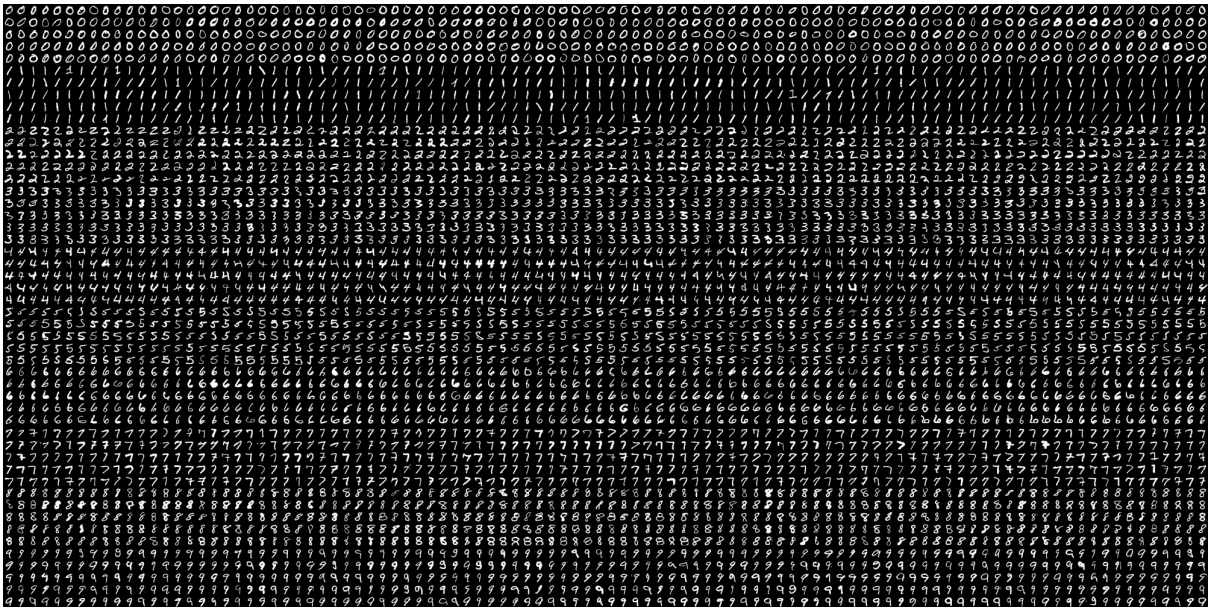
COMP3007 Machine Perception Assignment Report

Cover Page



Name: Zhenqi Zhang
Student ID: 20080833
Class: Wednesday 17:00 - 18:00

About Report

1. Source Code

2. Documentation - Each Task Contains the Following

    2.1. Statements on how much you have attempted the assignment for each task and summarize the performance and your finding

    2.2. The detail of your implementation for each task: this must clearly indicate your approach, and how the features you extract, the methods you use for model selection. It must allow the marker to understand how you approach the machine learning tasks. If hyper-parameters need to be selected, you need to split the training dataset into two subsets: one for training and the other for hyper-parameter selection.

    2.3 The Performance of your program on the Training and the Testing Datasets.

    2.4 Supporting Diagrams, Figures, Tables that help describe your programs and performance clearly

    2.5 Reference that your implementation is based on or inspired from.

3. Some Add on Explanation (KNN, KMean, SVM, CNN)

Source Code

```
#Importing some useful packages

import numpy as np #N umpy library provides various useful functions
and operators for scientific computing
import cv2 as cv # openCV is a key library that provides various useful
functions for computer vision
import os # Honestly this one is a bit optional.
import glob # again just optional
from matplotlib import pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay #
To display the Confusion Matrix
from google.colab.patches import cv2_imshow
from sklearn.cluster import KMeans
from scipy.spatial import distance

path = '/content/drive/MyDrive/MachinePerception/Assignment/' #
Defining the path
img = cv.imread(path+'digits.png')
height, width, channels = img.shape

print(img.shape, "\n") # Shape of the Image

gray = cv.cvtColor(img,cv.COLOR_BGR2GRAY) # This might not be needed
since the image is already in gray scale

cv2_imshow(gray)  # Display the Image

# 100 digits each row
# 5 rows for each digit
# 10 Digits

cv2_imshow(gray[:20,:20])

# Task One
# All Below Code are From Prac 5 Worksheet In COMP3007. Prac 05
Exercise 3

# Now we split the image to 5000 cells, each 20x20 size
# hsplit() into 100 cols and vsplit() into 50 rows
cells = [np.hsplit(row,100) for row in np.vsplit(gray,50)]
```

```
# Make it into a Numpy array: its size will be (50,100,20,20)
x = np.array(cells)

# Now we prepare the training data and test data
test = x[0:50,:20].reshape(-1,400).astype(np.float32) # Testing will be
the First 20 Samples from each Row
train = x[0:50,20:100].reshape(-1,400).astype(np.float32)  # Training
will be the Last 80 Samples of each Row

print(f"The Number of Testing Sample is {test.shape[0]}.")
print(f"\nThe Number of Training Sample is {train.shape[0]}.")

train_path =
'/content/drive/MyDrive/MachinePerception/Assignment/Training'
test_path =
'/content/drive/MyDrive/MachinePerception/Assignment/Testing'

# Function Iterate through the training/Testing samples and save them
to the Training/Testing folder
def save_image(data, path, name):
    for i in range(len(data)):
        digit = data[i].reshape(20, 20).astype(np.uint8)  # Reshape and
convert to uint8
        file_name =
os.path.join('/content/drive/MyDrive/MachinePerception/Assignment/Train
ing', f"{name}_{i}.jpeg")
        cv.imwrite(file_name, digit)

save_image(train, train_path, 'train')
save_image(test, test_path, 'test')

print("\nAll Extracted Image arranged into \"Training\" and \"Testing\"
Folders")

# Create labels for train and test data
k=np.arange(10) ## 2 Classes, 0 to 9
train_labels = np.repeat(k,400)[:,np.newaxis] # Repeat each 0 to 9 400
times, So adding up to 80% of 5000, Which is 4000
test_labels = np.repeat(k,100)[:,np.newaxis]  # Repeat each 0 to 9 100
times, So adding up to 20% of 5000, Which is 1000

# Task Two
```

```
# Define the K Values to be  Tested, Have to be Odd
k_values = [1,3,5,7,8]

# Initialise Lists to store the Accuracy Result and Confusion Matrix
Results for Different K Vlaues
accuracies = []
confusion_matrices = []

# Create and train the KNN classifier
knn = cv.ml.KNearest_create()
knn.train(train, cv.ml.ROW_SAMPLE, train_labels)

# Loop to test different K Values
# From Prac 5 Worksheet In COMP3007. Prac 05 Exercise 3
for k in k_values:

    # Make predictions on the test data
    ret, result, neighbors, dist = knn.findNearest(test, k=k)

    # Calculate accuracy and store it
    matches = result == test_labels
    correct = np.count_nonzero(matches)
    accuracy = correct * 100.0 / result.size
    accuracies.append(accuracy)

    # Calculate and store the confusion matrix
    # Every CM here will be the result of one label. So starting from
digit 0
    # 98 of them are correct and 1 seen as 3, one seen as 6
    cm = confusion_matrix(test_labels, result)
    confusion_matrices.append(cm)

    # # Plot confusion matrices for all K values
    # Idea From :
https://www.w3schools.com/python/python_ml_confusion_matrix.asp
    cm_display = ConfusionMatrixDisplay(confusion_matrix=cm)
    cm_display.plot()
    # Printing out different Test cases for different Hyper-Parameter
and Their accurracies.
    plt.title(f"\nConfusion Matrix for \nHyper-Parameter K={k},
\nAccuracy: {accuracy}\n")
    plt.show()
```

```
# So far the Highest is with the K Value of 5 and an Accuracy of 94.1%

# Quick Result for Further Comparison
print("\nShow all the Accuracy of different K_value:\n")
for i in range(len(k_values)):
    k = k_values[i]
    print(f"The Result For k = {k} is {accuracies[i]}%")

# Find the Highest Overall Accuracy Out of all the K_values
best_k_value = k_values[np.argmax(accuracies)]

# Quick Result for Accuracy for each Label from 0 to 9
print("\nShow all the Accuracy of different Class Labels in for the
Highest K_Value:\n")
for i in range(10):
  print(f"The Result For Class {i} is
{(confusion_matrices[best_k_value-1])[i][i]}% ")

# Another Way to Possibly do the KNN Method (Same Result after Testing)
A Quick Example Below
# from sklearn.neighbors import KNeighborsClassifier # This Part Import
the Library for the KNeighborsClassifier
# knn_1 = KNeighborsClassifier(n_neighbors = 5) # This Part Sets the
Number of K Value
# knn_1.fit(train,train_labels) # Train the Data
# y_pred_1 = knn_1.predict(test)  # Predict Test Image's Label
# print(accuracy_score(test_labels, y_pred_1))  # Calculate the Score
comparing to Predict label and Actual Label

# Task Three
# OvR Classifier Support Vector Machine -
# From Prac 5 Worksheet In COMP3007. Prac 05 Exercise 3
# SVC and OvR Methods from Website:
https://scikit-learn.org/stable/modules/generated/sklearn.multiclass.On
eVsRestClassifier.html#sklearn.multiclass.OneVsRestClassifier.fit

# OvR is to Compare each class with all the other class
# Eg. Compare 0 with 1-9, or Compare 1 with 0 and 2-9 etc.
# In this case, we will need to do it Ten times since we have 10
classes 0-9

# Initialise Lists to store the Accuracy Result and Confusion Matrix
Results for Different K Vlaues
```

```
accuracies = []

# Create the Support Vector Machine
svm = SVC(kernel='linear', max_iter=4000, tol=1e-8, C=1)

# Create a OneVsRestClassifier
classifier = OneVsRestClassifier(svm)

# Train the Classifier on the Training Data
classifier.fit(train, train_labels)

# Loop 10 Times to Go Through all the 10 Digit Classes
for i in range(10):
    # Create a binary label vector for the OvR. The Unique Class Label
been One while changing the Rest to Zero
    binary_test_labels = np.where(test_labels == i, 1, 0)

    # Make predictions on the test data using the Trained Classifier
for the Current Class
    predictions = classifier.predict(test)
    binary_prediction = np.where(predictions == i, 1, 0)

    # Calculate accuracy
    correct = 0
    for j in range(len(binary_prediction)):
        if (binary_prediction[j] == 1) & (binary_test_labels[j] == 1):
              correct += 1
    accuracies.append(correct)

    # Calculate and Plot the Confusion Matrix
    cm = confusion_matrix(binary_test_labels, binary_prediction)
    cm_display = ConfusionMatrixDisplay(confusion_matrix=cm)
    cm_display.plot()
    plt.title(f"Confusion Matrix for Class {i} \nAccuracy: {correct}%")
    plt.show()

# Quick Result for Further Comparison
for i in range(10):
    print(f"The Result For Class {i} is {accuracies[i]}%")

overall = round(sum(accuracies)/10, 2)
print(f"\nOverall Accuracy is {overall}%")
```

```
# Create and train the Support Vector Machine For Multi-Class

svm = cv.ml.SVM_create()
svm.setType(cv.ml.SVM_C_SVC)
svm.setKernel(cv.ml.SVM_LINEAR)
svm.setTermCriteria((cv.TERM_CRITERIA_MAX_ITER, 1000, 1e-8))
svm.train(train, cv.ml.ROW_SAMPLE, train_labels)
response = svm.predict(test)
result = np.uint8(response[1])

cm = confusion_matrix(test_labels, result)
print(cm)

# Calculate accuracy and store it
matches = result==test_labels
correct = np.count_nonzero(matches)
accuracy = correct*100.0/result.size
print( accuracy )

# Task Four
# BoVW Method Using (SIFT/HOG) + KMeans + Histogram + (SVM/KNN)
# Reference From the Given Material -> Website:
https://medium.com/@aybukeyalcinerr/bag-of-visual-words-bovw-db9500331b
2f

# Step One. We have all the 5000 Tiny Digits Extracted in the
tiny_digits List Already, Now we do the SIFT Part.

# A Dictionary to store the Unique Class Digits 0 to 9
train_images = {}
test_images = {}

# Loop 10 times to store the Unique Class (Train Data)
for i in range(10):
    # List to store all the images that are the same class
    image = []
    # The 400 of that Digit to be inserted into the Unique Class
    for j in range(400):
        img = train[(i * 400) + j]
        if img.shape != (20, 20):
            img = img.reshape(20, 20).astype(np.uint8)
        image.append(img)
    train_images[i] = image
```

```
# Loop 10 times to store the Unique Class (Test Data)
for i in range(10):
    # List to store all the images that are the same class
    image = []
    # The 100 of that Digit to be inserted into the Unique Class
    for j in range(100):
        img = test[(i * 100) + j]
        if img.shape != (20, 20):
            img = img.reshape(20, 20).astype(np.uint8)
        image.append(img)
    test_images[i] = image

# Create Function to do SIFT Extraction
# Following Ideas Taken from the Website mentioned above
def sift_features(images):
    # Create the SIFT Detector.
    sift = cv.SIFT_create()
    descriptors_list = [] # List to Hold ALL SIFT Descriptor From ALL
Image.
    sift_vectors = {} # Dictionary Which Store SIFT Vectors by Class.

    # For Every Unique Class Digits
    for key, value in images.items():
        features = []
        # Extract for each Image in that Class
        for img in value:
            # Compute and Detect the Features of the Given Image
            kp, des = sift.detectAndCompute(img, None)

            if kp is not None and des is not None:
                descriptors_list.extend(des)
                features.append(des)
        sift_vectors[key] = features
    return [descriptors_list, sift_vectors]

# Here, I only Implemented the SIFT Function Rather than Having the HOG
Aswell
# It is because that SIFT is Better at Scale Invariant, Roatation
Invariance etc
# And HOG is Also more Complex to Implement And Requires more CPU and
GPU Power to process.
```

```
sifts = sift_features(train_images)
descriptor_list = sifts[0]
all_bovw_feature = sifts[1]
test_bovw_feature = sift_features(test_images)[1]

# Apply KMeans Clustering For the Training Descriptor.
# To Group Similar Data Points into Clusters
def kmeans(k, descriptor_list):
    kmeans = KMeans(n_clusters=k, n_init=10)
    kmeans.fit(descriptor_list)
    visual_words = kmeans.cluster_centers_
    return visual_words

# Here, The KMean Number is Very Important! It is Grouping the Data
into Clusters. And Having a Right Cluster Will help Grouping The Data
into Meaningful Clusters
# But In here, Bigger KMeans also mean that it will Take Longer to
Process and Might not Give the Best Compare to Lower Ones. (Speed and
Efficiency and Accuracy Should be Considered)
# Another Important Thing! KMean Number Should not be More Than The
Amount Of Data Points!!!!

# The Difference Between K = 100 and K = 2500
#   Accruacy Around 10% Difference
#   Time Taken One Can be Within a Minute Or Two While One Can Take Up
to Ten Plus Minutes

k_mean_Number = 2500

# HERE!!! Visual Word Is Equivalent to a Cluster Centroid
visual_words = kmeans(k_mean_Number, descriptor_list)

# This function is to find the Index that is closest the Center ( The
Visual Words )
def find_index(feature_vector, centers):
    #Initialise the Variables to find the Index of the Nearest Center
    closest = float('inf')
    index = -1

    # Loop Though all the Centers
    for i, center in enumerate(centers):
        # Calc the Distance Between the Feature_Vector and the Current
Center
```

```
        dist = np.linalg.norm(feature_vector - center)
        # Check if New Center is Closer than Previous Closest
        if dist < closest:
            closest = dist
            index = i
    return index

# Create the Histograms for the Test and Train Datas
# Following Ideas Taken from the Website mentioned above
def create_histograms(all_bovw, centers):
    # Dictionary on Features for Different Classes
    dict_feature = {}
    for key, value in all_bovw.items():
        category = []
        for img in value:
            histogram = np.zeros(len(centers))
            for each_feature in img:
                ind = find_index(each_feature, centers)
                histogram[ind] += 1
            category.append(histogram)
        dict_feature[key] = category
    return dict_feature

# Create histograms for train data, Used for Training
train_histograms = create_histograms(all_bovw_feature, visual_words)

# Create histograms for test data, Used for Prediction
test_histograms = create_histograms(test_bovw_feature, visual_words)

# Following Ideas Taken from the Website mentioned above
# Directly tooken from Website
def knn(trained, test):
    # Initialise the Count of Test Image and Another for Correct
Prediction.
    num_test = 0
    correct_predict = 0
    # Create a Dictionary to Store Results for Each Class
    class_based = {}

    # Loop through each test image
    for test_key, test_val in test.items():
        # Initialise Correct and Total counts for this Class
        class_based[test_key] = [0, 0]
```

```
        # Loop through each test image in the current class
        for tst in test_val:
            predict_start = 0
            # Variable to store the minimum distance
            minimum = 0
            # Variable to store the predicted class label
            key = "a"

            # Compare the test image with all training images
            for train_key, train_val in trained.items():
                for train in train_val:
                    if predict_start == 0:
                        # If this is the first comparison, set the
minimum distance and predicted class
                        minimum = distance.euclidean(tst, train)
                        key = train_key
                        predict_start += 1
                    else:
                        # For subsequent comparisons, calculate the
distance
                        dist = distance.euclidean(tst, train)
                        if dist < minimum:
                            # If a closer match is found, update the
minimum distance and predicted class
                            minimum = dist
                            key = train_key

            # Check if the Predicted Class Matches the True Class
            if test_key == key:
                # Increase the Count of Correct predictions for this
test image and the Correct Count for this Class
                correct_predict += 1
                class_based[test_key][0] += 1

            # Increase the Total Test Count for this Image and Total
Count for this Class
            num_test += 1
            class_based[test_key][1] += 1

    # Return the total number of test images, total correct
predictions, and class-based results
    return [num_test, correct_predict, class_based]
```

```
# This is Implemented to be Similar to the Above Function But Instead
of KNN, You Use SVM
def svm_classifier(train_histograms, test_histograms):
    # Initialise the Count of Test Image and Another for Correct
Prediction.
    num_test = 0
    correct_predict = 0
    # Create a Dictionary to Store Results for Each Class
    class_based = {}

    # Create an SVM classifier with a linear kernel
    # MAX ITER here is Also Very Important! It is setting the Maximum
Number of Iterations the Algorithm Update its Model
    # Which is used to find the Best Decision Boundary to Separate the
Data Points.
    MAX_ITER = 4000
    svm = SVC(kernel='linear', max_iter=MAX_ITER, tol=1e-8, C=1)

    # Prepare the training data and labels
    X_train = []
    y_train = []
    for label, histograms in train_histograms.items():
        for histogram in histograms:
            X_train.append(histogram)
            y_train.append(label)

    # Train the SVM Classifier on the Training Data
    svm.fit(X_train, y_train)

    # Loop through each Test Image
    for test_key, test_val in test_histograms.items():
        # Initialise correct and total counts for this class
        class_based[test_key] = [0, 0]

        # Loop through each test image in the Current Class
        for histogram in test_val:
            # Predict the Class Label
            predicted_label = svm.predict([histogram])[0]

            # Check if the predicted class matches the true class
            if test_key == predicted_label:
                correct_predict += 1
```

```
                class_based[test_key][0] += 1

            num_test += 1
            class_based[test_key][1] += 1

    # Return the total number of test images, total correct
predictions, and class-based results
    print(f"\nFor K Mean = {k_mean_Number} and MAX ITERATION =
{MAX_ITER}")

    return [num_test, correct_predict, class_based]

# Call the knn function to perform k-nearest neighbor classification
knn_classifier(train_histograms, test_histograms)
results_bowl = svm_classifier(train_histograms, test_histograms)

# Calculates the average accuracy and class based accuracies.
def accuracy(results):
    avg_accuracy = (results[1] / results[0]) * 100
    print("\nAverage accuracy: %" + str(avg_accuracy))
    print("\nClass based accuracies: \n")
    for key,value in results[2].items():
        acc = (value[0] / value[1]) * 100
        new = "{:.2f}".format(acc)
        print(f"Class {key} : {new} %")

# Calculates the accuracies and write the results to the console.
accuracy(results_bowl)

# Task Five
# CNN Model for AlexNet, VGGNet and ResNet
CNN_test = test
CNN_train = train

# Installing the base library we need
!pip install --upgrade fastai

# Import required libraries, and set the random seed to improve
repeatability

%matplotlib inline
from fastai.basics import *
from fastai.callback.all import *
```

```
from fastai.vision.all import *

set_seed(42)

print(path)

# Prepare the Data to be Loaded For the CNN Models. Training and
Testing Folder
def CNN_Prep(data, type):
    for i in range(10):
        new_path = path + 'CNN/' + f'{type}/'
        for j in range(int(len(data)/10)):
            img = train[(i * 400) + j]
            if img.shape != (20, 20):
                img = img.reshape(20, 20).astype(np.uint8)
            file_name = os.path.join(new_path+f"{i}/",
f"Training_{j}.jpeg")
            cv.imwrite(file_name, img)

CNN_Prep(CNN_train, 'Training')
CNN_Prep(CNN_test, 'Testing')

# Checks how many Files are in each Folder
def check_files(type):
    print(f"\n{type} Folder Detail")
    sum = 0
    for i in range(10):
        new_path = path + 'CNN/' + f'{type}/'
        file_size = os.listdir(new_path + f'{i}')
        print(f"Class {i} have {len(file_size)} Files")
        sum += len(file_size)
    print(f"\nTotal Number of File in the {type} Folder is {sum}\n")

check_files('Training')
check_files('Testing')

CNN_path = path + 'CNN/'

# Load the Data from the Folder and Separate into Train and Valid with
a Size of 224
data = ImageDataLoaders.from_folder(CNN_path, train="Training",
valid="Testing", bs=64, item_tfms=Resize(224))
```

```
# I also Tried Adding batch_tfms=[*aug_transforms()] Here. Results
Shown in the Report
# Show the Different Classes and Their Example
data.show_batch(figsize=(5,5))

# AlexNet

# Create Custom AlexNet Model
learn = vision_learner(data, alexnet, metrics=accuracy)
# Training Loop
learn.fit_one_cycle(4)
# Save the Model so we Can Restore It.
learn.save('stage-1')

# interp = ClassificationInterpretation.from_learner(learn)

# Can Show If Want to
# interp.plot_confusion_matrix()

# Fine Tuning the Model. Unfreeze the Rest of Network and Train the
Whole Data.
# We are only Training the Last Layer of the Network so Far.
learn.unfreeze()
learn.fit_one_cycle(1)

learn.load('stage-1')

# Here, the lr_find() Function will provide you with a Valley that is
giving you a range of Training Rate
# Where your Model is Most likely to be Training Effective.
learn.lr_find()

# Use the Suitable Training Rate to find Tune the Machine and Check for
the Accuracy
learn.unfreeze()
learn.fit_one_cycle(2, lr_max=slice(1e-5,1e-4))

interp = ClassificationInterpretation.from_learner(learn)

# Show Confusion Matrix
interp.plot_confusion_matrix()
```

```
# Show Pre/A/L/Pro to show a few of the One we predict wrong. Not all
are wrong. it bases on the Accuracy of the Train.
losses, idxs = interp.top_losses()
interp.plot_top_losses(9, figsize=(5,5))

# VGGNet

learn = vision_learner(data, vgg16_bn, metrics=accuracy)
learn.fit_one_cycle(4)
learn.save('stage-2')

# interp = ClassificationInterpretation.from_learner(learn)
# interp.plot_confusion_matrix()

learn.unfreeze()
learn.fit_one_cycle(1)

learn.load('stage-2')
learn.lr_find()

learn.unfreeze()
learn.fit_one_cycle(2, lr_max=slice(1e-6,1e-5))

interp = ClassificationInterpretation.from_learner(learn)
interp.plot_confusion_matrix()

losses, idxs = interp.top_losses()
interp.plot_top_losses(9, figsize=(5,5))

# ResNet

learn = vision_learner(data, resnet34, metrics=accuracy)  #use a
resnet34 instead of alexnet
learn.fit_one_cycle(4)
learn.save('stage-3')

# interp = ClassificationInterpretation.from_learner(learn)
# interp.plot_confusion_matrix()

learn.unfreeze()
learn.fit_one_cycle(1)

learn.load('stage-3')
```

```
learn.lr_find()

learn.unfreeze()
learn.fit_one_cycle(2, lr_max=slice(1e-5,1e-4))

interp = ClassificationInterpretation.from_learner(learn)
interp.plot_confusion_matrix()

losses, idxs = interp.top_losses()
interp.plot_top_losses(9, figsize=(5,5))
```

Task One: (Source Code Pages 3 and 4)

Extract the tiny images from the big image, split them into training images (80%) and testing images (20%), and write them as images (e.g. jpeg files) and store the training images in the “Train” folder and store the testing images in the “Test” folder. Basic Image Pre Processing

For this Task, I first import the Image File from the GoogleDrive and Show the Image and its shape. (This is to make sure that I have the right image imported and check the shape of the Image to have 50 Rows of 100 Digits, Adding up to 5000 20*20 Pixel Size Digits)

After checking that the Image was correctly imported and the digits were in 20*20 Pixels form, I then converted the image into a grayscale image (to enhance contrast and highlight features of interest) and split the image into cells of Digits.

For the Task, we split the image into 5000 cells. Then we Horizontally Split the Image into 100 Columns and Vertically Split into 50 Rows which then added up to 100*50 = 5000 Digits. (Each Column in the row is a unique digit image)

After extracting all Individual Tiny Pixels, we then have to prepare the 80% Training Sample and 20% Test Sample. We do this by taking the first 20 Digits (Out of 100) from each row as the 20% Test Sample while leaving the other 80 Digits as the Training Sample. This will then fairly distribute the Training and Test Samples by having the same amount of each digit within the Training and Testing Sample. (This is done in case we have 1000 Test Samples and 900 of them are Digit One and the other 100 are distributed among the other Nine Digits, making it a Bad Testing Sample.) Here, we reshape each image in (-1,400) to turn the image into a One Dimension Array with 400 lengths to store each of the 20*20=400 Pixel Images.

The next step is to loop through both the Training Sample and Testing Sample List to Store each image inside those files to the specific file that they should go to. Training Sample should be stored in the “Training” Folder and the Testing Sample should be stored in the “Testing” Folder. When processing each List, we first make sure that each image we put into the Folder is sized 20*20 (Making sure that the image is written in the correct size instead of been (-1,400), which is used for image processing) and then Written into the Folder. (We do The Lists one by one so we don't have to switch between Folders to Write the Digits, making it more efficient.)

Last, We Create Labels for both the Training and Testing Sample by repeatedly placing each digit's number of times into the Training_Label (400 for each Digit) and Testing_Labels (100 for each Digit).
All the Requirements of this Task were completed and it has efficiently distributed all the Tiny Digits into the “Training” and “Testing” Folder. And now Training Has 4000

Samples and Testing Has 1000 Samples. The Performance of this part of the program can be seen in the “Training” and “Testing” Folder.

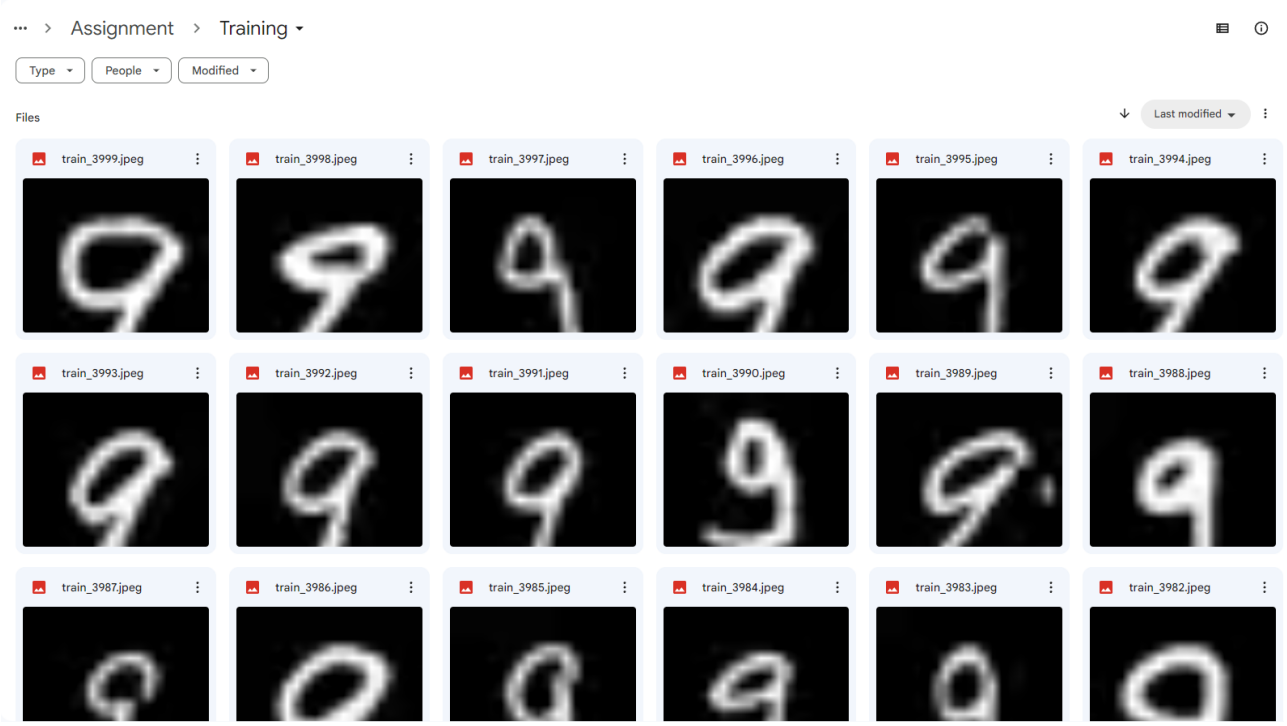
```
The Number of Testing Sample is 1000.

The Number of Training Sample is 4000.

All Extracted Image arranged into "Training" and "Testing" Folders
```

Reference Used for this Task was taken from:

Prac 5 Worksheet In COMP3007 Machine Perception. Prac 05 Exercise 3

This Task is done so that the next Two Tasks (KNN and OvR-SVM) can be done. We need to have the Data Split into Training and Testing Data in order for them to work.

.



Task Two: (Source Code Pages 5 and 6)

Nearest Neighbor Method For Image Classification

For this Task, We are applying different K Values to the K Nearest Neighbour Method and outputting their Confusion Matrix. The first thing I did was to prepare a list of Five K Values to be tested and Initialise ‘accuracies’ and ‘confusion_matrices’ lists to store the result of different K Values. Then Creating and Training the KNN Classifier for Further Processes Since the Creation and Training of KNN would not have to be repeated for Each K Value (Done once is enough)

I then created a Loop to repeat the Prediction on the Test Data for each K Values there are. Here, the K Value refers to the Number of Nearest Neighbour that We are Considering when Predicting the Test Data. This is done by calculating the Majority Class within the K Nearest Neighbours that it is assigned to. E.g. If K=3 and Two of the Nearest Points belong to Class 0 and One Point belongs to Class 1, then we will Classify the New Point to be CLass 0 since It has the highest majority within the nearest neighbours.

Then, we check the Prediction Results with the Actual Results (Which are the test_labels that we have created), Calculate the Accuracy and then Store the Result in the ‘accuracies’ and ‘confusion_matrices’ List for later plotting the Confusion Matrix. Below is an example of what the ‘cm’ stored in the ‘confusion_matrices’ looks like:

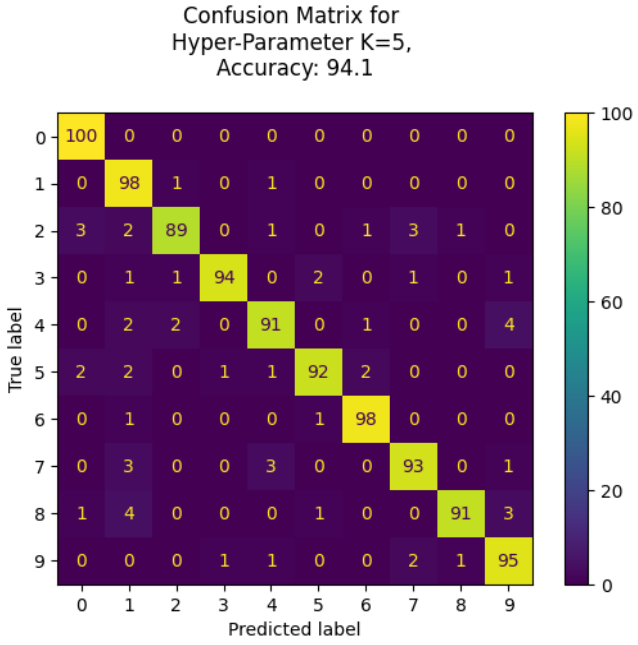
The Below Image is an example of what the cm store in the ‘confusion_matrices’ List looks like. For this example, we have the K been Six and we can see that there are Ten Rows and Ten Columns Indicating the Labels. E.g. The First Row is for Predicting Digit Zero and the result shown by this ‘cm’ shows that out of the 100 Digit Zeros, 100 of them are Predicted as Zero and none for others, meanwhile for Digit One, there are only 98 of them been Predicted as One, while the other two have been Predicted as Two and Four.

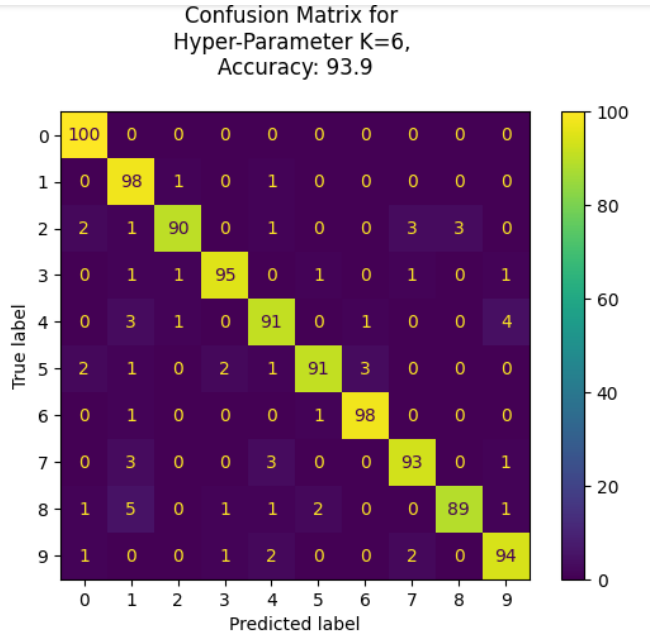
```
[[100   0   0   0   0   0   0   0   0   0]
 [  0  98   1   0   1   0   0   0   0   0]
 [  2   1  90   0   1   0   0   3   3   0]
 [  0   1   1  95   0   1   0   1   0   1]
 [  0   3   1   0  91   0   1   0   0   4]
 [  2   1   0   2   1  91   3   0   0   0]
 [  0   1   0   0   0   1  98   0   0   0]
 [  0   3   0   0   3   0   0  93   0   1]
 [  1   5   0   1   1   2   0   0  89   1]
 [  1   0   0   1   2   0   0   2   0  94]]
```

After Storing all the Results for different K Values, We just have to Plot them into the Confusion Matrix and Display them.

Lastly, we want to quickly display the Accuracy of Each Unique Class Label for the K Value with the Highest Accuracy so we can later use it to Compare with the SVM OvR.

Example of Confusion Matrix for K = 6 and K = 5:



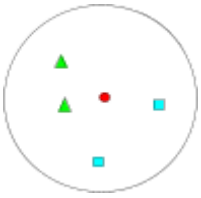


Result:

```
Show all the Accuracy of different K_value:

The Result For k = 1 is 93.7%
The Result For k = 3 is 93.9%
The Result For k = 5 is 94.1%
The Result For k = 7 is 94.0%
The Result For k = 9 is 93.8%

Show all the Accuracy of different Class Labels in for the Highest K_Value:

The Result For Class 0 is 100%
The Result For Class 1 is 98%
The Result For Class 2 is 88%
The Result For Class 3 is 93%
The Result For Class 4 is 91%
The Result For Class 5 is 92%
The Result For Class 6 is 99%
The Result For Class 7 is 96%
The Result For Class 8 is 89%
The Result For Class 9 is 92%
```

There is Something Important When considering using the KNN Method. We need to Remember that for K Vlaue, We should not Assign it as Even Value.

So Situation like the Below Image that I Draw would not happen, when the Rectangle and the Square have to same Amount, and the Class of the Circle Cannot be Identified because of that.



Another Way that we can Possibly do the KNN Method is to use the "KNeghborsClassifier" Functions from sklearn.neighbors. Shown in the Source Code. Commented Out at The End Of Task Two.

Reference Used for this Task was taken from:

Prac 5 Worksheet In COMP3007 Machine Perception. Prac 05 Exercise 3

https://www.w3schools.com/python/python_ml_confusion_matrix.asp

https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

Task Three: (Source Code Pages 6, 7 and 8)

Linear Classifier for Image Classification (SVM and Compare with KNN)

In this Task, we are to Compare the performance Of SVM OvR with the result we got from the Previous Task (KNN). To do this, We first need to Process the Support Vector Machine with the One VS Rest Strategy, which compares each Class (Label/Digit) with ALL the other classes (As One).

First, We Create an ‘accuracies’ List to Store the Accuracy for Each Unique OvR Class. We then create the Support Vector Machine. Here, the number we put for the MAX_ITER would change the result, so we can think of It as being similar to K Value but also Different. Max Iteration is like the Number of Iterations that the Algorithm will Attempt to Find the Best Solution (margin) and Lower the Errors. But a Too High Number will increase Training Time.

```
svm = SVC(kernel='linear', max_iter=4000, tol=1e-8, C=1)
```

After Creating the SVM, the Next step is to Computer the SVM with the One VS Rest Method. To do This, we will have to Create a OneVsRestClassifier to Apply the OvR Method and then Train the Classifier with the ‘.fit()’ function with the Train Data. The Following Idea for the OneVsRestClassifier Method is Taken from the Website: https://scikit-learn.org/stable/modules/generated/sklearn.multiclass.OneVsRestClassifier.html#sklearn.multiclass.OneVsRestClassifier.fit

Then the next Steps are similar to the KNN Method. We Calculate the Accuracy and for the further Plot Process.

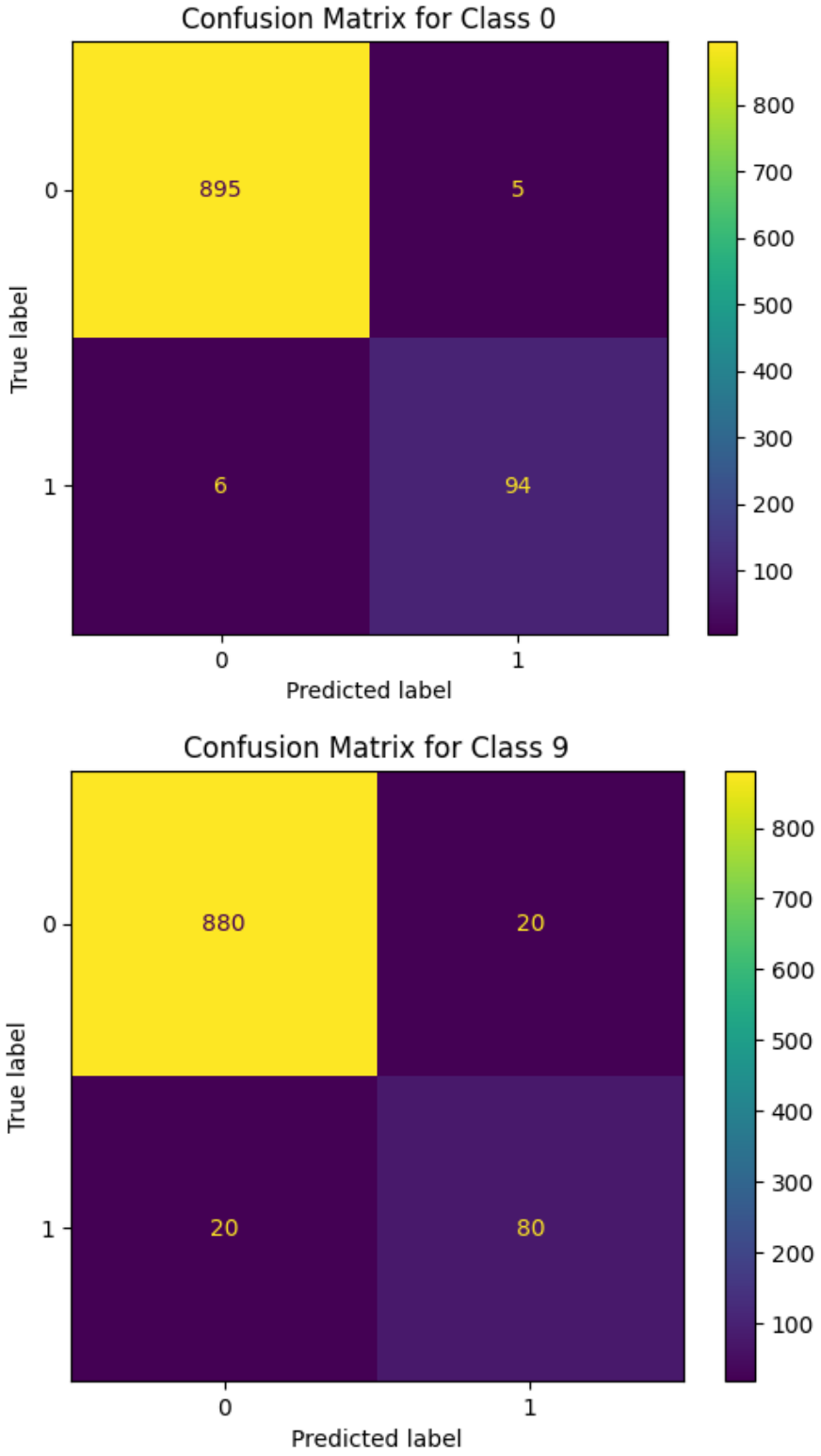
Last, We do the same thing as the KNN Method to Plot the Matrix, But this time Instead of showing the comparison of all Ten Classes, For each Confusion Matrix (Unique Label Class vs. All Others), we only see Two * Two Matrices. Where is it One Matrix is the Unique Class and the Other been the Rest of the Classes.

```
     -1  1
-1 [[880 20]
1 [ 20 80]]
```

Example Of The ‘cm’ for Class 9

I Label the Example with 1 and -1 So that we can tell what it is showing. The (-1, -1) is the Success Predict for Digits that is not Nine. (1, 1) is the Success Predict for Digit that is Nine. While (1, -1) and (-1, 1) are the Errors.

Example of Confusion Matrix for Class 0 and Class 9 With MAX_ITER = 1000:



As we can See here. The Label One is for the Unique Class Label while the Label Zero is for all the Rest of the Classes.

Now Let's Start on The Comparison Between KNN and SVM OvR:

```
# KNN -

# The Result For k = 1 is 93.7%
# The Result For k = 2 is 92.4%
# The Result For k = 3 is 93.9%
# The Result For k = 4 is 93.8%
# The Result For k = 5 is 94.1%
# The Result For k = 6 is 93.9%

# For the Highest K_value 5. The Individual Class Label Accuracy is:

# The Result For Class 0 is 100%
# The Result For Class 1 is 98%
# The Result For Class 2 is 90%
# The Result For Class 3 is 95%
# The Result For Class 4 is 91%
# The Result For Class 5 is 91%
# The Result For Class 6 is 98%
# The Result For Class 7 is 93%
# The Result For Class 8 is 89%
# The Result For Class 9 is 94%

# OvR-SVM

# For this, We Use the MAX_ITER with the Highest Overall Accuracy, and these are the few Highest:

# This Changes based on differenet MAX_ITER For 100: 63.7%
# This Changes based on differenet MAX_ITER For 2000: 82.40%
# This Changes based on differenet MAX_ITER For 4000: 84.30%

# So We show the Individual Class Label Accuracy for MAX_ITER 4000 since it has the Highest.

# The Result For Class 0 is 94%
# The Result For Class 1 is 97%
# The Result For Class 2 is 79%
# The Result For Class 3 is 80%
# The Result For Class 4 is 84%
# The Result For Class 5 is 68%
# The Result For Class 6 is 98%
# The Result For Class 7 is 84%
# The Result For Class 8 is 79%
# The Result For Class 9 is 80%

# Comparing Between KNN and OvR_SVM, KNN is the Better choice
```

Reference Used for this Task was taken from:

Prac 5 Worksheet In COMP3007 Machine Perception. Prac 05 Exercise 3

https://www.w3schools.com/python/python_ml_confusion_matrix.asp

https://scikit-learn.org/stable/modules/generated/sklearn.multiclass.OneVsRestClassifier.html#sklearn.multiclass.OneVsRestClassifier.fit

Task Four: *(Source Code Pages 8 to 14)*

Image classification using Bag of Visual Words

In this Task, We will be splitting the Task into Five Different Parts

<u>One - Extract Local Features from Images using the SIFT Method.</u>

The first step to do this, Is to create two Dictionaries to store the Training and Testing data in their Unique Class. Here, we are changing the Images back to the 20*20 Pixel Image Rather than Continuing to Use the 1D Array with 400 Values Since SIFT processes the Original 2D Image Rather than the Plattened 1D Image.

Then We Progress to the SIFT Method, where we create the SIFT Detector, Define the ‘descriptor_list’ List to store all the SIFT Descriptors from all Images and also a ‘sift_vector’ Dictionary to store the SIFT Vectors by Class.

To Apply the SIFT Method, we will have to loop through each and every Unique Class digit and extract each image in that class to Compute and Detect the Features of that given image. This is done by the ‘detectAndCompute’ Function which is used for Keypoint Detection and Feature Extraction. As long as the Keypoint is None, we add the features of that image to both the ‘descriptor_list’ and also their unique ‘sift_vector’.

Here, I have only Implemented the SIFT Function Rather than Having HOG As well, It is because SIFT is Better at Scale Invariant, Rotation Invariance etc. And HOG is Also more Complex to Implement And Requires more CPU and GPU Power to process.

<u>Two - KMean Clustering to Find the Center Points.</u>

Here, we simply use the KMean’s Function to Compute the Cluster Centroids (Centers). Here, we have to Consider the use of the KMean Number. The Number is Set to Group the Data into that Number of Clusters and Having the Right Number of Cluster will help Group the Data into Meaningful Clusters. But here, Bigger KMeans also mean that it will Take Longer to Process and Might not Give the Best Compared to Lower Ones. (Speed Efficiency and Accuracy Should be Considered)

Another Important Thing! KMean Number Should not be More Than The Amount Of Data Points!!!! So here I set the KMean to 2500 after a few Attempts to find the result with the highest Accuracy.

```
k_mean_Number = 2000

# HERE!!! Visual Word Is Equivalent to a Cluster Centroid
visual_words = kmeans(k_mean_Number, descriptor_list)
```

Some Examples:
1. For K Mean = 500 and MAX ITER = 4000 Accuracy: %72 .59 Time: 2 Mins
2. For K Mean = 1000 and MAX ITER = 4000 Accuracy: %75.42 Time: 6 Mins
3. For K Mean = 2000 and MAX ITER = 4000 Accuracy: %75.87 Time: 8 Mins
4. For K Mean = 2500 and MAX ITER = 4000 Accuracy: %78.03 Time: 9 Min
5. For K Mean = 2500 and MAX ITER = 4000 Accuracy: %78.03 Time: 9 Min
6. For K Mean = 2500 and MAX ITER = 4000 Accuracy: %76.67 Time: 12 Min

Three - Compare Extracted Local Features with the Visual Words to Create Histograms. Both for the Training and Testing Data

In This Part of the Code, We go Through Each Image to Create its Histogram representing the amount of SIFT Features from that Image that Belongs to Each Visual Word. The Idea was taken from the Website Provided.

We Input the Dictionary that contains the SIFT Vectors for Different Classes of Images. Each Image has its related List of SIFT Features. Then Creating a Dictionary to Store a Histogram of Images from Different Classes. In each Class (Loop), We Loop From Each Image in that Class and Process the Images one at a time. For each image, Initialise A List of Zeros (Size is the Number of Visual Words) and then Loop into that Image over Each SIFT Feature for that Image and Compute the Closest Cluster Center for the Current SIFT Feature.

If the Cluster is Determined, the Index of the Histogram Increases by one, to count how many SIFT Features belong to each Visual Word and after all SIFT Features of that Image have been Processed, they will be added to the Category List which Contains all the Histogram of the Current Class Image.

Four - Predict the Test Data and Compare them with Each Histogram of the Train Image. 1-NN Will be Used.

Now, based on the Histograms that We have Created for both Training and Testing Data, we pass them both into one of these two Methods: KNN or SVM. Inside those Functions, We First Initialise the Variables to Count the Total Number of Test Images, another to Count the Number of Correct predictions and a Dictionary to Store the Result for each Unique Class. Then There are Two Choices:

SVM -

We Create the SVM Classifier and Give it a Max Iter Number. MAX ITER here is Also Very Important! It sets the Maximum Number of Iterations the Algorithm Updates its Model, Which is used to find the Best Decision Boundary to Separate the Data Points. Then we Prepare the Training Data and Labels and Train the SVM Machine.

After Training the Machine, We would Loop Through Each Class and Each Image with that Class to Predict the Class Label for that Test Image Using Its Histogram. Then the Prediction is compared with the Actual Label to Determine if it is a Correct Prediction or not and the Correct Number in the Unique Class Dictionary. And then Returning the Result.

KNN-

Unlike in the SVM Method, We do not Have to Train the Data (K Nearest Neighbour Special). We can straight up and Loop through the Unique Classes and Each Image within that Class to compare that Image with all Training Images from Different Classes and Calculating Measure the Euclidean Distance to Check the Similarity Between the Test Image and Each Training Image. This will Continue through the Training Data and Continue to Find and Update the Predicted Label Based the the Euclidean Distance. Then End up Returning the Result Label of the Closest Training Image. After that, it does the same as SVM to Check with the Correct Label and Return the Result of Correct Predictions.

Five - Calculating the Result

Last, all we have to do is use the Results that we have computed from the Above Methods to Calculate the Overall Accuracy and the Accuracy for Each Unique Class and Display the Result. E.G.

```
For K Mean = 2500 and MAX ITERATION = 4000

Average accuracy: %77.01019252548132

Class based accuracies:
```

```
Class 0 : 87.37 %
Class 1 : 93.22 %
Class 2 : 69.89 %
Class 3 : 72.92 %
Class 4 : 73.68 %
Class 5 : 63.86 %
Class 6 : 73.91 %
Class 7 : 71.95 %
Class 8 : 86.60 %
Class 9 : 80.22 %
```

BoVW Results:

```
# A Few Comparasion:
# 1. For K Mean = 2000 and MAX ITERATION = 4000 Average accuracy: %74.63
# 2. For K Mean = 2250 and MAX ITERATION = 4000 Average accuracy: %75.67
# 3. For K Mean = 2500 and MAX ITERATION = 4000 Average accuracy: %78.03
# 4. For K Mean = 2750 and MAX ITERATION = 4000 Average accuracy: %77.92
# 5. For K Mean = 3000 and MAX ITERATION = 4000 Average accuracy: %76.89


# Best Choice:

# For K Mean = 2500 and MAX ITERATION = 4000

# Average accuracy: %78.02944507361268

# Class based accuracies:

# Class 0 : 89.47 %
# Class 1 : 93.22 %
# Class 2 : 69.89 %
# Class 3 : 77.08 %
# Class 4 : 80.00 %
# Class 5 : 69.88 %
# Class 6 : 72.83 %
# Class 7 : 60.98 %
# Class 8 : 86.60 %
# Class 9 : 82.42 %
```

Compare to KNN Results:

```
# KNN -

# The Result For k = 1 is 93.7%
# The Result For k = 2 is 92.4%
# The Result For k = 3 is 93.9%
# The Result For k = 4 is 93.8%
# The Result For k = 5 is 94.1%
# The Result For k = 6 is 93.9%

# For the Highest K_value 5. The Individual Class Label Accuracy is:

# The Result For Class 0 is 100%
# The Result For Class 1 is 98%
# The Result For Class 2 is 90%
# The Result For Class 3 is 95%
# The Result For Class 4 is 91%
# The Result For Class 5 is 91%
# The Result For Class 6 is 98%
# The Result For Class 7 is 93%
# The Result For Class 8 is 89%
# The Result For Class 9 is 94%
```

Compare This to the Result we received from the Linear Classification. It Shows that the Linear Classification Actually works better than the BoVW Method in this Scenario and is Also more Efficient, Needing Less Computation Power and Resources.

Some Reasons Why Linear Classification Might be Better than BoVW:

- The Images have Easily Compatible Features.
- Fewer Steps and Reduce of Complexity. (Calculations and Computational Power Required)
- Linear Classification Have Fewer Hyper Parameters to Use Compared to BoVW.
- If the SIFT Method Does not really Capture the Useful Descriptors, that Might affect the Performance of the BoVW as well as Compare to Using Linear Classification.

Reference Used for this Task was taken from:

https://medium.com/@aybukeyalcinerr/bag-of-visual-words-bovw-db9500331b2f

https://scikit-learn.org/stable/modules/svm.html

https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

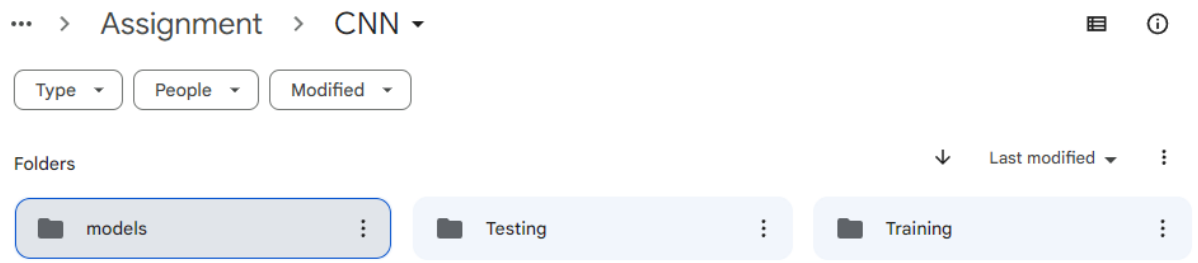
Task Five: *(Source Code Pages 14 to 18)*

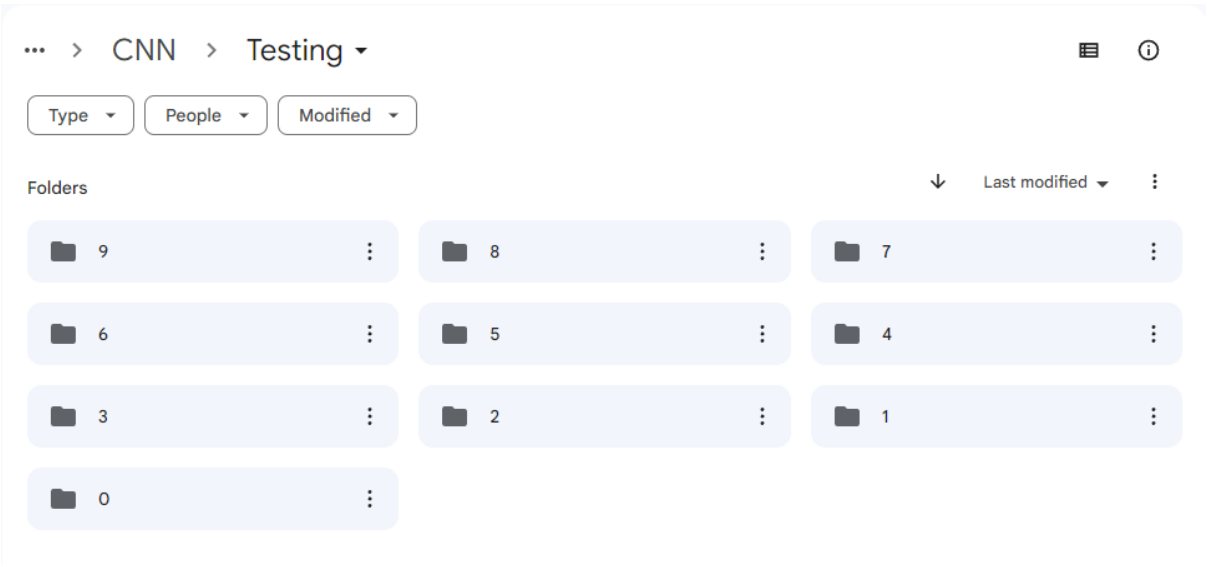
CNN Models (AlexNet, VGGNet, ResNet) Dealing with Digits

In this Task, We will be using all Three Different CNN Models and at the End Compare the Performance of the Three CNN Models.

The First step for This Task is to First Load the Required Library which is the ‘fastai’ Library. This Library provides users with Pre pre-trained models to use. After we Install and Imported the required packages from ‘fastai’, We then had to Prepare the Data to be Loaded for the CNN Models and the ImageDataLoaders’s from_folder() Functions, We will need to have both the Training and Testing Data in Two Different Folders Within a File, While having Listed Folder of Unique Classes with Their Matching Images within that Folder.

E.g





This is Where we Arrange the Data into the Correct Classes For the CNN Models.

After That, then Load the Data from the Folder and Resize them to 224 which is used for Most CNN Model's Learning.

Now After We Finish Setting Up the Data. We Now Start to Train and Test the Data with Different CNN Models.

These Steps are the Same for the Three Different Models other Than Changing the name of the Model when we Create the Custom Model.

First, We have to Create our Custom Model for Our CNN. Which is This:

```
# Create a Custom AlexNet Model
learn = vision_learner(data, alexnet, metrics=accuracy)
```

In the Above Code, We are Creating the Custom Model for AlexNet, But If we want to Create the Custom Model for VGGNet or ResNet, This is what we do:

Below Code for VGGNet

```
# Create a Custom VGGNet Model
learn = vision_learner(data, vgg16_bn, metrics=accuracy)
```

Below Code for ResNet

```
# Create a Custom ResNet Model
learn = vision_learner(data, resnet34, metrics=accuracy)
```

This Step is The Set up part to Track Accuracy as Metrics.

Second, We Train the Model and Specify the Number of Cycles (epoch) We want to Train the Model. In this Part, the Model presents the Training Data and its Parameters based on the Loss Computed.

```
learn.fit_one_cycle(4)
```

Third, We Save the Result from the Train to be used for Making Predictions for New Data.

```
learn.save('stage-1')
```

And These Are All The Steps for The CNN Model. These are the Results:

AlexNet:

epoch	train_loss	valid_loss	accuracy	time
0	1.584413	0.300366	0.926000	00:20
1	0.664772	0.129375	0.972000	00:22
2	0.376153	0.096387	0.981000	00:20
3	0.264301	0.088879	0.982000	00:23

VGGNet:

epoch	train_loss	valid_loss	accuracy	time
0	1.514413	0.246910	0.919000	00:56
1	0.586885	0.069745	0.981000	00:55
2	0.278637	0.035214	0.992000	00:57
3	0.159886	0.031286	0.992000	00:56

ResNet:

epoch	train_loss	valid_loss	accuracy	time
0	1.431928	0.347272	0.880000	24:41
1	0.499636	0.056258	0.980000	00:23
2	0.219086	0.026570	0.992000	00:26
3	0.119299	0.020295	0.995000	00:24

We can see from the Start without any Fine-Tuning, ResNet > VGGNet > AlexNet.

Then We Apply Some Fine-Tuning To These Results and See How It Goes. These Are the Steps:

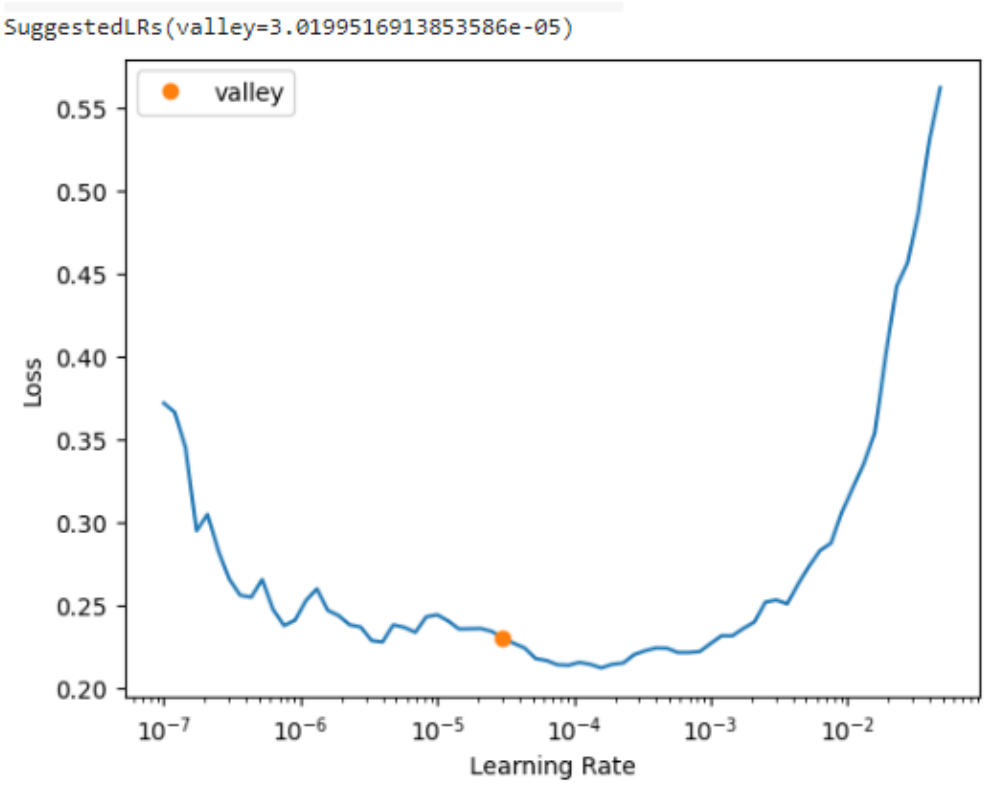
First, We use Allow Training of the Models to Train For the Previous Layers of the Pre-Trained Neural Network. By the use of the 'unfreeze()' Function, We Unfreeze the Rest of the Network Train the Whole Data and Cycle Through it Once to see How it Goes. Since We were only Training the Last Layer of the Network Before.

```
learn.unfreeze()
learn.fit_one_cycle(1)
```

Second, We then Want to Load the Previous Saved Result To Create A Table to Show the Best Learning Rate for The Model For Further Fine-Tuning. This is done by the Function Called 'lr_find()':

```
learn.load('stage-1')
learn.lr_find()
```

Below Is An Example of What the Valley Plot will Look Like:



As We can see From the Above Image, The Best Learning Rate for This Model (AlexNet) Would be Around the 10^{-5} to the 10^{-4}. This means That We Would Want to Test Our Model Within That Most Efficient Learning Rate in the Next Step.

Third, the Last Step Of This Fine-Tuning Would Be Using the Effective Learning Rate that We have Concluded Above, We want to Use the Function:

```
learn.fit_one_cycle(2, lr_max=slice(1e-5,1e-4))
```

To Cycle Through The Training with a Valley That is Giving a Range of Effective Training Rates, Where the Model is Most Likely to be Training Efficient. Then We Can Interpret Those Results and Display Things Such as a Confusion Matrix or Top top-loss images.

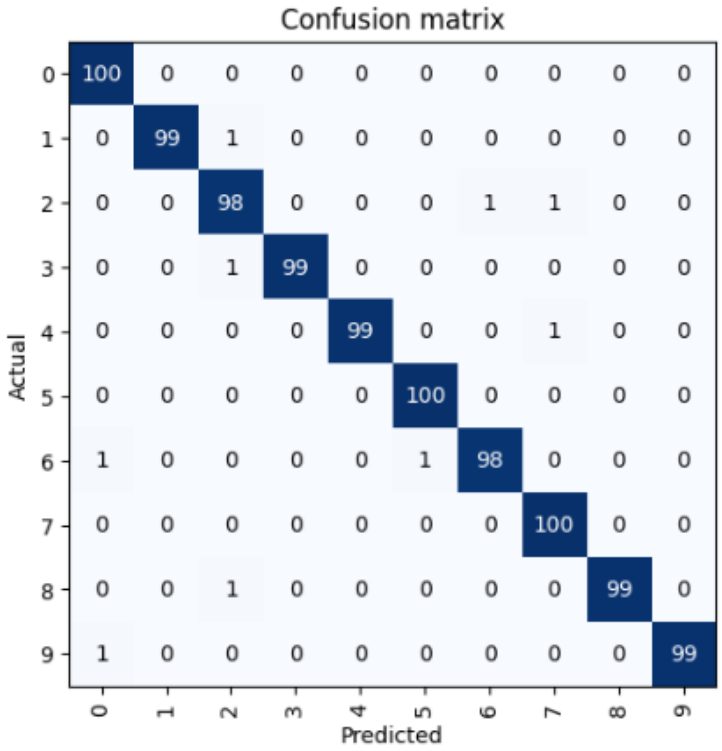
As for The AlexNet From Above:

epoch	train_loss	valid_loss	accuracy	time
0	0.197029	0.053016	0.986000	06:30
1	0.160118	0.042699	0.991000	06:27

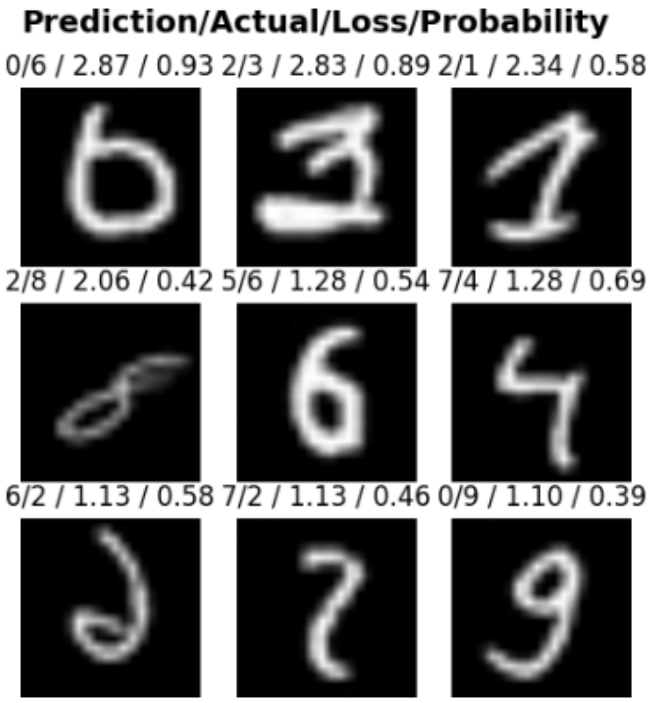
← Accuracy After Fine-Tuning

Some Examples Of Result Display for The AlexNet:

Confusion Matrix -



Top Lose Diagram -

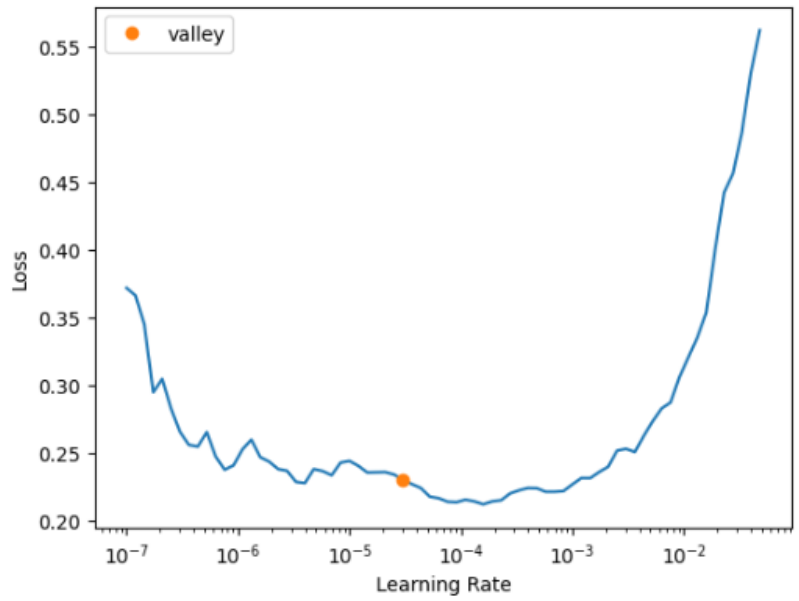


These are the Results For the After Fine-Tuning:

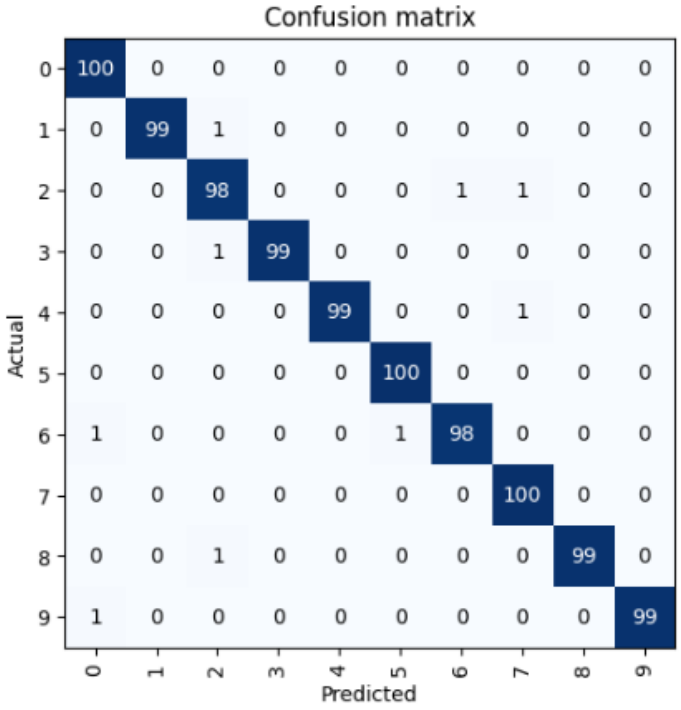
AlexNet - Highest Accuracy 99.1%

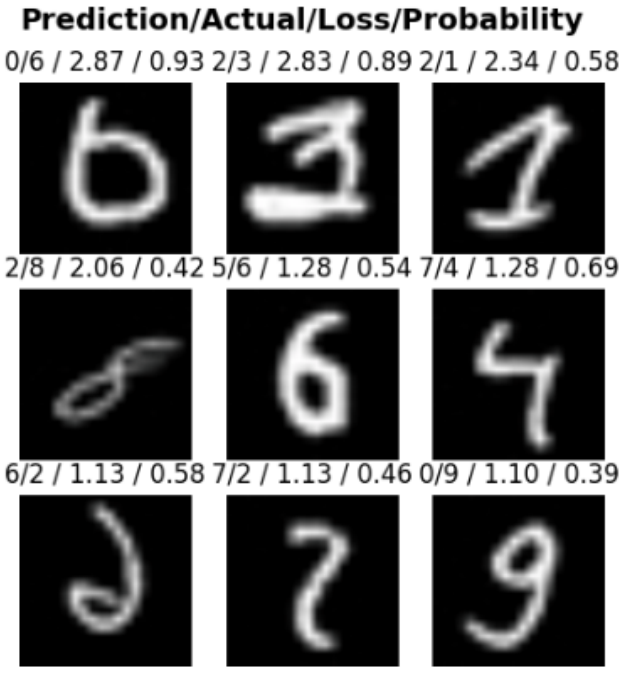
epoch	train_loss	valid_loss	accuracy	time
0	0.259779	0.096864	0.974000	06:32

SuggestedLRs(valley=3.0199516913853586e-05)



epoch	train_loss	valid_loss	accuracy	time
0	0.197029	0.053016	0.986000	06:30
1	0.160118	0.042699	0.991000	06:27

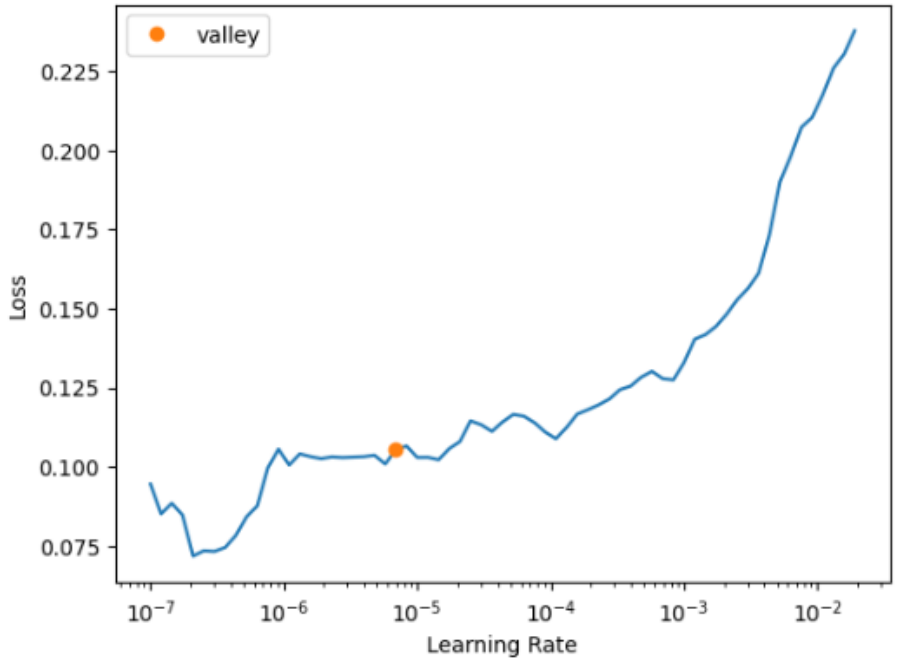




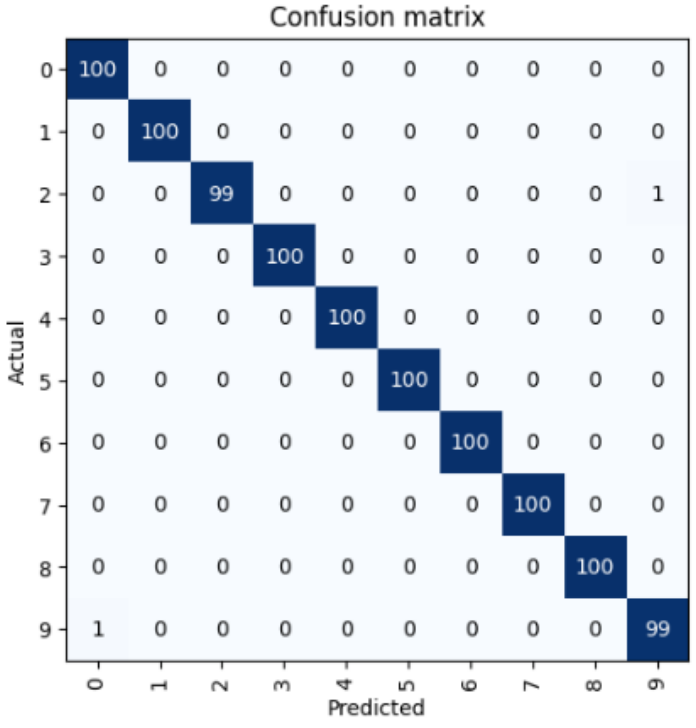
VGGNet - Highest Accuracy 99.8%

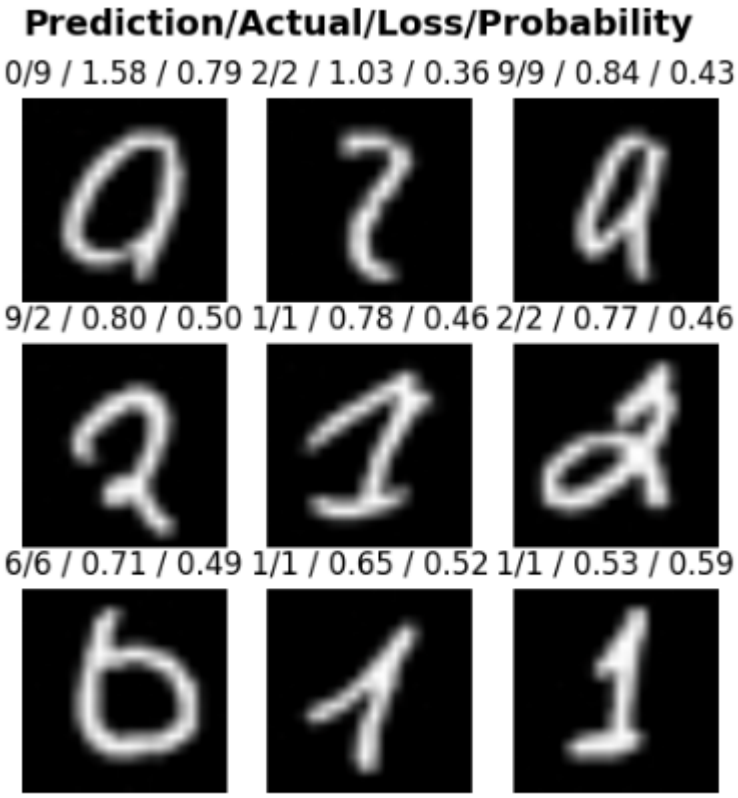
epoch	train_loss	valid_loss	accuracy	time
0	0.226885	0.056779	0.986000	01:13

SuggestedLRs(valley=6.918309736647643e-06)



epoch	train_loss	valid_loss	accuracy	time
0	0.106341	0.023134	0.997000	01:13
1	0.094778	0.020785	0.998000	01:13

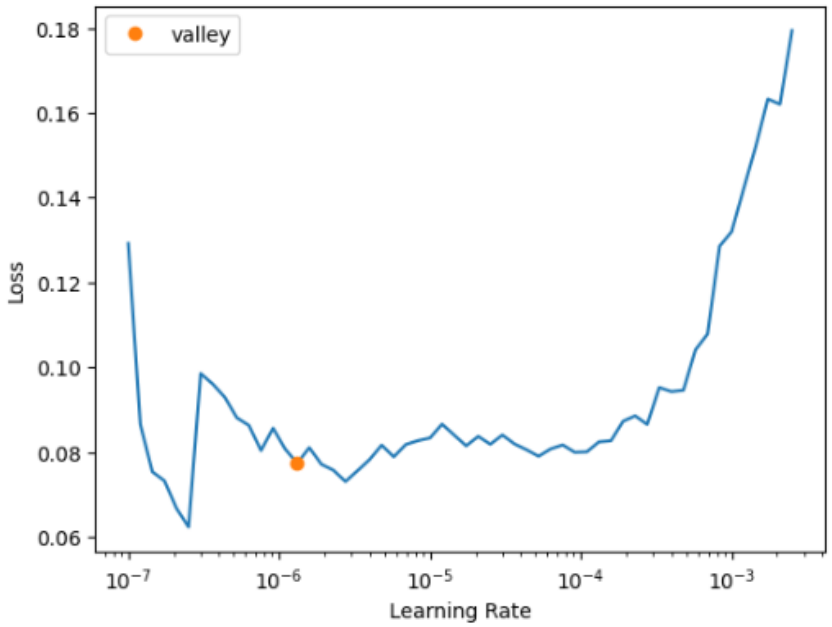



Prediction/Actual/Loss/Probability
0/9 / 1.58 / 0.79
2/2 / 1.03 / 0.36
9/9 / 0.84 / 0.43
9/2 / 0.80 / 0.50
1/1 / 0.78 / 0.46
2/2 / 0.77 / 0.46
6/6 / 0.71 / 0.49
1/1 / 0.65 / 0.52
1/1 / 0.53 / 0.59

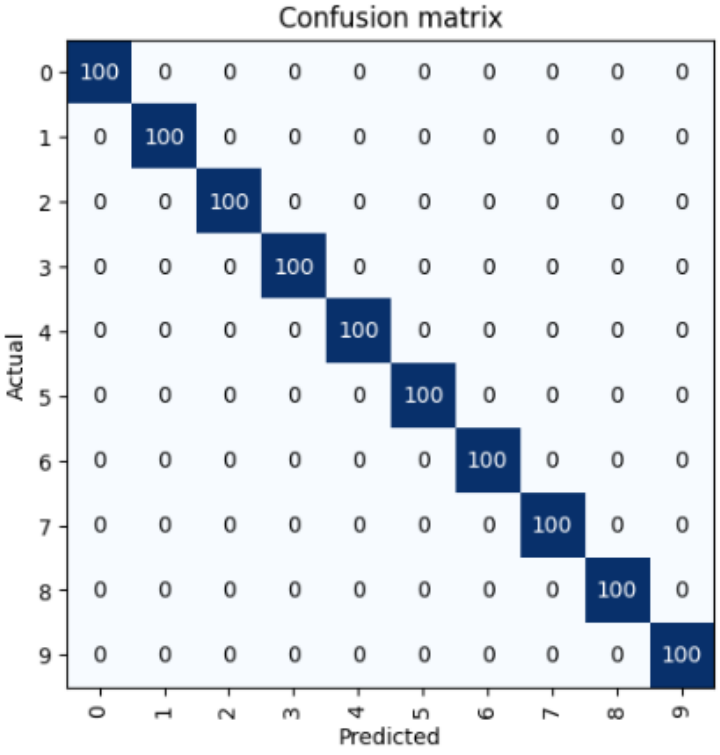
ResNet - Highest Accuracy 100%

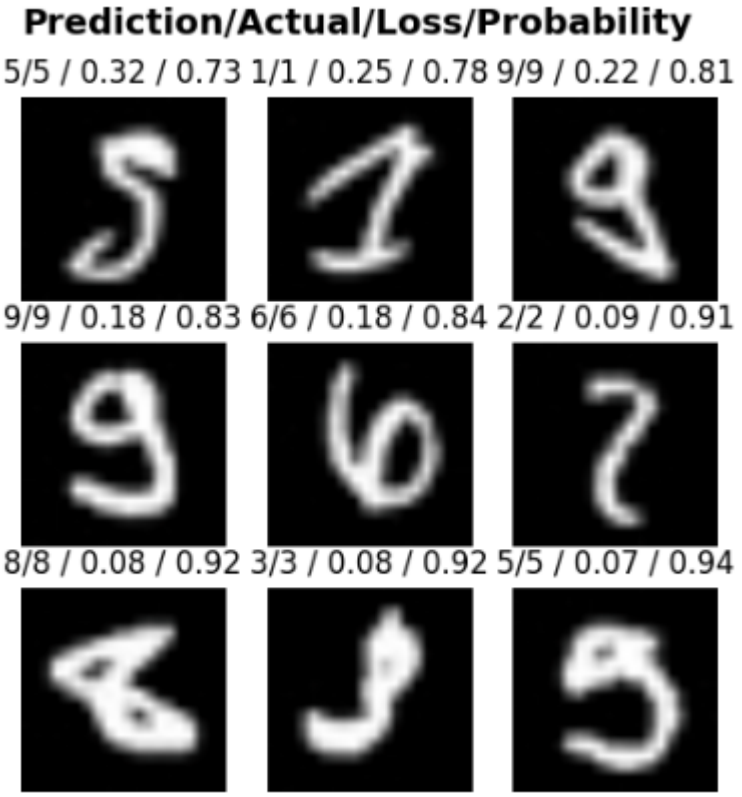
epoch	train_loss	valid_loss	accuracy	time
0	0.202484	0.061718	0.983000	00:28

SuggestedLRs(valley=1.3182567499825382e-06)



epoch	train_loss	valid_loss	accuracy	time
0	0.030228	0.003172	1.000000	00:35
1	0.019606	0.002736	1.000000	00:35




Prediction/Actual/Loss/Probability
5/5 / 0.32 / 0.73
1/1 / 0.25 / 0.78
9/9 / 0.22 / 0.81
9/9 / 0.18 / 0.83
6/6 / 0.18 / 0.84
2/2 / 0.09 / 0.91
8/8 / 0.08 / 0.92
3/3 / 0.08 / 0.92
5/5 / 0.07 / 0.94

Comparison of the Three CNN Models:

Based on the Results Shown, ResNet has the Best Performing Accuracy Both Before Fine-Tuning and After Fine-Tuning, Indicating that it has the Best Performance among the Three CNN Models. ResNet Have a Lower Complexity Than VGGNet and more layers compared to Both VGGNet and AlexNet, and it also has the ability to skip connections to overcome diminishing gradient problem. Making it the Most Reliable CNN Model among the Three.

Then Compared to AlexNet, VGGNet is to be the Better one. It is ‘Deeper’ than the AlexNet, Having more Layers. VGGNet Also has a Small Filter Size Kernel to Help the Convolutional Filter, and the Complexity is easier than AlexNet.

Reason for the Valley Taken During Fine-Tuning:

In our Training, The Learning Rate is a very Important Part of the Training. A Small Learning Rate might lead to the Model’s Weight being updated Very Poorly and Slowly, Meaning that the Training Process will take a Long Time to reach an Acceptable Performance. On the Other Hand, Having a High Learning Rate also affects the Training of the Data Due to the Model’s Weight being updated too Frequently, Causing the Training Process to become unstable and increasing the Error and Loss Rate.

Reference Used for this Task was taken from:

Lecture 6 - Neural Networks in COMP3007 Machine Perception

Prac 5 Worksheet In COMP3007 Machine Perception. Prac 05 Exercise 3

Some Add-on Explanation (KNN, KMean, SVM, CNN)

KNN - K Nearest Neighbour Classification

- KNN Choose The Majority Votes of K's Nearest Neighbours Found in the Training Sample.
- Key Parameters: Neighbourhood Size K (Odd Number)
- KNN is Easy To Train and Does not need Complex Mathematical Equations to Process.
- The K Value For the KNN Method is to Define the Number Of Nearest Neighbours that It is Going to Decide From.

SVM - Support Vector Machine

- SVM is to Find Decision Boundaries that Maximise the Margin Between Difference Classes.
- SVM Uses Various Mathematical Functions to Help Distribute the Data.
- In SVM, Max Iteration is the Number of Max Attempt the Algorithm Will goes on trying to find the Best Solution. After that Number, the Algorithm will Provide the Current Best Solution even if it is Not the Best.
- The Idea of Under-Fitting and Over-Fitting comes in when handling with the Max Iteration Number. Under-Fitting is when the Max Iteration is too Low, which means that there are not enough Patterns gathered by the Machine and would not provide the best Accurate prediction. Over-fitting on the other hand means when the Maximum Iteration is too High, Causing the Training Time to Increase by a lot.

KMeans - Clustering Method

- A Method Used to Distribute Data Into Groups or Clusters Based on Their Similarity. Partitions N Data Samples into K groups.

- It First Initializes K (Random or Set) Cluster Centroids, Then For Each Data Sample, Calculates its Distance to all Centroids. Assign the Data point to the cluster with the Minimum Distance. Lastly, try other Cluster Centroids and pick the one with the Best Outcome.

- In SVM, Max Iteration is the Number of Max Attempt the Algorithm Will goes on trying to find the Best Solution. After that Number, the Algorithm will Provide the Current Best Solution even if it is Not the Best.

- The Idea of Under-Fitting and Over-Fitting comes in when handling with the Max Iteration Number. Under-Fitting is when the Max Iteration is too Low, which means that there are not enough Patterns gathered by the Machine and would not provide the best Accurate prediction. Over-fitting on the other hand means when the Maximum Iteration is too High, Causing the Training Time to Increase by a lot.

References in APA 7th Format.

Python Machine Learning - Confusion Matrix. (n.d.). Www.w3schools.com. https://www.w3schools.com/python/python_ml_confusion_matrix.asp

scikit learn. (2018). *1.4. Support Vector Machines — scikit-learn 0.20.3 documentation*. Scikit-Learn.org. https://scikit-learn.org/stable/modules/svm.html

scikit-learn developers. (2019). *sklearn.neighbors.KNeighborsClassifier — scikit-learn 0.22.1 documentation*. Scikit-Learn.org. https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

sklearn.multiclass.OneVsRestClassifier. (n.d.). Scikit-Learn. Retrieved October 11, 2023, from https://scikit-learn.org/stable/modules/generated/sklearn.multiclass.OneVsRestClassifier.html#sklearn.multiclass.OneVsRestClassifier.fit

Yalçıner, A. (2019, July 12). *Bag of Visual Words(BoVW)*. Medium. https://medium.com/@aybukeyalcinerr/bag-of-visual-words-bovw-db9500331b2f

Other Reference:

COMP3007 Machine Perception Lecture and Prac Worksheets