240ms/step - loss: 2.9456 - acc: 0.2540 - val_loss: 2.8981 - val_acc: 0.2552
     Epoch 23/60
     100/100 [==============================] - 25s 245ms/step - loss: 2.8512 - acc: 0.2598 - val_loss: 2.8700 - val_acc: 0.2638
     Epoch 24/60
     100/100 [==============================] - 25s 246ms/step - loss: 2.7904 - acc: 0.2823 - val_loss: 3.1677 - val_acc: 0.2275
     Epoch 25/60
     100/100 [==============================] - 24s 240ms/step - loss: 2.7825 - acc: 0.2867 - val_loss: 2.8977 - val_acc: 0.2568
     Epoch 26/60
     100/100 [==============================] - 24s 244ms/step - loss: 2.7346 - acc: 0.2835 - val_loss: 3.3776 - val_acc: 0.1882
     Epoch 27/60
     100/100 [==============================] - 25s 247ms/step - loss: 2.6790 - acc: 0.3006 - val_loss: 3.4734 - val_acc: 0.2130
     Epoch 28/60
     100/100 [==============================] - 27s 267ms/step - loss: 2.6702 - acc: 0.3032 - val_loss: 2.8115 - val_acc: 0.2749
     Epoch 29/60
     100/100 [==============================] - 25s 247ms/step - loss: 2.6257 - acc: 0.3132 - val_loss: 2.6972 - val_acc: 0.2927
     Epoch 30/60
     100/100 [==============================] - 24s 240ms/step - loss: 2.5555 - acc: 0.3213 - val_loss: 2.9184 - val_acc: 0.2747
     Epoch 31/60
     100/100 [==============================] - 24s 243ms/step - loss: 2.5381 - acc: 0.3342 - val_loss: 3.0438 - val_acc: 0.2615
     Epoch 32/60
     100/100 [==============================] - 25s 246ms/step - loss: 2.5214 - acc: 0.3296 - val_loss: 2.5689 - val_acc: 0.3248
     Epoch 33/60
     100/100 [==============================] - 24s 242ms/step - loss: 2.4891 - acc: 0.3410 - val_loss: 2.8684 - val_acc: 0.2815
     Epoch 34/60
     100/100 [==============================] - 24s 240ms/step - loss: 2.4476 - acc: 0.3537 - val_loss: 2.6300 - val_acc: 0.3162
     Epoch 35/60
     100/100 [==============================] - 24s 243ms/step - loss: 2.4020 - acc: 0.3586 - val_loss: 2.5366 - val_acc: 0.3333
     Epoch 36/60
     100/100 [==============================] - 24s 244ms/step - loss: 2.4023 - acc: 0.3645 - val_loss: 2.4995 - val_acc: 0.3401
     Epoch 37/60
     100/100 [==============================] - 25s 246ms/step - loss: 2.3617 - acc: 0.3696 - val_loss: 2.9615 - val_acc: 0.2644
     Epoch 38/60
     100/100 [==============================] - 24s 240ms/step - loss: 2.3291 - acc: 0.3763 - val_loss: 2.5997 - val_acc: 0.3242
     Epoch 39/60
     100/100 [==============================] - 27s 268ms/step - loss: 2.3054 - acc: 0.3827 - val_loss: 2.4001 - val_acc: 0.3639
     Epoch 40/60
     100/100 [==============================] - 24s 245ms/step - loss: 2.2805 - acc: 0.3805 - val_loss: 2.4857 - val_acc: 0.3450
     Epoch 41/60
     100/100 [==============================] - 24s 245ms/step - loss: 2.2642 - acc: 0.3922 - val_loss: 2.5951 - val_acc: 0.3290
     Epoch 42/60
     100/100 [==============================] - 24s 240ms/step - loss: 2.2260 - acc: 0.3949 - val_loss: 2.3824 - val_acc: 0.3700
     Epoch 43/60
     100/100 [==============================] - 24s 241ms/step - loss: 2.1809 - acc: 0.4066 - val_loss: 2.4597 - val_acc: 0.3526
     Epoch 44/60
     100/100 [==============================] - 24s 245ms/step - loss: 2.1809 - acc: 0.4082 - val_loss: 2.4534 - val_acc: 0.3590
     Epoch 45/60
     100/100 [==============================] - 24s 244ms/step - loss: 2.1434 - acc: 0.4198 - val_loss: 2.5464 - val_acc: 0.3478
     Epoch 46/60
     100/100 [==============================] - 24s 245ms/step - loss: 2.1102 - acc: 0.4224 - val_loss: 2.2959 - val_acc: 0.3846
     Epoch 47/60
     100/100 [==============================] - 24s 245ms/step - loss: 2.0798 - acc: 0.4302 - val_loss: 2.2779 - val_acc: 0.3915
     Epoch 48/60
     100/100 [==============================] - 24s 239ms/step - loss: 2.0605 - acc: 0.4414 - val_loss: 2.2463 - val_acc: 0.4042
     Epoch 49/60
     100/100 [==============================] - 24s 243ms/step - loss: 2.0502 - acc: 0.4437 - val_loss: 2.3496 - val_acc: 0.3866
     Epoch 50/60
     100/100 [==============================] - 25s 246ms/step - loss: 2.0018 - acc: 0.4553 - val_loss: 2.2062 - val_acc: 0.4097
     Epoch 51/60
     100/100 [==============================] - 27s 268ms/step - loss: 2.0028 - acc: 0.4531 - val_loss: 2.5857 - val_acc: 0.3634
     Epoch 52/60
     100/100 [==============================] - 24s 240ms/step - loss: 1.9679 - acc: 0.4559 - val_loss: 2.3966 - val_acc: 0.3918
     Epoch 53/60
     100/100 [==============================] - 24s 243ms/step - loss: 1.9531 - acc: 0.4657 - val_loss: 2.2679 - val_acc: 0.4022
     Epoch 54/60
     100/100 [==============================] - 27s 268ms/step - loss: 1.9080 - acc: 0.4780 - val_loss: 2.1134 - val_acc: 0.4288
```

```
Epoch 54/60
100/100 [==============================] - 27s 268ms/step - loss: 1.9080 - acc: 0.4780 - val_loss: 2.1134 - val_acc: 0.4288
Epoch 55/60
100/100 [==============================] - 24s 244ms/step - loss: 1.9371 - acc: 0.4701 - val_loss: 2.5957 - val_acc: 0.3568
Epoch 56/60
100/100 [==============================] - 25s 246ms/step - loss: 1.8880 - acc: 0.4830 - val_loss: 2.5936 - val_acc: 0.3663
Epoch 57/60
100/100 [==============================] - 24s 240ms/step - loss: 1.8816 - acc: 0.4813 - val_loss: 2.1124 - val_acc: 0.4366
Epoch 58/60
100/100 [==============================] - 25s 248ms/step - loss: 1.8721 - acc: 0.4835 - val_loss: 2.0874 - val_acc: 0.4399
Epoch 59/60
100/100 [==============================] - 24s 243ms/step - loss: 1.8384 - acc: 0.4921 - val_loss: 2.4626 - val_acc: 0.3813
Epoch 60/60
100/100 [==============================] - 25s 247ms/step - loss: 1.8168 - acc: 0.4923 - val_loss: 2.2882 - val_acc: 0.4103
Training time: 1883.1435627937317
```

4.6. Training and Validation Accuracies



4.7. Training and Validation Loss

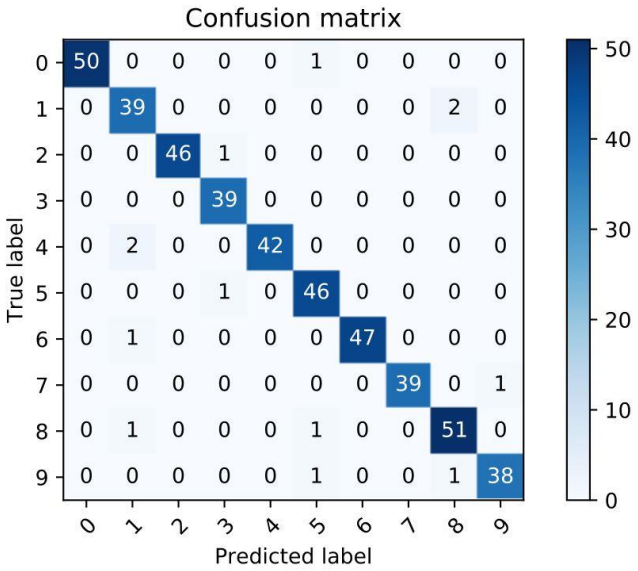


4.8. Model Performance and Accuracy

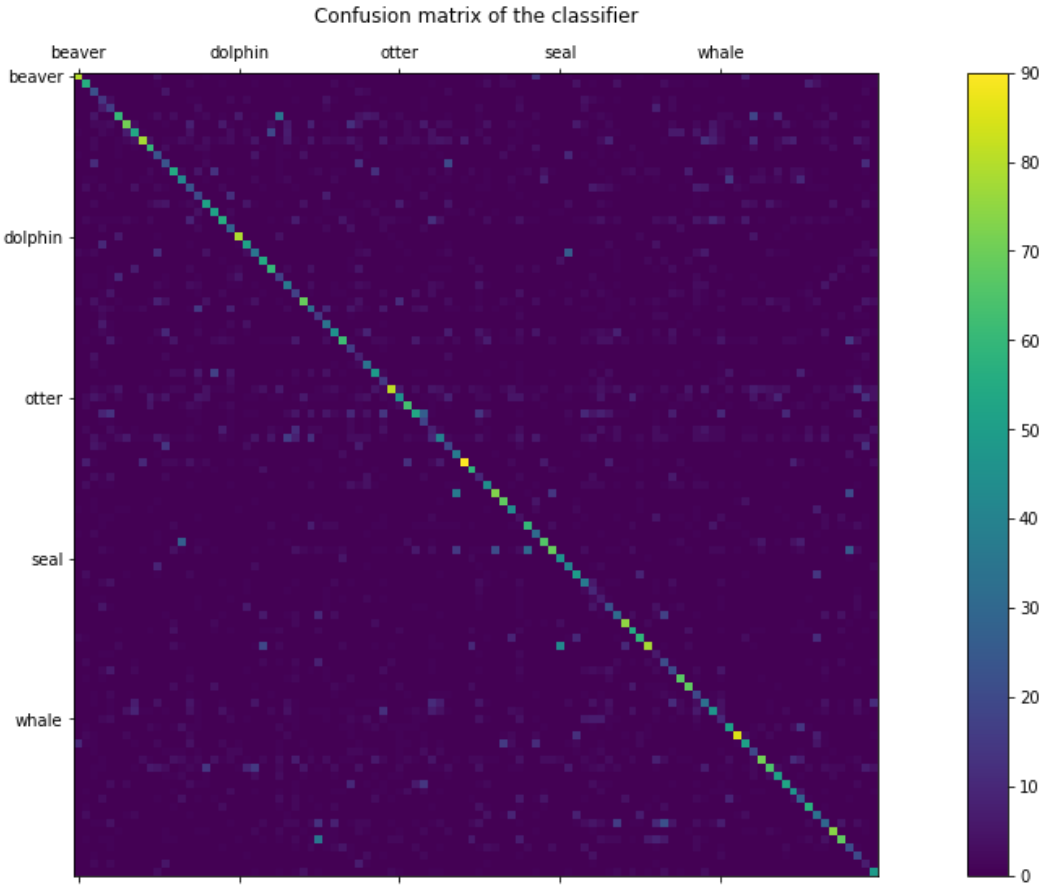
```
[ ]  #test models on testing data
     scores = model.evaluate (test_images, test_labels)
     print (f'accuracy on test set: {model.metrics_names [1]} of {scores [1]*100}')

     313/313 [==============================] - 7s 23ms/step - loss: 2.2196 - acc: 0.4280
     accuracy on test set: acc of 42.80000030994415
```

4.9. Confusion Matrix

Confusion Matrix for multiclass classification is used to know the performance of a Machine learning classification. It is represented in a matrix form. Confusion Matrix gives a comparison between Actual and predicted values. The confusion matrix is a N x N matrix, where N is the number of classes or outputs.



4.9.1. Output Confusion Matrix



4.10. Comment

On our first trial, Jupyter notebook was used with our device CPU capabilities and as a result, the model training took too much time; it took approximately 3 hours and 15 minutes and the output graph was terrible and inaccurate so on a second trial we decided to use Google Collab with the aid of our device GPU and results were mind-blowing; even with a higher number of epochs (40 vs 30 on the 1st trial), the model finished training and we got our outputs in under 20 minutes with a much precise graph and overall improved results.

APPENDIX

The entire code can be found in this link: [MV_Milestone02]

```
import tensorflow as tf
```

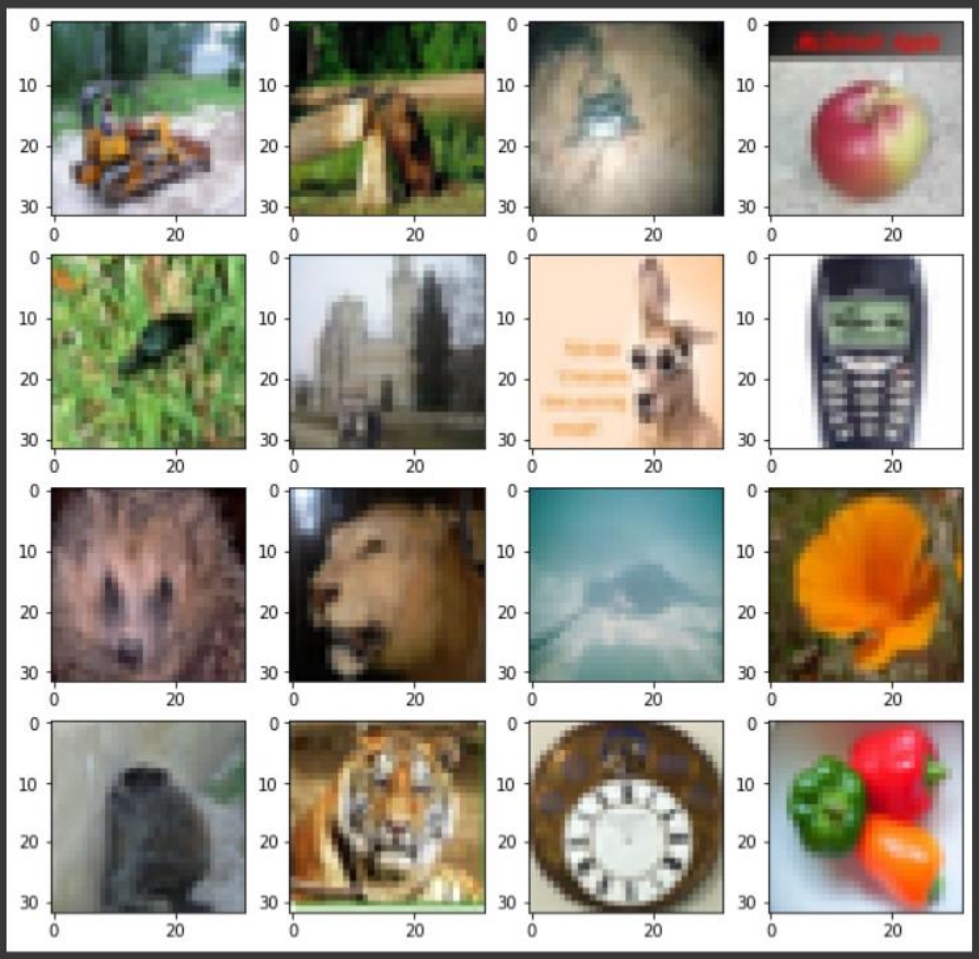
```
[ ] device_list = tf.test.gpu_device_name()
```

```
[ ] device_list
```

```
'/device:GPU:0'
```

```
[ ] from keras.datasets import cifar100
    from keras.models import Sequential
    #cnn related
    from keras.layers import Dense, Dropout, Conv2D, MaxPool2D, Flatten, Activation, Dropout
    from keras.utils import np_utils
    from tensorflow.keras.utils import to_categorical
    import matplotlib.pyplot as plt
    from tensorflow.keras.preprocessing.image import ImageDataGenerator
    from sklearn.model_selection import train_test_split
    import numpy as np
    from tensorflow.keras.layers import BatchNormalization
    from tensorflow.keras.initializers import RandomNormal, Constant
    # Loading the dataset
    (x_train, y_train), (x_test, y_test) = cifar100.load_data()
    # Check the shape of the array
    print(f"x_train shape: {x_train.shape}")



    print(f"Train: {y_train.shape[0]}")
    print(f"Test: {x_test.shape [0]}")
    # Data format
    print(type(x_train))
    print(type(y_train))
```

```
[ ]  plt.figure(figsize= (10, 10))
     for i in range (16):
       rand_num = np.random. randint (0, 50000)
       cifar_img = plt.subplot (4,4, i+1)
       plt.imshow(x_train[rand_num])
```



```
     #normalize image
     train_images = x_train.astype('float32')/255
     test_images = x_test.astype('float32')/255
     print("Shape before one-hot encoding: ", y_train.shape)
     # Transform labels to one hot encoding
     train_labels = to_categorical(y_train)
     test_labels = to_categorical (y_test)
     print("Shape after one-hot encoding: ", train_labels.shape)
```

```
Shape before one-hot encoding:  (50000, 1)
Shape after one-hot encoding:  (50000, 100)
```

```
[ ]  train_datagen = ImageDataGenerator (rotation_range=20, horizontal_flip=True)
     X_train, X_validation, Y_train, Y_validation = train_test_split(train_images, train_labels)
     train_datagen.fit(x_train)
```

```
[ ]  model = Sequential ()

     model. add(Conv2D(256, (3, 3) , padding= 'same', input_shape= (32, 32, 3)))
     model.add(BatchNormalization())
     model.add(Activation('relu'))
     model.add(Conv2D (256, (3, 3) ,padding= 'same' ))
     model.add(BatchNormalization ( ))
     model.add(Activation('relu'))
     model.add(MaxPool2D (pool_size=(2,2)))
     model. add (Dropout (0.2))

     model.add(Conv2D(512, (3,3),padding='same'))
     model.add(BatchNormalization())
     model.add(Activation('relu'))
     model.add(Conv2D(512, (3,3),padding='same'))
     model.add (BatchNormalization())
     model.add(Activation('relu'))
     model.add(MaxPool2D (pool_size=(2,2)))
     model.add(Dropout (0.2))

     model.add(Conv2D(512, (3,3),padding='same'))
     model.add(BatchNormalization ())
     model.add(Activation('relu'))
     model.add(Conv2D(512, (3,3),padding='same'))
     model.add(BatchNormalization ())
     model.add(Activation('relu'))
     model.add(MaxPool2D(pool_size=(2,2)))
     model.add(Dropout (0.2))

     model.add(Conv2D (512, (3,3),padding= 'same'))
     model.add(BatchNormalization())
     model.add(Activation('relu'))
     model. add(Conv2D (512, (3, 3) , padding= 'same'))
     model.add(BatchNormalization ( ))
     model.add(Activation('relu'))
     model.add(MaxPool2D(pool_size= (2,2)))
     model. add(Dropout (0.2))

     model. add(Conv2D (512, (3,3) ,padding= 'same' ))
     model.add(BatchNormalization ( ))
     model.add(Activation('relu'))
     model.add(Conv2D(512, (3,3),padding='same'))
     model.add(BatchNormalization ())
     model.add(Activation('relu'))
     model.add(MaxPool2D(pool_size= (2,2)))
     model.add(Dropout (0.2))

     model. add(Flatten ())
     model. add(Dense(1024))
     model.add(Activation('relu'))
     model.add(Dropout (0.2))
     model.add(BatchNormalization (momentum=0.95,
       epsilon=0.005,
       beta_initializer-RandomNormal (mean-0.0, stddev-0.05),
       gamma_initializer=Constant(value=0.9)))
     model.add(Dense(100,activation='softmax'))
     model.summary()
```

```
[ ]  #from keras import optimizers
     #import keras
     from tensorflow.keras import optimizers
     model.compile(loss='categorical_crossentropy',optimizer=optimizers.RMSprop(learning_rate=1e-5), metrics=['acc'] )
```

```
[ ]  # Training model
     import time
     training_start = time.time()
     history = model.fit(train_datagen.flow(X_train, Y_train, batch_size=100), steps_per_epoch=100,epochs=60,
               validation_data=(X_validation, Y_validation),
               verbose=1)
     training_stop = time.time()
     training_time = training_stop - training_start
     print( f"Training time: {training_time}")
```

```
[ ]  def plot (history):
       acc = history.history['acc']
       val_acc = history. history[ 'val_acc']
       loss = history.history['loss']
       val_loss = history.history[ 'val_loss']
       epochs = range(1, len(acc) + 1)
       plt.plot(epochs, acc, color='red'  , label='Training acc')
       plt.plot(epochs, val_acc, color='black', label= 'Validation acc')
       plt.title('Training and validation accuracy')
       plt.legend()
       plt.figure()
       plt.plot(epochs, loss, color='red', label='Training loss')
       plt.plot(epochs, val_loss, color='black', label='Validation loss')
       plt.title('Training and validation loss')
       plt.legend()
       plt.show()
```

```
[ ]  # Visualize training process
     plot(history)
```

```
[ ]  #test models on testing data
     scores = model.evaluate (test_images, test_labels)
     print (f'accuracy on test set: {model.metrics_names [1]} of {scores [1]*100}')
```

```
313/313 [==============================] - 7s 23ms/step - loss: 2.2196 - acc: 0.4280
accuracy on test set: acc of 42.80000030994415
```

```
     # Translate categorial to array for drawing confusion matrix
     from sklearn.metrics import confusion_matrix
     from numpy import argmax
     prediction = []
     true_labels = []
     pred = model.predict(test_images)
     print(test_labels.shape[0])
     for i in range(test_labels.shape[0]):
       prediction.append(argmax(pred[i]))
       true_labels.append(argmax(test_labels[i]))

     cm = confusion_matrix(prediction, true_labels)
```

```
313/313 [==============================] - 6s 19ms/step
10000
```

```
[ ]  #Name of all classes in CIFAR-100
     classes = ['beaver',
     'dolphin', 'otter', 'seal', 'whale',
     'aquarium' ,'fish', 'ray', 'shark', 'trout',
     'orchids', 'poppies', 'roses', 'sunflowers', 'tulips',
     'bottles','bowls', 'cans', 'cups', 'plates',
     'apples','mushrooms', 'oranges', 'pears', 'sweet peppers',
     'clock' , 'computer keyboard','bee'
     'beetle','lamp','telephone' , 'television', 'bed', 'chair', 'cou'
     'butterfly','caterpillar', 'cockroach','bear'
     'leopard','lion', 'tiger','bridge'
     'castle','house', 'road',
     'wolf','skyscraper','cloud'
     'forest','camel','mountain',
     'cattle','plain','fox','chimpanzee','elephant','porcupine'
     'possum', 'sea', 'kangaroo',
     'crab','lobster','snail',
     'raccoon','spider','skunk','worm','girl',
     'baby', 'boy', 'dinosaur', 'lizard',
     'man', 'woman',
     'rocodile','mouse', 'rabbit', 'shrew','snake', 'turtle',
     'hamster',
     'squirrel',
     'bicycle', 'bus', 'motorcycle', 'pickup truck', 'train',
     'lawn-mower', 'rocket', 'streetcar', 'tank', 'tractor']
```

```
[ ]  # Plot the confusion matrix
     import matplotlib.pyplot as plt
     print(cm)
     fig = plt.figure(figsize= (20, 20))
     ax = fig.add_subplot (211)
     cax = ax.matshow(cm)
     plt.title('Confusion matrix of the classifier')
     fig.colorbar(cax)
     ax.set_xticklabels([''] + classes)
     ax.set_yticklabels([''] + classes)
     pit.xlabel('Predicted')
     plt.ylabel('True')
     plt.show()
```