**TASK 1.**

**import numpy as np**

**# global variables**

**BOARD\_ROWS = 3**

**BOARD\_COLS = 4**

**WIN\_STATE = (0, 3)**

**LOSE\_STATE = (1, 3)**

**START = (2, 0)**

**DETERMINISTIC = True**

**class State:**

**def \_\_init\_\_(self, state=START):**

**self.board = np.zeros([BOARD\_ROWS, BOARD\_COLS])**

**self.board[1, 1] = -1**

**self.state = state**

**self.isEnd = False**

**self.determine = DETERMINISTIC**

**def giveReward(self):**

**if self.state == WIN\_STATE:**

**return 1**

**elif self.state == LOSE\_STATE:**

**return -1**

**else:**

**return 0**

**def isEndFunc(self):**

**if (self.state == WIN\_STATE) or (self.state == LOSE\_STATE):**

**self.isEnd = True**

**def nxtPosition(self, action):**

**"""**

**action: up, down, left, right**

**-------------**

**0 | 1 | 2| 3|**

**1 |**

**2 |**

**return next position**

**"""**

**if self.determine:**

**if action == "up":**

**nxtState = (self.state[0] - 1, self.state[1])**

**elif action == "down":**

**nxtState = (self.state[0] + 1, self.state[1])**

**elif action == "left":**

**nxtState = (self.state[0], self.state[1] - 1)**

**else:**

**nxtState = (self.state[0], self.state[1] + 1)**

**# if next state legal**

**if (nxtState[0] >= 0) and (nxtState[0] <= (BOARD\_ROWS -1)):**

**if (nxtState[1] >= 0) and (nxtState[1] <= (BOARD\_COLS -1)):**

**if nxtState != (1, 1):**

**return nxtState**

**return self.state**

**def showBoard(self):**

**self.board[self.state] = 1**

**for i in range(0, BOARD\_ROWS):**

**print('-----------------')**

**out = '| '**

**for j in range(0, BOARD\_COLS):**

**if self.board[i, j] == 1:**

**token = '\*'**

**if self.board[i, j] == -1:**

**token = 'z'**

**if self.board[i, j] == 0:**

**token = '0'**

**out += token + ' | '**

**print(out)**

**print('-----------------')**

**# Agent of player**

**class Agent:**

**def \_\_init\_\_(self):**

**self.states = []**

**self.actions = ["up", "down", "left", "right"]**

**self.State = State()**

**self.lr = 0.2**

**self.exp\_rate = 0.3**

**# initial state reward**

**self.state\_values = {}**

**for i in range(BOARD\_ROWS):**

**for j in range(BOARD\_COLS):**

**self.state\_values[(i, j)] = 0 # set initial value to 0**

**def chooseAction(self):**

**# choose action with most expected value**

**mx\_nxt\_reward = 0**

**action = ""**

**if np.random.uniform(0, 1) <= self.exp\_rate:**

**action = np.random.choice(self.actions)**

**else:**

**# greedy action**

**for a in self.actions:**

**# if the action is deterministic**

**nxt\_reward = self.state\_values[self.State.nxtPosition(a)]**

**if nxt\_reward >= mx\_nxt\_reward:**

**action = a**

**mx\_nxt\_reward = nxt\_reward**

**return action**

**def takeAction(self, action):**

**position = self.State.nxtPosition(action)**

**return State(state=position)**

**def reset(self):**

**self.states = []**

**self.State = State()**

**def play(self, rounds=10):**

**i = 0**

**while i < rounds:**

**# to the end of game back propagate reward**

**if self.State.isEnd:**

**# back propagate**

**reward = self.State.giveReward()**

**# explicitly assign end state to reward values**

**self.state\_values[self.State.state] = reward # this is optional**

**print("Game End Reward", reward)**

**for s in reversed(self.states):**

**reward = self.state\_values[s] + self.lr \* (reward - self.state\_values[s])**

**self.state\_values[s] = round(reward, 3)**

**self.reset()**

**i += 1**

**else:**

**action = self.chooseAction()**

**# append trace**

**self.states.append(self.State.nxtPosition(action))**

**print("current position {} action {}".format(self.State.state, action))**

**# by taking the action, it reaches the next state**

**self.State = self.takeAction(action)**

**# mark is end**

**self.State.isEndFunc()**

**print("nxt state", self.State.state)**

**print("---------------------")**

**def showValues(self):**

**for i in range(0, BOARD\_ROWS):**

**print('----------------------------------')**

**out = '| '**

**for j in range(0, BOARD\_COLS):**

**out += str(self.state\_values[(i, j)]).ljust(6) + ' | '**

**print(out)**

**print('----------------------------------')**

**if \_\_name\_\_ == "\_\_main\_\_":**

**ag = Agent()**

**ag.play(50)**

**print(ag.showValues())**

**Task 2.**

The presence of outliers in the records units influences the shape of multicollinearity which arises from an excessive diploma of correlation between explanatory variables in a linear regression analysis. This have an effect on ought to be viewed as an expand or minimize in the diagnostics used to decide multicollinearity. Thus, the instances of outliers minimize the reliability of diagnostics such as variance inflation factors, situation numbers and variance decomposition proportions. In this study, we endorse to use a strong estimation of the correlation matrix received with the aid of the minimal covariance determinant technique to decide the diagnostics of multicollinearity in the presence of outliers. As a result, the current paper demonstrates that the diagnostics of multicollinearity bought by using the strong estimation of the correlation matrix are extra dependable in the presence of outliers.

**Task 3.**

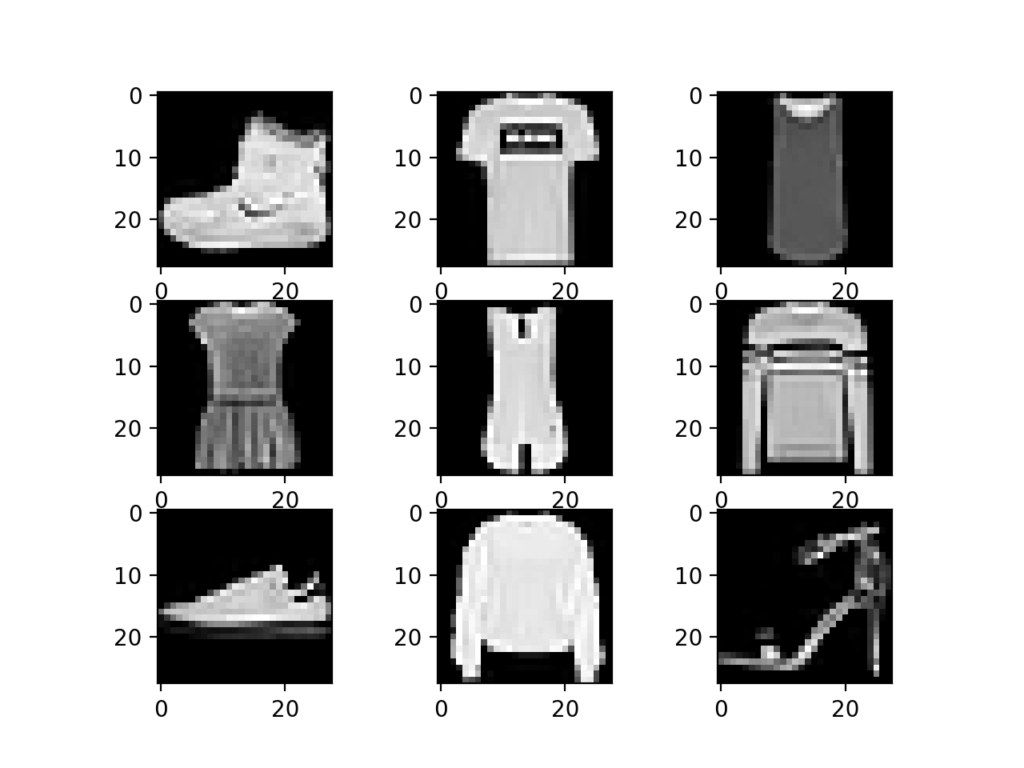
**Classification of MNIST Clothing.**

**Loading the Dataset**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16** | **# example of loading the fashion mnist dataset**  **from matplotlib import pyplot**  **from keras.datasets import fashion\_mnist**  **# load dataset**  **(trainX, trainy), (testX, testy) = fashion\_mnist.load\_data()**  **# summarize loaded dataset**  **print('Train: X=%s, y=%s' % (trainX.shape, trainy.shape))**  **print('Test: X=%s, y=%s' % (testX.shape, testy.shape))**  **# plot first few images**  **for i in range(9):**  **# define subplot**  **pyplot.subplot(330 + 1 + i)**  **# plot raw pixel data**  **pyplot.imshow(trainX[i], cmap=pyplot.get\_cmap('gray'))**  **# show the figure**  **pyplot.show()** |

|  |  |
| --- | --- |
| **1**  **2** | **Train: X=(60000, 28, 28), y=(60000,)**  **Test: X=(10000, 28, 28), y=(10000,)** |

**The plot gives 9 images shown below**

****

**We have to convert integers to floats**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6** | **# convert from integers to floats**  **train\_norm = train.astype('float32')**  **test\_norm = test.astype('float32')**  **# normalize to range 0-1**  **train\_norm = train\_norm / 255.0**  **test\_norm = test\_norm / 255.0** |

**Scaling the pixels**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10** | **# scale pixels**  **def prep\_pixels(train, test):**  **# convert from integers to floats**  **train\_norm = train.astype('float32')**  **test\_norm = test.astype('float32')**  **# normalize to range 0-1**  **train\_norm = train\_norm / 255.0**  **test\_norm = test\_norm / 255.0**  **# return normalized images**  **return train\_norm, test\_norm** |

**Defining the CNN model**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12** | **# define cnn model**  **def define\_model():**  **model = Sequential()**  **model.add(Conv2D(32, (3, 3), activation='relu', kernel\_initializer='he\_uniform', input\_shape=(28, 28, 1)))**  **model.add(MaxPooling2D((2, 2)))**  **model.add(Flatten())**  **model.add(Dense(100, activation='relu', kernel\_initializer='he\_uniform'))**  **model.add(Dense(10, activation='softmax'))**  **# compile model**  **opt = SGD(lr=0.01, momentum=0.9)**  **model.compile(optimizer=opt, loss='categorical\_crossentropy', metrics=['accuracy'])**  **return model** |

**Evaluating the Model**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20** | **# evaluate a model using k-fold cross-validation**  **def evaluate\_model(dataX, dataY, n\_folds=5):**  **scores, histories = list(), list()**  **# prepare cross validation**  **kfold = KFold(n\_folds, shuffle=True, random\_state=1)**  **# enumerate splits**  **for train\_ix, test\_ix in kfold.split(dataX):**  **# define model**  **model = define\_model()**  **# select rows for train and test**  **trainX, trainY, testX, testY = dataX[train\_ix], dataY[train\_ix], dataX[test\_ix], dataY[test\_ix]**  **# fit model**  **history = model.fit(trainX, trainY, epochs=10, batch\_size=32, validation\_data=(testX, testY), verbose=0)**  **# evaluate model**  **\_, acc = model.evaluate(testX, testY, verbose=0)**  **print('> %.3f' % (acc \* 100.0))**  **# append scores**  **scores.append(acc)**  **histories.append(history)**  **return scores, histories** |

**Plotting Diagnostic learning curves**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14** | **# plot diagnostic learning curves**  **def summarize\_diagnostics(histories):**  **for i in range(len(histories)):**  **# plot loss**  **pyplot.subplot(211)**  **pyplot.title('Cross Entropy Loss')**  **pyplot.plot(histories[i].history['loss'], color='blue', label='train')**  **pyplot.plot(histories[i].history['val\_loss'], color='orange', label='test')**  **# plot accuracy**  **pyplot.subplot(212)**  **pyplot.title('Classification Accuracy')**  **pyplot.plot(histories[i].history['accuracy'], color='blue', label='train')**  **pyplot.plot(histories[i].history['val\_accuracy'], color='orange', label='test')**  **pyplot.show()** |

**Summarizing the performance of the model**

**.**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7** | **# summarize model performance**  **def summarize\_performance(scores):**  **# print summary**  **print('Accuracy: mean=%.3f std=%.3f, n=%d' % (mean(scores)\*100, std(scores)\*100, len(scores)))**  **# box and whisker plots of results**  **pyplot.boxplot(scores)**  **pyplot.show()** |

**Now we look at the complete Code**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12** | **# run the test harness for evaluating a model**  **def run\_test\_harness():**  **# load dataset**  **trainX, trainY, testX, testY = load\_dataset()**  **# prepare pixel data**  **trainX, testX = prep\_pixels(trainX, testX)**  **# evaluate model**  **scores, histories = evaluate\_model(trainX, trainY)**  **# learning curves**  **summarize\_diagnostics(histories)**  **# summarize estimated performance**  **summarize\_performance(scores)** |

**Baseline Complete Code**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28**  **29**  **30**  **31**  **32**  **33**  **34**  **35**  **36**  **37**  **38**  **39**  **40**  **41**  **42**  **43**  **44**  **45**  **46**  **47**  **48**  **49**  **50**  **51**  **52**  **53**  **54**  **55**  **56**  **57**  **58**  **59**  **60**  **61**  **62**  **63**  **64**  **65**  **66**  **67**  **68**  **69**  **70**  **71**  **72**  **73**  **74**  **75**  **76**  **77**  **78**  **79**  **80**  **81**  **82**  **83**  **84**  **85**  **86**  **87**  **88**  **89**  **90**  **91**  **92**  **93**  **94**  **95**  **96**  **97**  **98**  **99**  **100**  **101**  **102**  **103**  **104**  **105**  **106**  **107**  **108**  **109** | **# baseline cnn model for fashion mnist**  **from numpy import mean**  **from numpy import std**  **from matplotlib import pyplot**  **from sklearn.model\_selection import KFold**  **from keras.datasets import fashion\_mnist**  **from keras.utils import to\_categorical**  **from keras.models import Sequential**  **from keras.layers import Conv2D**  **from keras.layers import MaxPooling2D**  **from keras.layers import Dense**  **from keras.layers import Flatten**  **from keras.optimizers import SGD**    **# load train and test dataset**  **def load\_dataset():**  **# load dataset**  **(trainX, trainY), (testX, testY) = fashion\_mnist.load\_data()**  **# reshape dataset to have a single channel**  **trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))**  **testX = testX.reshape((testX.shape[0], 28, 28, 1))**  **# one hot encode target values**  **trainY = to\_categorical(trainY)**  **testY = to\_categorical(testY)**  **return trainX, trainY, testX, testY**    **# scale pixels**  **def prep\_pixels(train, test):**  **# convert from integers to floats**  **train\_norm = train.astype('float32')**  **test\_norm = test.astype('float32')**  **# normalize to range 0-1**  **train\_norm = train\_norm / 255.0**  **test\_norm = test\_norm / 255.0**  **# return normalized images**  **return train\_norm, test\_norm**    **# define cnn model**  **def define\_model():**  **model = Sequential()**  **model.add(Conv2D(32, (3, 3), activation='relu', kernel\_initializer='he\_uniform', input\_shape=(28, 28, 1)))**  **model.add(MaxPooling2D((2, 2)))**  **model.add(Flatten())**  **model.add(Dense(100, activation='relu', kernel\_initializer='he\_uniform'))**  **model.add(Dense(10, activation='softmax'))**  **# compile model**  **opt = SGD(lr=0.01, momentum=0.9)**  **model.compile(optimizer=opt, loss='categorical\_crossentropy', metrics=['accuracy'])**  **return model**    **# evaluate a model using k-fold cross-validation**  **def evaluate\_model(dataX, dataY, n\_folds=5):**  **scores, histories = list(), list()**  **# prepare cross validation**  **kfold = KFold(n\_folds, shuffle=True, random\_state=1)**  **# enumerate splits**  **for train\_ix, test\_ix in kfold.split(dataX):**  **# define model**  **model = define\_model()**  **# select rows for train and test**  **trainX, trainY, testX, testY = dataX[train\_ix], dataY[train\_ix], dataX[test\_ix], dataY[test\_ix]**  **# fit model**  **history = model.fit(trainX, trainY, epochs=10, batch\_size=32, validation\_data=(testX, testY), verbose=0)**  **# evaluate model**  **\_, acc = model.evaluate(testX, testY, verbose=0)**  **print('> %.3f' % (acc \* 100.0))**  **# append scores**  **scores.append(acc)**  **histories.append(history)**  **return scores, histories**    **# plot diagnostic learning curves**  **def summarize\_diagnostics(histories):**  **for i in range(len(histories)):**  **# plot loss**  **pyplot.subplot(211)**  **pyplot.title('Cross Entropy Loss')**  **pyplot.plot(histories[i].history['loss'], color='blue', label='train')**  **pyplot.plot(histories[i].history['val\_loss'], color='orange', label='test')**  **# plot accuracy**  **pyplot.subplot(212)**  **pyplot.title('Classification Accuracy')**  **pyplot.plot(histories[i].history['accuracy'], color='blue', label='train')**  **pyplot.plot(histories[i].history['val\_accuracy'], color='orange', label='test')**  **pyplot.show()**    **# summarize model performance**  **def summarize\_performance(scores):**  **# print summary**  **print('Accuracy: mean=%.3f std=%.3f, n=%d' % (mean(scores)\*100, std(scores)\*100, len(scores)))**  **# box and whisker plots of results**  **pyplot.boxplot(scores)**  **pyplot.show()**    **# run the test harness for evaluating a model**  **def run\_test\_harness():**  **# load dataset**  **trainX, trainY, testX, testY = load\_dataset()**  **# prepare pixel data**  **trainX, testX = prep\_pixels(trainX, testX)**  **# evaluate model**  **scores, histories = evaluate\_model(trainX, trainY)**  **# learning curves**  **summarize\_diagnostics(histories)**  **# summarize estimated performance**  **summarize\_performance(scores)**    **# entry point, run the test harness**  **run\_test\_harness()** |

**Results**

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5** | **> 91.200**  **> 91.217**  **> 90.958**  **> 91.242**  **> 91.317** |

**Task 4.**

**Convolutional Neural implementation with PyTorch on CIFAR-10 Dataset**

**Importing the PyTorch Library**

**import numpy as np**

**import pandas as pd**

**import torch**

**import torch.nn.functional as F**

**from torchvision import datasets,transforms**

**from torch import nn**

**import matplotlib.pyplot as plt**

**import numpy as np**

**import seaborn as sns**

**#from tqdm.notebook import tqdm**

**Reading the required Dataset**

**trainData = pd.read\_csv('cifar-10/trainLabels.csv')**

**trainData.head()**

**Analysis**

**print("Number of points:",trainData.shape[0])**

**print("Number of features:",trainData.shape[1])**

**print("Features:",trainData.columns.values)**

**print("Number of Unique Values")**

**for col in trainData:**

**print(col,":",len(trainData[col].unique()))**

**plt.figure(figsize=(12,8))**

**Output:**

**Number of points: 50000**

**Number of features: 2**

**Features: ['id' 'label']**

**Number of Unique Values**

**id : 50000**

**label : 10**

**Validation**

**from torch.utils.data import random\_split**

**val\_size = 5000**

**train\_size = len(dataset) - val\_size**

**train\_ds, val\_ds = random\_split(dataset, [train\_size, val\_size])**

**len(train\_ds), len(val\_ds)**

**(45000, 5000)**

**from torch.utils.data.dataloader import DataLoader**

**batch\_size=64**

**train\_dl = DataLoader(train\_ds, batch\_size, shuffle=True, num\_workers=4, pin\_memory=True)**

**val\_dl = DataLoader(val\_ds, batch\_size, num\_workers=4, pin\_memory=True)**

**Required functions definition**

**@torch.no\_grad()**

**def accuracy(outputs, labels):**

**\_, preds = torch.max(outputs, dim=1)**

**return torch.tensor(torch.sum(preds == labels).item() / len(preds))**

**class ImageClassificationBase(nn.Module):**

**def training\_step(self, batch):**

**images, labels = batch**

**out = self(images) # Generate predictions**

**loss = F.cross\_entropy(out, labels) # Calculate loss**

**accu = accuracy(out,labels)**

**return loss,accu**

**def validation\_step(self, batch):**

**images, labels = batch**

**out = self(images) # Generate predictions**

**loss = F.cross\_entropy(out, labels) # Calculate loss**

**acc = accuracy(out, labels) # Calculate accuracy**

**return {'Loss': loss.detach(), 'Accuracy': acc}**

**def validation\_epoch\_end(self, outputs):**

**batch\_losses = [x['Loss'] for x in outputs]**

**epoch\_loss = torch.stack(batch\_losses).mean() # Combine losses**

**batch\_accs = [x['Accuracy'] for x in outputs]**

**epoch\_acc = torch.stack(batch\_accs).mean() # Combine accuracies**

**return {'Loss': epoch\_loss.item(), 'Accuracy': epoch\_acc.item()}**

**def epoch\_end(self, epoch, result):**

**print("Epoch :",epoch + 1)**

**print(f'Train Accuracy:{result["train\_accuracy"]\*100:.2f}% Validation Accuracy:{result["Accuracy"]\*100:.2f}%')**

**print(f'Train Loss:{result["train\_loss"]:.4f} Validation Loss:{result["Loss"]:.4f}')**

**Implementing the convolutional neural network module**

**class Cifar10CnnModel(ImageClassificationBase):**

**def \_\_init\_\_(self):**

**super().\_\_init\_\_()**

**self.network = nn.Sequential(**

**nn.Conv2d(3, 32, kernel\_size=3, padding=1),**

**nn.ReLU(),**

**nn.Conv2d(32, 64, kernel\_size=3, stride=1, padding=1),**

**nn.ReLU(),**

**nn.MaxPool2d(2, 2), # output: 64 x 16 x 16**

**nn.BatchNorm2d(64),**

**nn.Conv2d(64, 128, kernel\_size=3, stride=1, padding=1),**

**nn.ReLU(),**

**nn.Conv2d(128, 128, kernel\_size=3, stride=1, padding=1),**

**nn.ReLU(),**

**nn.MaxPool2d(2, 2), # output: 128 x 8 x 8**

**nn.BatchNorm2d(128),**

**nn.Conv2d(128, 256, kernel\_size=3, stride=1, padding=1),**

**nn.ReLU(),**

**nn.Conv2d(256, 256, kernel\_size=3, stride=1, padding=1),**

**nn.ReLU(),**

**nn.MaxPool2d(2, 2), # output: 256 x 4 x 4**

**nn.BatchNorm2d(256),**

**nn.Flatten(),**

**nn.Linear(256\*4\*4, 1024),**

**nn.ReLU(),**

**nn.Linear(1024, 512),**

**nn.ReLU(),**

**nn.Linear(512, 10))**

**def forward(self, xb):**

**return self.network(xb)**

**Training the model**

**@torch.no\_grad()**

**def evaluate(model, data\_loader):**

**model.eval()**

**outputs = [model.validation\_step(batch) for batch in data\_loader]**

**return model.validation\_epoch\_end(outputs)**

**def fit(model, train\_loader, val\_loader,epochs=10,learning\_rate=0.001):**

**best\_valid = None**

**history = []**

**optimizer = torch.optim.Adam(model.parameters(), learning\_rate,weight\_decay=0.0005)**

**for epoch in range(epochs):**

**# Training Phase**

**model.train()**

**train\_losses = []**

**train\_accuracy = []**

**for batch in tqdm(train\_loader):**

**loss,accu = model.training\_step(batch)**

**train\_losses.append(loss)**

**train\_accuracy.append(accu)**

**loss.backward()**

**optimizer.step()**

**optimizer.zero\_grad()**

**# Validation phase**

**result = evaluate(model, val\_loader)**

**result['train\_loss'] = torch.stack(train\_losses).mean().item()**

**result['train\_accuracy'] = torch.stack(train\_accuracy).mean().item()**

**model.epoch\_end(epoch, result)**

**if(best\_valid == None or best\_valid<result['Accuracy']):**

**best\_valid=result['Accuracy']**

**torch.save(model.state\_dict(), 'cifar10-cnn.pth')**

**history.append(result)**

**return history**

**history = fit(model, train\_dl, val\_dl)**

**Plotting the results**

**def plot\_accuracies(history):**

**Validation\_accuracies = [x['Accuracy'] for x in history]**

**Training\_Accuracies = [x['train\_accuracy'] for x in history]**

**plt.plot(Training\_Accuracies, '-rx')**

**plt.plot(Validation\_accuracies, '-bx')**

**plt.xlabel('epoch')**

**plt.ylabel('accuracy')**

**plt.legend(['Training', 'Validation'])**

**plt.title('Accuracy vs. No. of epochs');**

**plot\_accuracies(history)**

**def plot\_losses(history):**

**train\_losses = [x.get('train\_loss') for x in history]**

**val\_losses = [x['Loss'] for x in history]**

**plt.plot(train\_losses, '-bx')**

**plt.plot(val\_losses, '-rx')**

**plt.xlabel('epoch')**

**plt.ylabel('loss')**

**plt.legend(['Training', 'Validation'])**

**plt.title('Loss vs. No. of epochs');**

**plot\_losses(history)**

**Getting the accuracy**

**test\_dataset = ImageFolder(data\_dir+'/test', transform=ToTensor())**

**test\_loader = DeviceDataLoader(DataLoader(test\_dataset, batch\_size), device)**

**result = evaluate(final\_model, test\_loader)**

**print(f'Test Accuracy:{result["Accuracy"]\*100:.2f}%')**

**Test Accuracy:81.07%**

**With that the result is 81.07% accurate**

References.

Shamsuddin, M. R., Abdul-Rahman, S., & Mohamed, A. (2018, August). Exploratory analysis of MNIST handwritten digit for machine learning modelling. In *International Conference on Soft Computing in Data Science* (pp. 134-145). Springer, Singapore.

Cheng, K., Tahir, R., Eric, L. K., & Li, M. (2020). An analysis of generative adversarial networks and variants for image synthesis on MNIST dataset. *Multimedia Tools and Applications*, *79*(19), 13725-13752.