ASSIGNMENT 1B

CAB420, Machine Learning

Ash Phillips, N10477659
Date: 10/05/2022

Problem 1: Training and Adapting Deep Networks

Discussion of Design Choices and Model Training

Network Design

To build each DCNN evaluated on the training set several filters and layers were used to ensure peak performance of the classification models. Once initialised, these layers were then repeated multiple times, flattened into a signal vector, and dense layers were added to model the classification result. Explanation of each layer can be seen below:

Six convolution layers, stacked in pairs, were setup to help filter the data to search for items of interest. These filters started at 8, then, as the layers repeated, they were increased to 16 and then 32. The 8 filters were used to learn initial features and detect the edges of the numbers. The 16 filters were used to combines those vector features to then detect further information about the numbers, e.g., the shapes of the numbers. The 32 filters then looked at the determined combination vectors to detect only detailed and required information.

Batch normalisation was used to standardise the magnitude of all data so the following layers can be completed on data that is formatted to a consistent shape.

Activation was also used to add non-linearity to the representation.

Spatial dropout was used to drop a random number of filters, assisting the network to maintain the learning of meaningful filters, while still allowing protection from overfitting the data.

Max pooling was used to down sample the data. This reduces the size of the representation and therefore only keeps the most important features of the data, aiding the process as it repeats.

Augmentation

As can be seen in *Figure 1* below, augmentation was performed on the training images to attempt to improve the prediction results of the deep convoluted neural network (DCNN) model. This augmentation was performed using the Sequential Keras operation and included rotations between -10 and +10 degrees, moderate translations of horizontal and vertical shifts by +/- 5% of the images' width/height, scaling changes with a 10% range, and reflections on the x and y axes. These augmentation values were chosen as they were realistic for the data set whilst preserving the image information that could be gained.

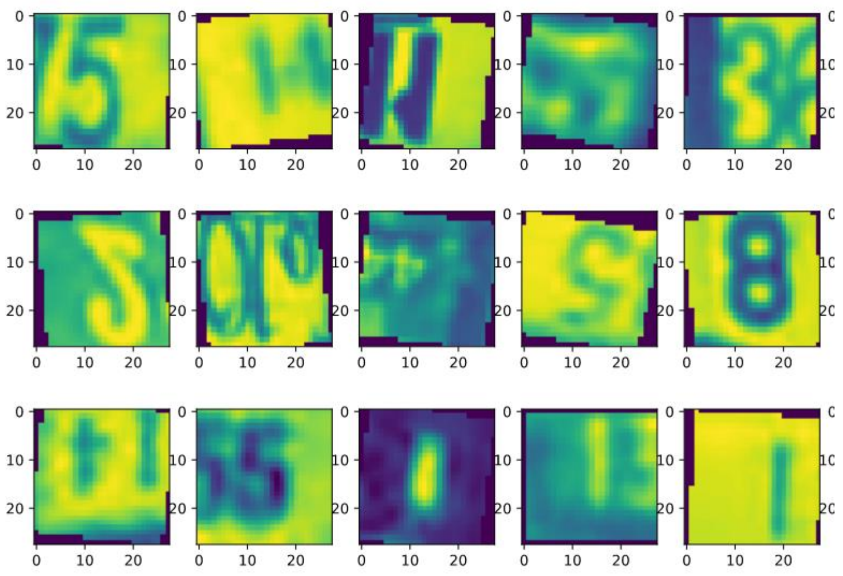


Figure 1: Augmentation on some images from the training data for the augmentation DCNN model.

Pre-Training

The data set used to pre-train the model was the K-MNIST dataset. This chosen model to train was a simple, VGG-like DCNN. This model was chosen due its high performance and its shorter computation time. It was considered because VGG uses multiple dense layers to learn how to map the convolutional features in the confusion matrix, unlike a ResNet model which only uses one.

Key Parameters and Considerations

The data was reshaped to a 4D structure for Keras; the number of samples (-1), width (28), height (28), and number of channels (1). The data was also converted to grayscale and resized to be 28x28 for use in the models to allow for shorter computational times. The batch size, number of epochs, and the steps for epochs were set as constants to use for each model; 8, 20, and 100 respectively. These numbers were selected due to the computational requirements of the models and the models produced during fine-tuning; the fine-tuning process included changing these values so the best number could be chosen for each.

Discussion of Computational Considerations

The training and execution times of deep learning models is affected by the batch size, the computational time of a single epoch passing through the training data, the number of steps within that epoch, and the number of epochs needed to be performed. This time could range from minutes to hours based on the model design and computational power of the hardware it is executed on.

Large amounts of data may need to be processed; some of this data may have a high impact on the outcome of the model and some may not. Consideration is needed in model design on possible ways to filter out the unimportant data or to increase the amount of important data. This generally is affected by changing the batch size which in turn may improve or impair the accuracy of the model.

Each epoch and its steps may improve the accuracy of the model. However, this also increases the execution time overall. Consideration is needed in the design on the number of epochs performed for the desired level of accuracy.

Execution time can be reduced by processing data in batches and using parallel computing on the training process. Consideration is needed in the model design on the size of these batches based on the ability of hardware to support parallel processing. This is mainly affected by the hardware's number of CPU cores and, as CNN is analysing visual imagery, also the number of GPU cores.

Model Comparison

Performance

The performance average for all four models' accuracy was approximately 74.875%. Each model's performance accuracy was impacted by the chosen batch size, number of epochs, and the number of steps within those epochs; as stated, the chosen values were set as they produced the best results during fine-tuning. As there are four models to discuss they will be referenced as SVM, non-augmentation DCNN (trained from scratch, without augmentation), augmentation DCNN (trained from scratch, with augmentation), and pre-trained DCNN (pre-trained using the K-MNIST dataset, without augmentation) from now on.

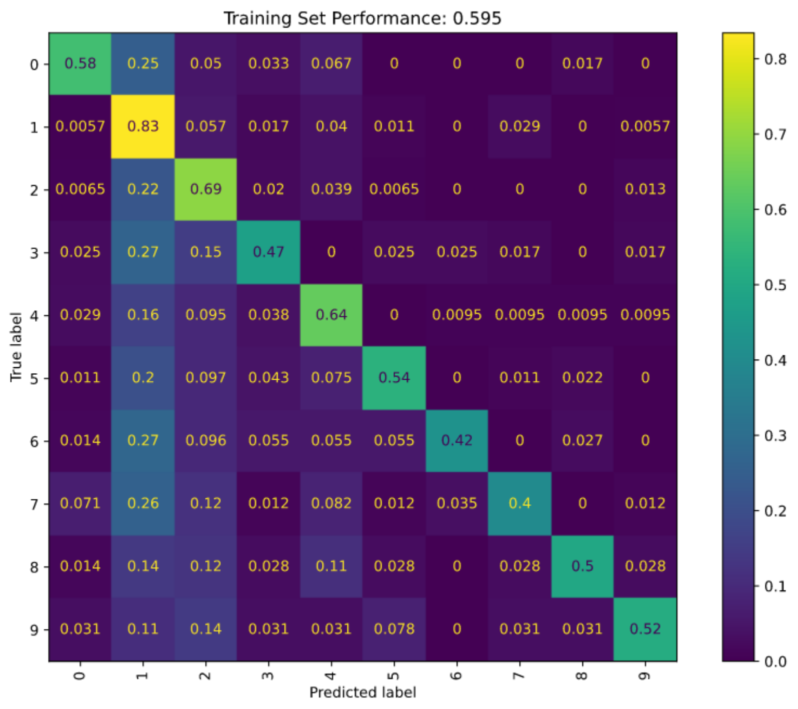


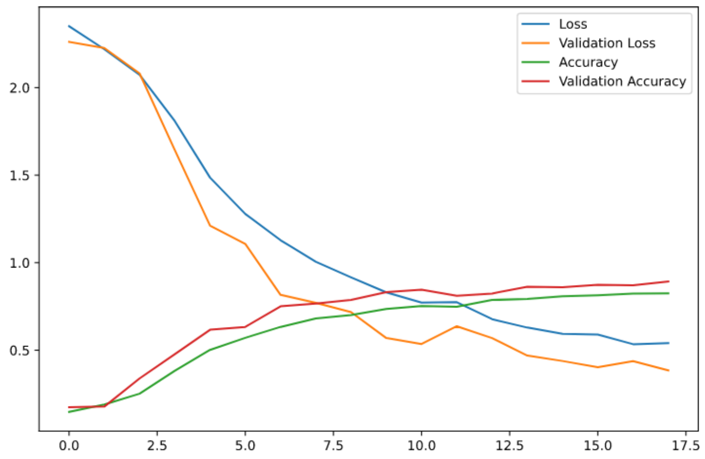
Figure 2: The linear one vs one SVM model, evaluated on the vectorized training set.

In *Figure 2* above, the SVM model can be seen. This model had an accuracy of 59.5%, as shown in the title label for the model. This is quite low, demonstrating that an SVM model is not the best fit to represent the data. This accuracy sets the precedence for what the deep learning models should achieve and ideally surpass in terms of accuracy.

The performance accuracy of the non-augmentation DCNN can be seen in *Figure 3* below. As shown, the overall accuracy of the model was 92%, surpassing the accuracy of the SVM model. This 92% accuracy can be seen in the distribution of the confusion matrix; true values are mostly being predicted correctly. The performance of each class (their f1-scores) also seems consistent with little variation, though the smaller classes 6 and 8 had performances of 0.87 and 0.85 respectively – lower accuracy than the other classes which were all 90 or above.

The training and validation accuracy and loss can also be seen in the *Figure 3* graph below. The training and validation loss seems to indicate a good fit for the data as they both decrease together and begin to level out as the training completes at around 0.5. Although, the validation loss is less stable than the training loss, as they both decrease and reach similar points, it showcases a good fit. The training and validation accuracies mapped also showcase a good fit as they both increase and stabilise together. Overall, this network had a high accuracy and performed better than the SVM, while also fitting the provided data.

```
              precision    recall  f1-score   support
           0       0.88      0.94      0.91       711
           1       0.96      0.95      0.96      1894
           2       0.97      0.95      0.96      1497
           3       0.90      0.91      0.91      1141
           4       0.92      0.97      0.94      1035
           5       0.88      0.92      0.90       892
           6       0.88      0.86      0.87       758
           7       0.91      0.95      0.93       789
           8       0.95      0.77      0.85       683
           9       0.89      0.89      0.89       600

    accuracy                           0.92     10000
   macro avg       0.92      0.91      0.91     10000
weighted avg       0.92      0.92      0.92     10000
```



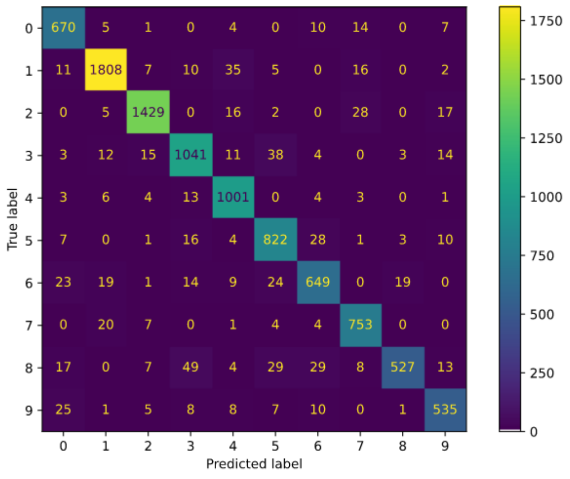
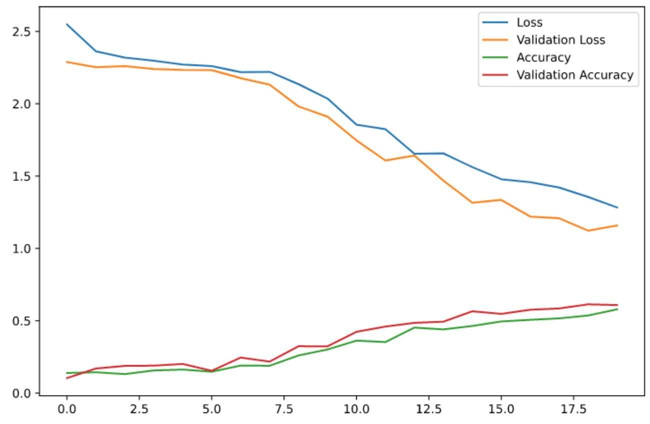
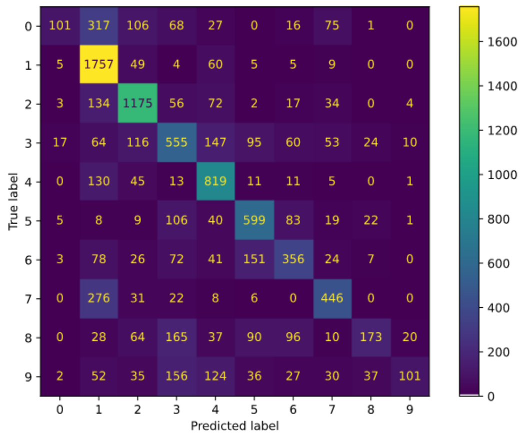


Figure 3: The accuracy and models for the non-augmentation DCNN, trained from scratch without augmentation, evaluated on the training set.

```
              precision    recall  f1-score   support
           0       0.74      0.14      0.24       711
           1       0.62      0.93      0.74      1894
           2       0.71      0.78      0.75      1497
           3       0.46      0.49      0.47      1141
           4       0.60      0.79      0.68      1035
           5       0.60      0.67      0.63       892
           6       0.53      0.47      0.50       758
           7       0.63      0.57      0.60       789
           8       0.66      0.25      0.37       683
           9       0.74      0.17      0.27       600

    accuracy                           0.61     10000
   macro avg       0.63      0.53      0.52     10000
weighted avg       0.62      0.61      0.58     10000
```



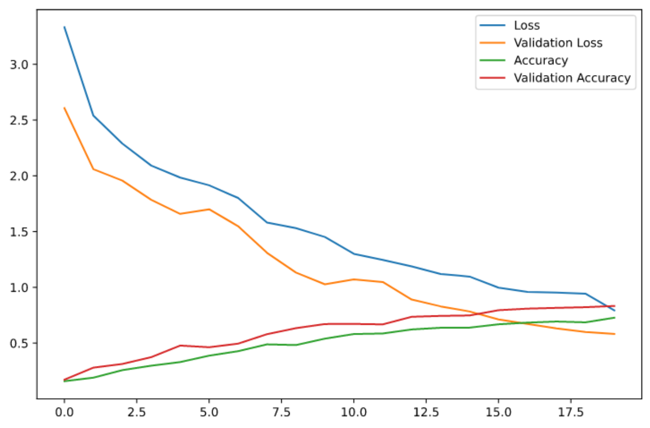


True label	0	1	2	3	4	5	6	7	8	9
0	101	317	106	68	27	0	16	75	1	0
1	5	1757	49	4	60	5	5	9	0	0
2	3	134	1175	56	72	2	17	34	0	4
3	17	64	116	555	147	95	60	53	24	10
4	0	130	45	13	819	11	11	5	0	1
5	5	8	9	106	40	599	83	19	22	1
6	3	78	26	72	41	151	356	24	7	0
7	0	276	31	22	8	6	0	446	0	0
8	0	28	64	165	37	90	96	10	173	20
9	2	52	35	156	124	36	27	30	37	101

Figure 4: The accuracy and models for the augmentation DCNN, trained from scratch with augmentation, evaluated on the training set.

Figure 4 above displays the augmentation DCNN. This model has an accuracy of 61%, lower than that of the non-augmentation DCNN though still higher than the SVM. Also, the f1-scores of the classes are not as consistent as the non-augmentation DCNN ranging from 0.24 to 0.75. From these accuracy values alone, it can be judged that this model did not work as well as the model without augmentation. This is unexpected as with augmentation there is more variation on the images, therefore causing them to be easier to classify. The loss and validation loss are also not great as they stay quite high (approximately 1.3), though they do decrease and stabilise together, indicating a good fit. In future iterations of the augmentation DCNN performance may be enhanced by increasing the batch size or number of epochs. For this representation, these were not increased for consistency purposes.

```
              precision    recall  f1-score   support

           0       0.82      0.89      0.85       711
           1       0.88      0.94      0.91      1894
           2       0.91      0.93      0.92      1497
           3       0.96      0.69      0.80      1141
           4       0.76      0.96      0.85      1035
           5       0.78      0.91      0.84       892
           6       0.84      0.85      0.85       758
           7       0.97      0.84      0.90       789
           8       0.88      0.70      0.78       683
           9       0.92      0.79      0.85       600

    accuracy                           0.87     10000
   macro avg       0.87      0.85      0.86     10000
weighted avg       0.87      0.87      0.86     10000
```



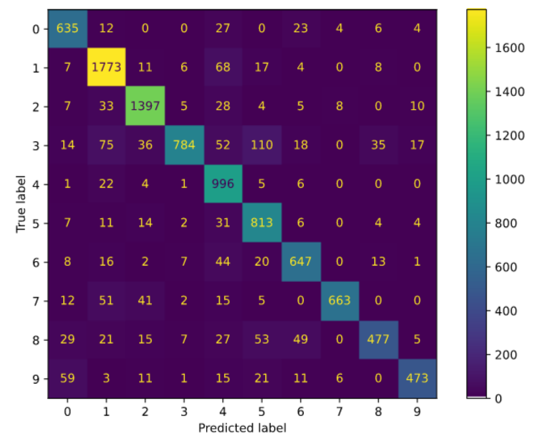


Figure 5: The accuracy and models for the pre-trained DCNN, without augmentation, evaluated on the training set.

Figure 5 above showcases the accuracy of the pre-trained DCNN and its training/validation accuracy/loss graph. This model's accuracy was determined to be 87%, with each class mostly showcasing consistent results; the exception being class 8 with 0.78 accuracy. The confusion matrix confirms this accuracy as the true values were mostly predicted correctly with some variation. The training and validation loss are also good; they both decrease and then stabilise at the same point, around 1, indicating a good fit.

From each model performance and mapped training/validation loss, it can be seen all but the SVM fit the data well. The DCNN with the best overall performance was the non-augmentation DCNN with an accuracy of 92% and most loss, stabilising at around 0.5.

Training and Inference Times

The time it took to train and predict (inference time) each model depended on the number of epochs, their steps, and the complexity of the data. Each of these models where calibrated so they only took a few minutes maximum to train to maximize the efficiency of the fine-tuning stage and allow checking the models to not take too long.

In *Table 1* below, the training output and times for each model can be seen. Each model took 100 steps for each epoch (as set), the time per step averaging at around 26ms for the non-augmentation DCNN, 10ms for the augmentation DCNN, and 25ms for the pre-trained

DCNN. The training times show that the pre-trained DCNN was the quickest to train for our data at 48.62 seconds, with the non-augmentation DCNN coming second at 1 minute 22.97 seconds, and augmentation DCNN coming third at 3 minutes and 23.76 seconds. From these times, and the performance results, the non-augmentation DCNN still ranks the highest as it provided accurate data at a fast speed.

DCNN Model	Summarised Training Output (Epochs 1, 2, and 20)	Training Time
Non-Augmentation DCNN	Epoch 1/20 100/100 [==============================] - 4s 27ms/step - loss: 2.4908 - accuracy: 0.1462 - val_loss: 2.2704 - val_accuracy: 0.1750 Epoch 2/20 100/100 [==============================] - 2s 25ms/step - loss: 2.3320 - accuracy: 0.1550 - val_loss: 2.2679 - val_accuracy: 0.1750 … Epoch 20/20 100/100 [==============================] - 3s 25ms/step - loss: 1.4858 - accuracy: 0.4950 - val_loss: 1.2491 - val_accuracy: 0.6260	0:01:22.969734
Augmentation DCNN	Epoch 1/20 100/100 [==============================] - 12s 105ms/step - loss: 2.5487 - accuracy: 0.1388 - val_loss: 2.2880 - val_accuracy: 0.1035 Epoch 2/20 100/100 [==============================] - 8s 76ms/step - loss: 2.3627 - accuracy: 0.1437 - val_loss: 2.2522 - val_accuracy: 0.1696 … Epoch 20/20 100/100 [==============================] - 8s 76ms/step - loss: 1.2829 - accuracy: 0.5788 - val_loss: 1.1587 - val_accuracy: 0.6082	0:03:23.764151
Pre-Trained DCNN	Epoch 1/20 100/100 [==============================] - 4s 26ms/step - loss: 3.3318 - accuracy: 0.1587 - val_loss: 2.6074 - val_accuracy: 0.1720 Epoch 2/20 100/100 [==============================] - 2s 24ms/step - loss: 2.5394 - accuracy: 0.1900 - val_loss: 2.0597 - val_accuracy: 0.2800 … Epoch 20/20 100/100 [==============================] - 2s 23ms/step - loss: 0.7932 - accuracy: 0.7275 - val_loss: 0.5818 - val_accuracy: 0.8330	0:00:48.615545

Table 1: Summarised training output and the times calculated for each DCNN model.

In *Table 2* below, the time it took to make predictions on each model, the inference times, can be seen. Again, the pre-trained DCNN completed the fastest at 3.96 seconds, with the augmentation DCNN coming second at 4.18 seconds, and non-augmentation DCNN third at 4.3 seconds. As these times are so short, they do not affect the overall performance of the models.

DCNN Model	Inference Time (seconds)
Non-Augmentation DCNN	04.295984
Augmentation DCNN	04.178070
Pre-Trained DCNN	03.962496

Table 2: The inference times calculated for each DCNN model.

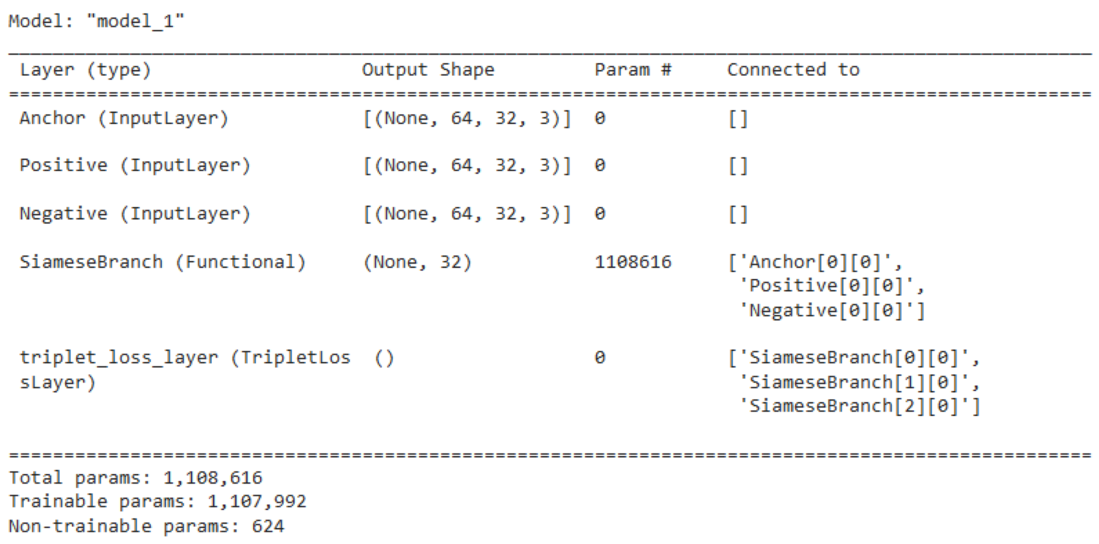
Problem 2: Person Re-Identification

Data Characteristics and Pre-processing

The data provided showcased similar images of people to identify with multiple different people and the same people in multiple images in varied positions. To use the provided data for modelling it was resized to 64x32. This pre-processing was done as issues arose when attempting to train the unprocessed data. The training would either take too long to compile or crash mid-way through; the original data was too large. For these reasons the data was scaled, whilst preserving the information, to allow shorter training times and less risk of failure.

Details and Justification of Approach Selection

The chosen approaches to train the data was Principal Component Analysis (PCA) into Linear Discriminate Analysis (LDA) and Siamese Networks. The model for each method was different; the PCA model was created via the sklearn PCA function, the LDA model was created via the sklearn LinearDiscriminateAnalysis function, and the Siamese model was created using input layers anchor, positive, and negative. The Siamese model also included a functional SiameseBranch and a triplet loss layer, as can be seen in *Figure 6* below.



```
Model: "model_1"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 Anchor (InputLayer)            [(None, 64, 32, 3)]  0           []

 Positive (InputLayer)          [(None, 64, 32, 3)]  0           []

 Negative (InputLayer)          [(None, 64, 32, 3)]  0           []

 SiameseBranch (Functional)     (None, 32)           1108616     ['Anchor[0][0]',
                                                                  'Positive[0][0]',
                                                                  'Negative[0][0]']

 triplet_loss_layer (TripletLos  ()                  0           ['SiameseBranch[0][0]',
 sLayer)                                                          'SiameseBranch[1][0]',
                                                                  'SiameseBranch[2][0]']

==================================================================================================
Total params: 1,108,616
Trainable params: 1,107,992
Non-trainable params: 624
```

Figure 6: The created model for the Siamese Network

Each of these models were chosen as they are fast to train and fit the data well. As seen in *Table 3* below, each model took less than 7 minutes to train and only milliseconds to predict (inference time).

Model	Training Time (minutes)	Inference Time (seconds)
PCA	01:21.954921	01.189961
LDA	01:41.507781	00.198185
Siamese Network	06:58.864229	00.059419

Table 3: The training and inference times for each model.

Other models were used during testing and these chosen methods gave the best results for a final model. In relation to the LDA method, this was chosen over Eigenfaces as that model requires the data to match, i.e., if the data was of faces the features of each face would have to be aligned at the same heights, etc. This type of data processing could not be done on our dataset as the images were not always the same; a person may be standing facing the camera or away from it, etc. As for the Siamese method, this was used as our images are all quite similar and this network seems to perform the best in terms of computational times and requirements; uses a simple VGG method.

The data used to train each model was the training data. Then the CMC curves were calculated on the gallery and probe data so the top 1, 5, and 10 results could be seen.

Model Comparison

Performance

The performance for each model will be based off the accuracy for the Rank-1, Rank-5, and Rank-10 accuracies. This means the accuracy comes from the chance that the gallery will return the matching sample to the probe in the top-n results.

In *Table 4* below the number of samples found for each rank for each model can be seen. For the LDA model, 1 gallery sample was found to match the probe sample for Rank-1, 4 matches for Rank-5, and 1 matched for Rank-10. This looks accurate as the people matched in Rank-1 are those who are easy to tell apart, whereas those in Rank-10 are less so. For LDA, the increasing pattern does not occur with 1 sample matched for Rank-1, 4 for Rank-5, and 1 for Rank-10, showcasing the model does not work as well as PCA's model. Finally, the Siamese model shows the samples matched were 77 for Rank-1, 9 for Rank-5, and 3 for Rank-10. This suggests the model for the Siamese method has the highest performance as more samples can be matched for each ranking.

Model	Samples		
	Rank-1	Rank-5	Rank-10
PCA	11	2	1
LDA	1	4	1
Siamese	77	9	3

Table 4: The number of samples in each rank of the models.

In Figure 7 below, CMC curves for each model can be seen. These map the true positive identification rates (TPIR) against the rank. It can be seen the best CMC was the one for the Siamese method as for Rank-1 the accuracy of matching samples is around 25%, whereas for the PCA and LDA for Rank-1 it is around 5% (guesstimated via the curve). The accuracy does get better for Rank-5 and Rank-10 but not by much, with the highest accuracy in sample matching seen for Rank-300. This showcases the best model in terms of accuracy is the Siamese, PCA coming second, and LDA third; the Siamese Networks method had the best accuracy from the beginning and was therefore the best for the dataset, which was in line with the results found from the sample match data.

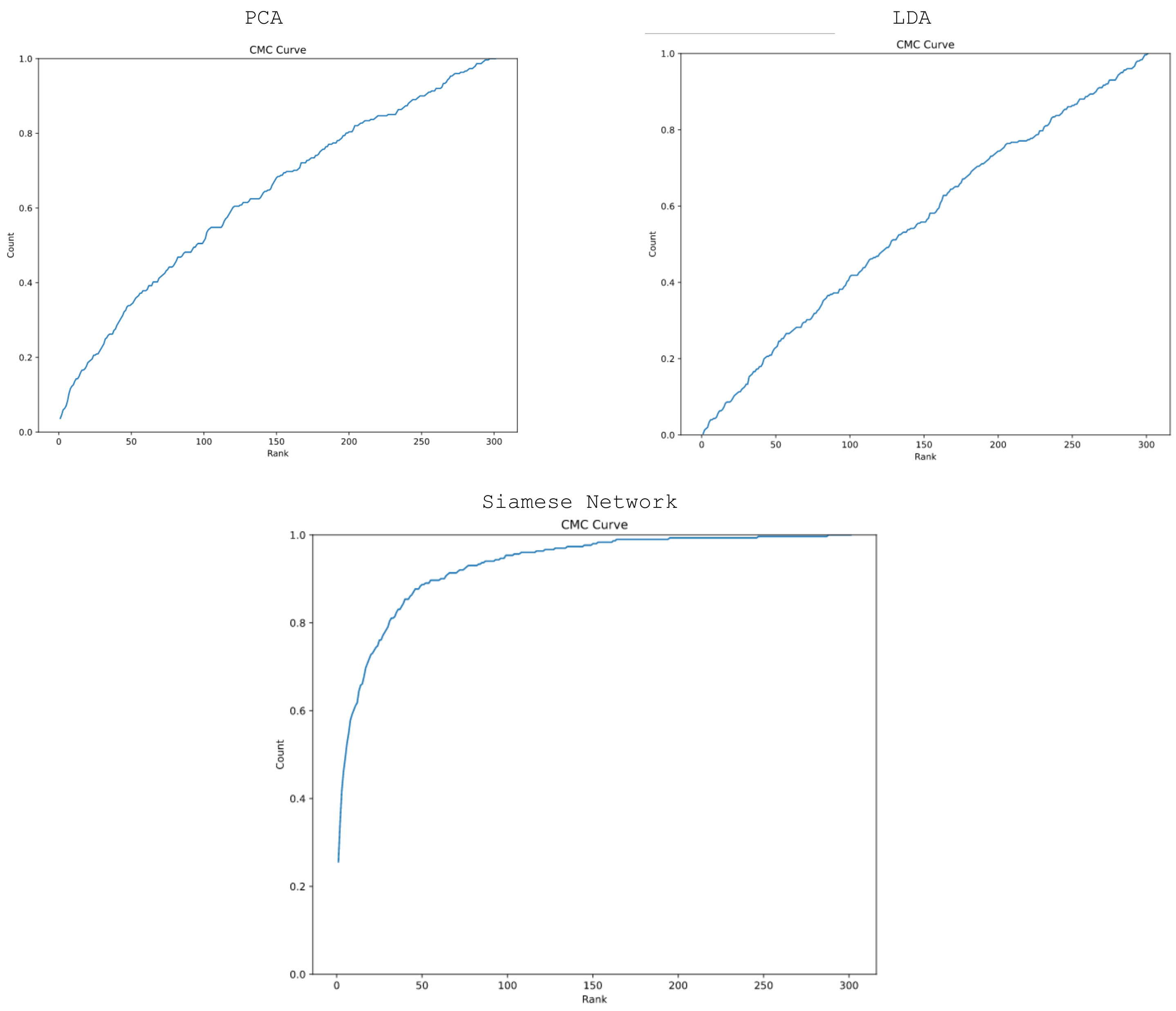


Figure 7: The CMC curves for PCA, LDA, and Siamese Network.

Strengths and Weaknesses

As mentioned, when justifying approach selection, the training and inference times were quite fast. As can be seen in *Table 5* below, the PCA to LDA method did not have a trained model output and took approximately 1 minute and 21.94 seconds, 1 minute 41.51 seconds respectively. Overall, this method took approximately 3.06 minutes. As for the Siamese method, it took approximately 6 minutes and 58.86 seconds to train. From this it can be seen the Siamese method took over double the time to train than the PCA to LDA method. The computational runtime for the PCA to LDA method is a strength and is a slight weakness for the Siamese Network, especially using bigger batch sizes, number or epochs, and steps per epoch.

Model	Summarised Training Output (Epochs 1, 2, and 10)	Training Time (minutes)	Inference Time (seconds)
PCA	Nil	01:21.954921	01.189961
LDA	Nil	01:41.507781	00.198185
Siamese Network	Epoch 1/10 15/15 [==============================] - 51s 3s/step - loss: 81.3049 - val_loss: 28.6904 Epoch 2/10 15/15 [==============================] - 45s 3s/step - loss: 52.4663 - val_loss: 21.0407 … Epoch 10/10 15/15 [==============================] - 40s 3s/step - loss: 28.6465 - val_loss: 7.0043	06:58.864229	00.059419

Table 5: The summarised training out and training and inference times for the models.

Overall, even with the longer training time, the Siamese network worked the best for matching the correct samples to each rank and was therefore the best model for the data.

Appendix

All code used was built upon using code from Lecture examples and Tutorial solutions.

n10477659 Final Assignment_1B

May 15, 2022

1 Assignment 1B

2 Import Functions

[59]:
```
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

# numpy handles pretty much anything that is a number/vector/matrix/array
import numpy as np
# pandas handles dataframes
import pandas as pd
# matplotlib emulates Matlabs plotting functionality
import matplotlib.pyplot as plt
# seaborn is another good plotting library. In particular, I like it for␣
 ↪heatmaps (https://seaborn.pydata.org/generated/seaborn.heatmap.html)
import seaborn as sns;
# stats models is a package that is going to perform the regression analysis
from statsmodels import api as sm
from scipy import stats
from sklearn.metrics import mean_squared_error, r2_score
# os allows us to manipulate variables on out local machine, such as paths and␣
 ↪environment variables
import os
# self explainatory, dates and times
from datetime import datetime, date
# a helper package to help us iterate over objects
import itertools

import time

## Q1
import os
import datetime

import tensorflow as tf
```

```
from tensorflow import keras
from tensorflow.keras import layers
from tensorboard import notebook
from tensorflow.keras.preprocessing.image import ImageDataGenerator

import sklearn
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from sklearn.metrics import precision_score, recall_score, f1_score,␣
↪classification_report
from sklearn.svm import SVC, NuSVC
from tensorflow.keras.callbacks import ModelCheckpoint


# tf.keras.backend.clear_session()

## Q2
from sklearn import decomposition
from sklearn import discriminant_analysis
from sklearn import datasets
from sklearn.manifold import TSNE
from sklearn.neighbors import KNeighborsClassifier

import random
from tensorflow.keras import backend as K

# To export as pdf with better quality plots
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('pdf', 'svg')
```

```
Mounted at /content/drive
```

3 Problem 1. Training and Adapting Deep Networks.

Using these datasets you are to: 1. Train a Linear One vs One SVM (with C = 1), on the provided abridged SVHN training set. 2. Design/select a DCNN architecture and using this one architecture:

(a) Train a model from scratch, using no data augmentation, on the provided abridged SVHN training set.

(b) Train a model from scratch, using the data augmentation of your choice, on the provided abridged SVHN training set.

(c) Fine tune an existing model, trained on another dataset used in CAB420 (such as MNIST, KMINST or CIFAR), on the provided abridged SVHN training set. Data augmentation may also be used if you so choose.

3. Compare the performance of the four, considering the accuracy, training time and inference time (i.e. time taken to evaluate the models), using the provided test set.

All models should be evaluated on the provided SVHN test set, and their performance should be compared. DCNN architectures and pre-trained models may be taken from lecture ex- amples or practical solutions. Your selection of model may take computational constraints into consideration.

You have been provided with sample python code that will load and prepare the dataset for training for both the One vs One SVM, and DCNN. Sample code also illustrates how you can time a process with Python, and how to resize images and convert them to greyscale if this is required for your approach. Note that any pre-processing (i.e. resizing of images, colour conversion) should be applied for all models (SVM and DCNNs) to enable a fair comparison.

[116]:
```
#
# Utility functions for CAB420, Assignment 1B, Q1
# Author: Simon Denman (s.denman@qut.edu.au)
#

from scipy.io import loadmat        # to load mat files
import matplotlib.pyplot as plt     # for plotting
import numpy as np                  # for reshaping, array manipulation
import cv2                          # for colour conversion
import tensorflow as tf             # for bulk image resize

# Load data for Q1
#  train_path: path to training data mat file
#  test_path:  path to testing data mat file
#
#  returns:    arrays for training and testing X and Y data
#
def load_data(train_path, test_path):

    # load files
    train = loadmat(train_path)
    test = loadmat(test_path)

    # transpose, such that dimensions are (sample, width, height, channels),␣
 ↪and divide by 255.0
    train_X = np.transpose(train['train_X'], (3, 0, 1, 2)) / 255.0
    train_Y = train['train_Y']
    # change labels '10' to '0' for compatability with keras/tf. The label '10'␣
 ↪denotes the digit '0'
    train_Y[train_Y == 10] = 0
    train_Y = np.reshape(train_Y, -1)

    # transpose, such that dimensions are (sample, width, height, channels),␣
 ↪and divide by 255.0
    test_X = np.transpose(test['test_X'], (3, 0, 1, 2)) / 255.0
    test_Y = test['test_Y']
    # change labels '10' to '0' for compatability with keras/tf. The label '10'␣
 ↪denotes the digit '0'
```

```
    test_Y[test_Y == 10] = 0
    test_Y = np.reshape(test_Y, -1)

    # return loaded data
    return train_X, train_Y, test_X, test_Y

# vectorise an array of images, such that the shape is changed from {samples,␣
 ↪width, height, channels} to
# (samples, width * height * channels)
#   images: array of images to vectorise
#
#   returns: vectorised array of images
#
def vectorise(images):
    # use numpy's reshape to vectorise the data
    return np.reshape(images, [len(images), -1])

# Plot some images and their labels. Will plot the first 100 samples in a 10x10␣
 ↪grid
#  x: array of images, of shape (samples, width, height, channels)
#  y: labels of the images
#
def plot_images(x, y):
    fig = plt.figure(figsize=[15, 18])
    for i in range(100):
        ax = fig.add_subplot(10, 10, i + 1)
        ax.imshow(x[i,:])
        ax.set_title(y[i])
        ax.axis('off')

# Resize an array of images
#  images:   array of images, of shape (samples, width, height, channels)
#  new_size: tuple of the new size, (new_width, new_height)
#
#  returns:  resized array of images, (samples, new_width, new_height, channels)
#
def resize(images, new_size):
    # tensorflow has an image resize funtion that can do this in bulk
    # note the conversion back to numpy after the resize
    return tf.image.resize(images, new_size).numpy()

# Convert images to grayscale
#   images:  array of colour images to convert, of size (samples, width,␣
 ↪height, 3)
#
#   returns: array of converted images, of size (samples, width, height, 1)
#
```

```
def convert_to_grayscale(images):
    # storage for converted images
    gray = []
    # loop through images
    for i in range(len(images)):
        # convert each image using openCV
        gray.append(cv2.cvtColor(images[i,:], cv2.COLOR_BGR2GRAY))
    # pack converted list as an array and return
    return np.array(gray)
```

[117]:
```
# Load the data
train_path = '/content/drive/My Drive/Assignment 1B/CAB420_Assessment_1B_Data/
 ↪Q1/q1_train.mat'
test_path = '/content/drive/My Drive/Assignment 1B/CAB420_Assessment_1B_Data/Q1/
 ↪q1_test.mat'

train, train_y, test, test_y = load_data(train_path, test_path)
```

[118]:
```
## Convert to greyscale and resize
train = convert_to_grayscale(resize(train, (28, 28)))
test = convert_to_grayscale(resize(test, (28, 28)))

print(np.shape(train))
print(np.shape(test))
```

```
(1000, 28, 28)
(10000, 28, 28)
```

[105]:
```
## Set constants
num_epochs = 20
batch_size = 8
steps_per_epoch = 100
```

3.1 1. Train a Linear One vs One SVM (with C = 1), on the provided abridged SVHN training set.

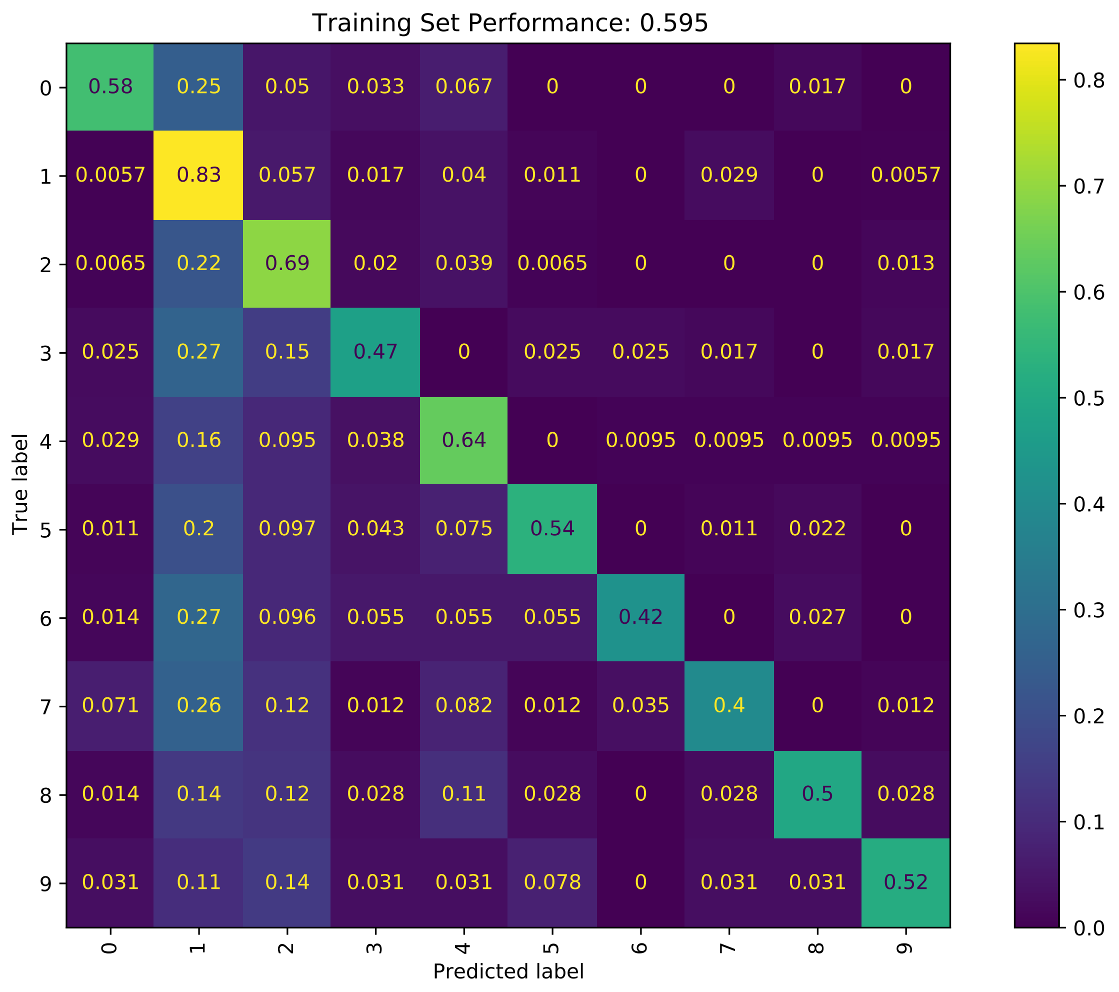
[119]:
```
## Vectorise for SVM
train_vector_X = vectorise(train)
test_vector_X = vectorise(test)

print(train_vector_X.shape)
print(test_vector_X.shape)
```

```
(1000, 784)
```

```
(10000, 784)
```

[120]:
```
# function to do our eval for us, this is quite simple and will
# - create a figure
# - draw a confusion matrix for the trainign data in a sub-fig on the left
# - draw a confusion matrix for the testing data in a sub-fig on the right
# - compute the overall classification accuracy on the testing data
# this has simply been created as we're going to do this for each test that we␣
↪run
def eval_model(model, X_train, Y_train):
    fig = plt.figure(figsize=[25, 8])
    ax = fig.add_subplot(1, 2, 1)
    conf = ConfusionMatrixDisplay.from_estimator(model, X_train, Y_train,␣
↪normalize="true", xticks_rotation='vertical', ax=ax)
    pred = model.predict(X_train)

    conf.ax_.set_title('Training Set Performance: ' + str(sum(pred == Y_train)/
↪len(Y_train)));
```

[121]:
```
## SVM
C = 1; kernel = 'linear'
params = {'C': 1, 'kernel': 'linear'}
svm = SVC().set_params(**params)
svm.fit(train_vector_X, train_y)
eval_model(svm, train_vector_X, train_y)
```



3.2 2. Design/select a DCNN architecture and using this one architecture:

[64]:
```
## Reshape data
train = train.reshape(-1, 28, 28, 1)
test = test.reshape(-1, 28, 28, 1)

train_y = train_y.reshape(-1, 1)
test_y = test_y.reshape(-1, 1)

print(np.shape(train))
print(np.shape(train_y))
print(np.shape(test))
print(np.shape(test_y))
```

```
(1000, 28, 28, 1)
(1000, 1)
(10000, 28, 28, 1)
(10000, 1)
```

[65]:
```
def eval_model(model, history, x_train, y_train, x_test, y_test):

    fig = plt.figure(figsize=[20, 6])

    ax = fig.add_subplot(1, 2, 1)
    ax.plot(history['loss'], label='Loss')
    ax.plot(history['val_loss'], label='Validation Loss')
    ax.plot(history['accuracy'], label='Accuracy')
    ax.plot(history['val_accuracy'], label='Validation Accuracy')
    ax.legend()

    pred = model.predict(x_test);
    indexes = tf.argmax(pred, axis=1)
    cm = confusion_matrix(y_test, indexes)

    ax = fig.add_subplot(1, 2, 2)
    c = ConfusionMatrixDisplay(cm, display_labels=range(len(np.unique(y_test))))
    c.plot(ax = ax)

    print(classification_report(y_test, indexes))
```

[106]:
```
# num_epochs = 20

# function to build a model, takes the number of classes. Can optionally change␣
↪the output activation.
# See the discussion at bottom of this script for more details about that.
def build_model(num_classes, output_activation=None, with_augmentation = False):
    # our model, input in an image shape
    inputs = keras.Input(shape=(28, 28, 1, ), name='img')

    if (with_augmentation):
      x = data_augmentation(inputs)
    #   x = layers.Conv2D(16, 3, activation='swish')(x)
    # else:
    #   x = layers.Conv2D(16, 3, activation='swish')(inputs)

    # run pairs of conv layers, all 3s3 kernels
    x = layers.Conv2D(filters=8, kernel_size=(3,3), padding='same',␣
↪activation='relu')(inputs)
```

```
    x = layers.Conv2D(filters=8, kernel_size=(3,3), padding='same',␣
↪activation=None)(x)
    # batch normalisation, before the non-linearity
    x = layers.BatchNormalization()(x)
    x = layers.Activation('relu')(x)
    # spatial dropout, this will drop whole kernels, i.e. 20% of our 3x3␣
↪filters will be dropped out rather
    # than dropping out 20% of the invidual pixels
    x = layers.SpatialDropout2D(0.2)(x)
    # max pooling, 2x2, which will downsample the image
    x = layers.MaxPool2D(pool_size=(2, 2))(x)
    # rinse and repeat with 2D convs, batch norm, dropout and max pool
    x = layers.Conv2D(filters=16, kernel_size=(3,3), padding='same',␣
↪activation='relu')(x)
    x = layers.Conv2D(filters=16, kernel_size=(3,3), padding='same',␣
↪activation=None)(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation('relu')(x)
    x = layers.SpatialDropout2D(0.2)(x)
    x = layers.MaxPool2D(pool_size=(2, 2))(x)
    # final conv2d, batch norm and spatial dropout
    x = layers.Conv2D(filters=32, kernel_size=(3,3), padding='same',␣
↪activation='relu')(x)
    x = layers.Conv2D(filters=32, kernel_size=(3,3), padding='same',␣
↪activation=None)(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation('relu')(x)
    x = layers.SpatialDropout2D(0.2)(x)

    # flatten layer
    x = layers.Flatten()(x)
    # we'll use a couple of dense layers here, mainly so that we can show what␣
↪another dropout layer looks like
    # in the middle
    x = layers.Dense(256, activation='relu')(x)
    x = layers.Dropout(0.5)(x)
    x = layers.Dense(64, activation='relu')(x)
    # the output
    outputs = layers.Dense(num_classes, activation=output_activation)(x)

    # build the model, and print a summary
    model_cnn = keras.Model(inputs=inputs, outputs=outputs,␣
↪name='kmnist_cnn_model')

    return model_cnn
```

```
model_cnn = build_model(10)
model_cnn.summary()
```

```
Model: "kmnist_cnn_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 img (InputLayer)            [(None, 28, 28, 1)]       0         
                                                                 
 conv2d_120 (Conv2D)         (None, 28, 28, 8)         80        
                                                                 
 conv2d_121 (Conv2D)         (None, 28, 28, 8)         584       
                                                                 
 batch_normalization_62 (Bat  (None, 28, 28, 8)        32        
 chNormalization)                                                
                                                                 
 activation_62 (Activation)  (None, 28, 28, 8)         0         
                                                                 
 spatial_dropout2d_60 (Spati  (None, 28, 28, 8)        0         
 alDropout2D)                                                    
                                                                 
 max_pooling2d_40 (MaxPoolin  (None, 14, 14, 8)        0         
 g2D)                                                            
                                                                 
 conv2d_122 (Conv2D)         (None, 14, 14, 16)        1168      
                                                                 
 conv2d_123 (Conv2D)         (None, 14, 14, 16)        2320      
                                                                 
 batch_normalization_63 (Bat  (None, 14, 14, 16)       64        
 chNormalization)                                                
                                                                 
 activation_63 (Activation)  (None, 14, 14, 16)        0         
                                                                 
 spatial_dropout2d_61 (Spati  (None, 14, 14, 16)       0         
 alDropout2D)                                                    
                                                                 
 max_pooling2d_41 (MaxPoolin  (None, 7, 7, 16)         0         
 g2D)                                                            
                                                                 
 conv2d_124 (Conv2D)         (None, 7, 7, 32)          4640      
                                                                 
 conv2d_125 (Conv2D)         (None, 7, 7, 32)          9248      
                                                                 
 batch_normalization_64 (Bat  (None, 7, 7, 32)         128       
 chNormalization)                                                
                                                                 
 activation_64 (Activation)  (None, 7, 7, 32)          0         
 spatial_dropout2d_62 (Spati  (None, 7, 7, 32)          0
 alDropout2D)

 flatten_20 (Flatten)        (None, 1568)              0

 dense_58 (Dense)            (None, 256)               401664

 dropout_18 (Dropout)        (None, 256)               0

 dense_59 (Dense)            (None, 64)                16448

 dense_60 (Dense)            (None, 10)                650

=================================================================
Total params: 437,026
Trainable params: 436,914
Non-trainable params: 112
_________________________________________________________________
```

3.2.1 (a) Train a model from scratch, using no data augmentation, on the provided abridged SVHN training set.

[107]:
```
tic = time.perf_counter()
model_cnn.compile(loss=keras.losses.
 ↪SparseCategoricalCrossentropy(from_logits=True),
              optimizer=keras.optimizers.Adam(),
              metrics=['accuracy'])
history = model_cnn.fit(test, test_y,
                        batch_size=batch_size,
                        steps_per_epoch = steps_per_epoch,
                        epochs=num_epochs,
                        validation_data=(train, train_y))
toc = time.perf_counter()

timer = str(datetime.timedelta(seconds=(toc - tic)))
print(f'Training Timer: {timer} seconds')
```

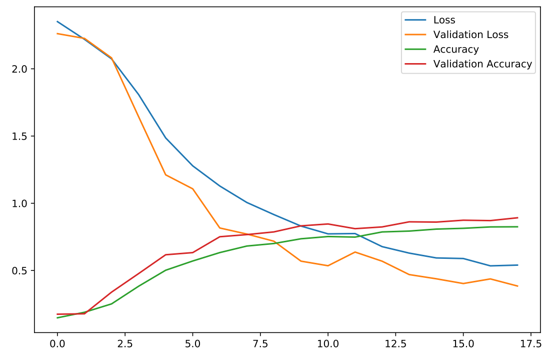
```
Epoch 1/20
100/100 [==============================] - 4s 27ms/step - loss: 2.4908 -
accuracy: 0.1462 - val_loss: 2.2704 - val_accuracy: 0.1750
Epoch 2/20
100/100 [==============================] - 2s 25ms/step - loss: 2.3320 -
accuracy: 0.1550 - val_loss: 2.2679 - val_accuracy: 0.1750
Epoch 3/20
100/100 [==============================] - 2s 25ms/step - loss: 2.2883 -
accuracy: 0.1600 - val_loss: 2.2561 - val_accuracy: 0.1760
```

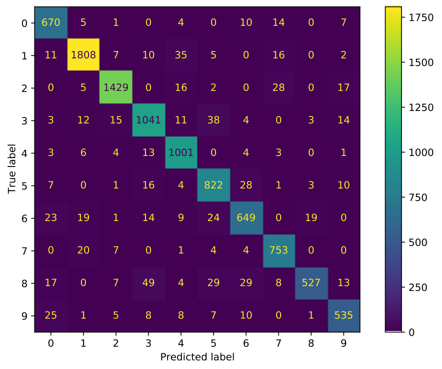
```
Epoch 4/20
100/100 [==============================] - 3s 26ms/step - loss: 2.2742 -
accuracy: 0.1850 - val_loss: 2.2443 - val_accuracy: 0.1750
Epoch 5/20
100/100 [==============================] - 2s 25ms/step - loss: 2.2451 -
accuracy: 0.1725 - val_loss: 2.2180 - val_accuracy: 0.1930
Epoch 6/20
100/100 [==============================] - 3s 25ms/step - loss: 2.2004 -
accuracy: 0.1937 - val_loss: 2.1680 - val_accuracy: 0.2620
Epoch 7/20
100/100 [==============================] - 2s 25ms/step - loss: 2.1621 -
accuracy: 0.2387 - val_loss: 2.0637 - val_accuracy: 0.2830
Epoch 8/20
100/100 [==============================] - 2s 25ms/step - loss: 2.0730 -
accuracy: 0.2637 - val_loss: 1.9993 - val_accuracy: 0.3320
Epoch 9/20
100/100 [==============================] - 3s 25ms/step - loss: 1.9853 -
accuracy: 0.3038 - val_loss: 1.9133 - val_accuracy: 0.3130
Epoch 10/20
100/100 [==============================] - 3s 25ms/step - loss: 1.8666 -
accuracy: 0.3250 - val_loss: 1.7547 - val_accuracy: 0.4070
Epoch 11/20
100/100 [==============================] - 2s 25ms/step - loss: 1.8106 -
accuracy: 0.3250 - val_loss: 1.7644 - val_accuracy: 0.3690
Epoch 12/20
100/100 [==============================] - 3s 25ms/step - loss: 1.7013 -
accuracy: 0.4000 - val_loss: 1.5401 - val_accuracy: 0.4690
Epoch 13/20
100/100 [==============================] - 2s 25ms/step - loss: 1.6741 -
accuracy: 0.3900 - val_loss: 1.5245 - val_accuracy: 0.5150
Epoch 14/20
100/100 [==============================] - 2s 25ms/step - loss: 1.6407 -
accuracy: 0.4275 - val_loss: 1.5616 - val_accuracy: 0.4650
Epoch 15/20
100/100 [==============================] - 2s 25ms/step - loss: 1.5671 -
accuracy: 0.4350 - val_loss: 1.4721 - val_accuracy: 0.5250
Epoch 16/20
100/100 [==============================] - 2s 25ms/step - loss: 1.5729 -
accuracy: 0.4363 - val_loss: 1.4370 - val_accuracy: 0.5250
Epoch 17/20
100/100 [==============================] - 3s 26ms/step - loss: 1.5337 -
accuracy: 0.4725 - val_loss: 1.3868 - val_accuracy: 0.5260
Epoch 18/20
100/100 [==============================] - 3s 26ms/step - loss: 1.4499 -
accuracy: 0.5150 - val_loss: 1.4267 - val_accuracy: 0.5130
Epoch 19/20
100/100 [==============================] - 3s 25ms/step - loss: 1.4580 -
accuracy: 0.4825 - val_loss: 1.2045 - val_accuracy: 0.5860
```

```
Epoch 20/20
100/100 [==============================] - 3s 25ms/step - loss: 1.4858 -
accuracy: 0.4950 - val_loss: 1.2491 - val_accuracy: 0.6260
Training Timer: 0:01:22.969734 seconds
```

[95]:
```
tic = time.perf_counter()
eval_model(model_cnn, history.history, train, train_y, test, test_y)
toc = time.perf_counter()

timer = str(datetime.timedelta(seconds=(toc - tic)))
print(f'Inference Timer: {timer} seconds')
```

```
              precision    recall  f1-score   support

           0       0.88      0.94      0.91       711
           1       0.96      0.95      0.96      1894
           2       0.97      0.95      0.96      1497
           3       0.90      0.91      0.91      1141
           4       0.92      0.97      0.94      1035
           5       0.88      0.92      0.90       892
           6       0.88      0.86      0.87       758
           7       0.91      0.95      0.93       789
           8       0.95      0.77      0.85       683
           9       0.89      0.89      0.89       600

    accuracy                           0.92     10000
   macro avg       0.92      0.91      0.91     10000
weighted avg       0.92      0.92      0.92     10000

Inference Timer: 0:00:04.295984 seconds
```





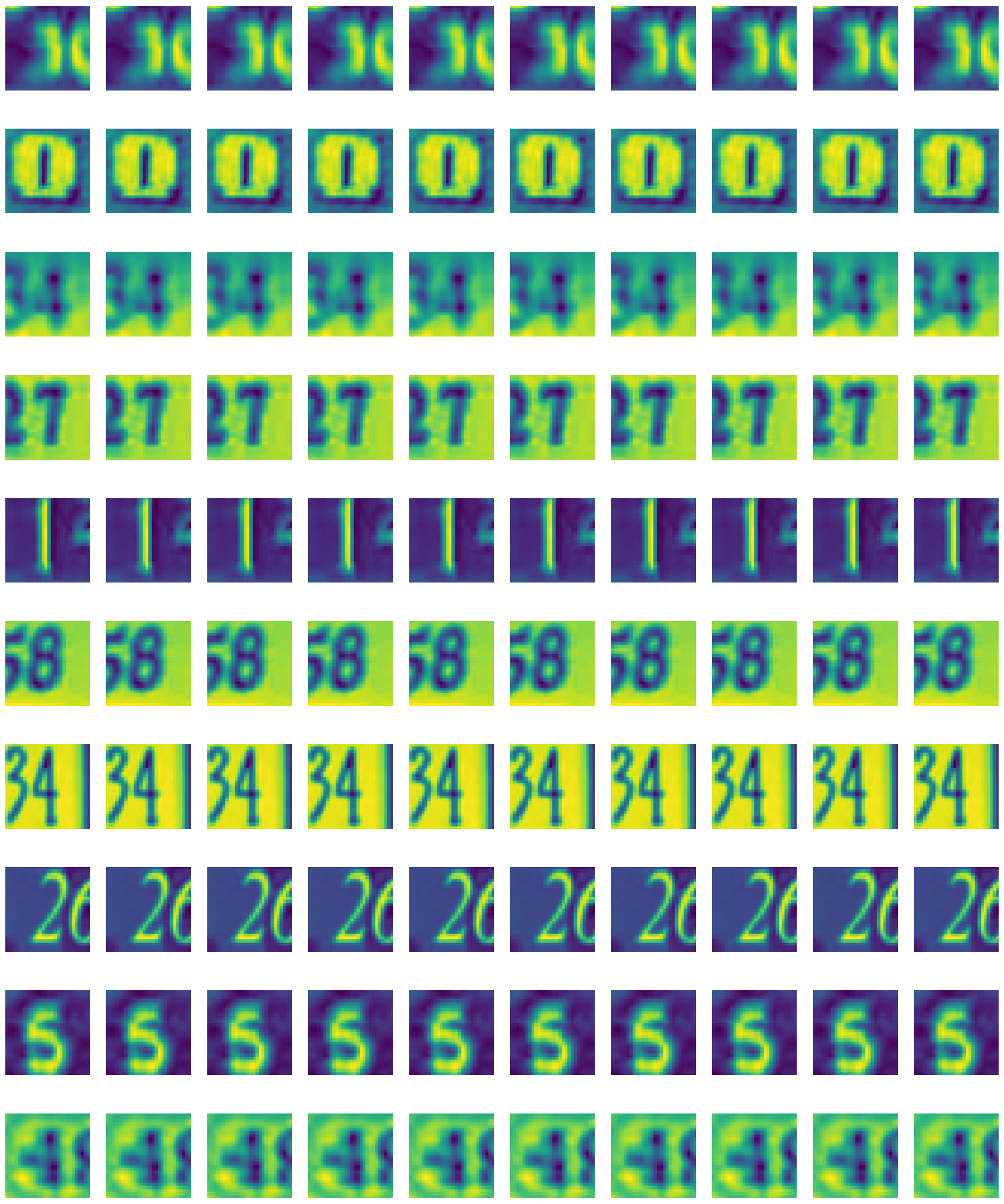
	0	1	2	3	4	5	6	7	8	9
0	670	5	1	0	4	0	10	14	0	7
1	11	1808	7	10	35	5	0	16	0	2
2	0	5	1429	0	16	2	0	28	0	17
3	3	12	15	1041	11	38	4	0	3	14
4	3	6	4	13	1001	0	4	3	0	1
5	7	0	1	16	4	822	28	1	3	10
6	23	19	1	14	9	24	649	0	19	0
7	0	20	7	0	1	4	4	753	0	0
8	17	0	7	49	4	29	29	8	527	13
9	25	1	5	8	8	7	10	0	1	535

3.2.2 (b) Train a model from scratch, using the data augmentation of your choice, on the provided abridged SVHN training set.

[48]:
```
print(train.shape)
```

```
(1000, 28, 28, 1)
```

[96]:
```
data_augmentation = keras.Sequential([
  layers.RandomFlip("horizontal_and_vertical"),
  layers.RandomRotation(0.10),
  layers.RandomTranslation(height_factor=(-0.05, 0.05), width_factor=(-0.05, 0.
↪05)),
  layers.RandomZoom(0.1)
])

fig = plt.figure(figsize=[20, 25])
for i in range(10):
    for j in range(10):
        ax = fig.add_subplot(10, 10, i*10 + (j + 1))
        augmented_image = data_augmentation(tf.expand_dims(train[i,:,:,:],0))
        plt.imshow(vectorise(augmented_image[0]))
        plt.axis("off")

# fig = plt.figure(figsize=[20, 25])
# for i in range(10):
#     for j in range(10):
#         ax = fig.add_subplot(10, 10, i*10 + (j + 1))
#         augmented_image = data_augmentation(tf.expand_dims(train[i,:,:,:],0))
#         img2 = vectorise(augmented_image)
#         ax.imshow(img2)
        # plt.imshow(augmented_image[0])
        # plt.axis("off")

#         batch = datagen.flow(train, train_y, batch_size=100)

# fig = plt.figure(figsize=[20, 25])
# for i,img in enumerate(batch[0][0]):
#     ax = fig.add_subplot(10, 10, i + 1)
#     img2 = vectorise(img)
#     ax.imshow(img2)
```



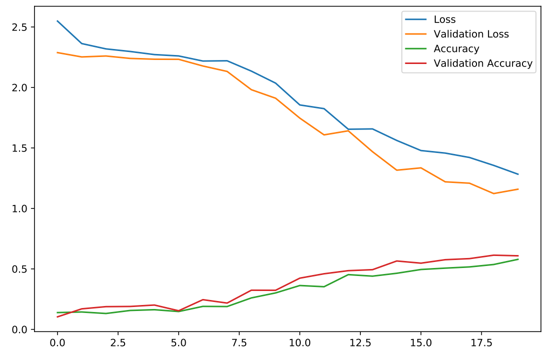
[103]:
```
## Refresh model
model_cnn = build_model(10, with_augmentation = True)

tic = time.perf_counter()
model_cnn.compile(loss=keras.losses.
 ↪SparseCategoricalCrossentropy(from_logits=True),
              optimizer=keras.optimizers.Adam(),
              metrics=['accuracy'])
history = model_cnn.fit(train, train_y, batch_size=8,
                        steps_per_epoch = 100,
                        epochs=20,
                        validation_data=(test, test_y), verbose=True)
toc = time.perf_counter()

timer = str(datetime.timedelta(seconds=(toc - tic)))
print(f'Training Timer: {timer} seconds')

# dataset.repeat(num_epochs).batch(batch_size)


# history = magic_property_forecaster_no_aug.fit(train, train_y, batch_size=32,
#                       epochs=50,
#                       validation_data=(test, test_y), verbose=True)

# eval(magic_property_forecaster_no_aug, history.history, test, test_y)
```

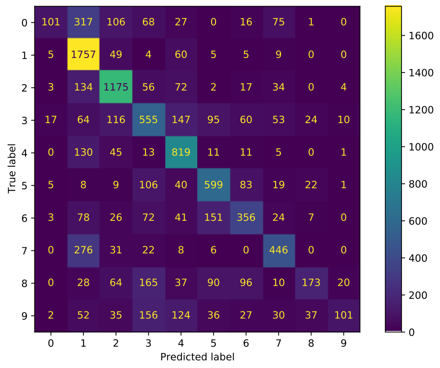
```
Epoch 1/20
100/100 [==============================] - 12s 105ms/step - loss: 2.5487 -
accuracy: 0.1388 - val_loss: 2.2880 - val_accuracy: 0.1035
Epoch 2/20
100/100 [==============================] - 8s 76ms/step - loss: 2.3627 -
accuracy: 0.1437 - val_loss: 2.2522 - val_accuracy: 0.1696
Epoch 3/20
100/100 [==============================] - 8s 77ms/step - loss: 2.3187 -
accuracy: 0.1312 - val_loss: 2.2600 - val_accuracy: 0.1878
Epoch 4/20
100/100 [==============================] - 8s 77ms/step - loss: 2.2973 -
accuracy: 0.1562 - val_loss: 2.2399 - val_accuracy: 0.1894
Epoch 5/20
100/100 [==============================] - 8s 77ms/step - loss: 2.2716 -
accuracy: 0.1625 - val_loss: 2.2333 - val_accuracy: 0.2011
Epoch 6/20
100/100 [==============================] - 8s 78ms/step - loss: 2.2604 -
accuracy: 0.1475 - val_loss: 2.2326 - val_accuracy: 0.1539
Epoch 7/20
100/100 [==============================] - 8s 76ms/step - loss: 2.2183 -
accuracy: 0.1900 - val_loss: 2.1773 - val_accuracy: 0.2454
Epoch 8/20
100/100 [==============================] - 8s 76ms/step - loss: 2.2204 -
accuracy: 0.1887 - val_loss: 2.1324 - val_accuracy: 0.2174
Epoch 9/20
100/100 [==============================] - 8s 78ms/step - loss: 2.1350 -
accuracy: 0.2600 - val_loss: 1.9813 - val_accuracy: 0.3237
Epoch 10/20
```

```
100/100 [==============================] - 8s 77ms/step - loss: 2.0357 -
accuracy: 0.3013 - val_loss: 1.9109 - val_accuracy: 0.3231
Epoch 11/20
100/100 [==============================] - 8s 77ms/step - loss: 1.8553 -
accuracy: 0.3625 - val_loss: 1.7463 - val_accuracy: 0.4234
Epoch 12/20
100/100 [==============================] - 8s 76ms/step - loss: 1.8246 -
accuracy: 0.3525 - val_loss: 1.6079 - val_accuracy: 0.4596
Epoch 13/20
100/100 [==============================] - 8s 76ms/step - loss: 1.6545 -
accuracy: 0.4525 - val_loss: 1.6414 - val_accuracy: 0.4853
Epoch 14/20
100/100 [==============================] - 8s 77ms/step - loss: 1.6571 -
accuracy: 0.4400 - val_loss: 1.4686 - val_accuracy: 0.4934
Epoch 15/20
100/100 [==============================] - 8s 76ms/step - loss: 1.5619 -
accuracy: 0.4638 - val_loss: 1.3159 - val_accuracy: 0.5653
Epoch 16/20
100/100 [==============================] - 8s 76ms/step - loss: 1.4782 -
accuracy: 0.4950 - val_loss: 1.3352 - val_accuracy: 0.5473
Epoch 17/20
100/100 [==============================] - 8s 77ms/step - loss: 1.4573 -
accuracy: 0.5063 - val_loss: 1.2197 - val_accuracy: 0.5759
Epoch 18/20
100/100 [==============================] - 8s 76ms/step - loss: 1.4210 -
accuracy: 0.5163 - val_loss: 1.2085 - val_accuracy: 0.5848
Epoch 19/20
100/100 [==============================] - 8s 77ms/step - loss: 1.3556 -
accuracy: 0.5362 - val_loss: 1.1227 - val_accuracy: 0.6133
Epoch 20/20
100/100 [==============================] - 8s 76ms/step - loss: 1.2829 -
accuracy: 0.5788 - val_loss: 1.1587 - val_accuracy: 0.6082
Training Timer: 0:03:23.764151 seconds
```

[104]:
```python
tic = time.perf_counter()
eval_model(model_cnn, history.history, train, train_y, test, test_y)
toc = time.perf_counter()

timer = str(datetime.timedelta(seconds=(toc - tic)))
print(f'Inference Timer: {timer} seconds')
```

```
              precision    recall  f1-score   support

           0       0.74      0.14      0.24       711
           1       0.62      0.93      0.74      1894
           2       0.71      0.78      0.75      1497
           3       0.46      0.49      0.47      1141
           4       0.60      0.79      0.68      1035
           5       0.60      0.67      0.63       892
           6       0.53      0.47      0.50       758
           7       0.63      0.57      0.60       789
           8       0.66      0.25      0.37       683
           9       0.74      0.17      0.27       600

    accuracy                           0.61     10000
   macro avg       0.63      0.53      0.52     10000
weighted avg       0.62      0.61      0.58     10000

Inference Timer: 0:00:04.178070 seconds
```





3.2.3 (c) Fine tune an existing model, trained on another dataset used in CAB420 (such as MNIST, KMINST or CIFAR), on the provided abridged SVHN training set.Data augmentation may also be used if you so choose.

[ ]:
```
## FROM LOTS OF VGG LIKE MODELS ##
```

[108]:
```
def conv_block(inputs, filters, spatial_dropout = 0.0, max_pool = True):

    x = layers.Conv2D(filters=filters, kernel_size=(3,3), padding='same',␣
 ↪activation='relu')(inputs)
    x = layers.Conv2D(filters=filters, kernel_size=(3,3), padding='same',␣
 ↪activation=None)(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation('relu')(x)
    if (spatial_dropout > 0.0):
        x = layers.SpatialDropout2D(spatial_dropout)(x)
    if (max_pool == True):
        x = layers.MaxPool2D(pool_size=(2, 2))(x)

    return x
```

```
def fc_block(inputs, size, dropout):
    x = layers.Dense(size, activation=None)(inputs)
    x = layers.BatchNormalization()(x)
    x = layers.Activation('relu')(x)
    if (dropout > 0.0):
        x = layers.Dropout(dropout)(x)

    return x


def vgg_net(inputs, filters, fc, spatial_dropout = 0.0, dropout = 0.0):

    x = inputs
    for idx,i in enumerate(filters):
        x = conv_block(x, i, spatial_dropout, not (idx==len(filters) - 1))

    x = layers.Flatten()(x)

    for i in fc:
        x = fc_block(x, i, dropout)

    return x
```

[109]:
```
def train_and_eval(model, x_train, y_train, x_test, y_test, filename,␣
↪batch_size, epochs):

    checkpoint = ModelCheckpoint(filename, verbose=1,␣
↪monitor='val_loss',save_best_only=True, mode='auto')

    history = model.fit(x_train, y_train,
                        batch_size=batch_size,
                        steps_per_epoch = steps_per_epoch,
                        epochs=epochs,
                        validation_data=(x_test, y_test),
                        callbacks=[checkpoint])

    model.load_weights(filename)
    model.save(filename)

    fig = plt.figure(figsize=[30, 10])
    ax = fig.add_subplot(1, 3, 1)
    plt.plot(history.history['loss'], label='loss')
    plt.plot(history.history['accuracy'], label='accuracy')
    plt.plot(history.history['val_loss'], label='val_loss')
    plt.plot(history.history['val_accuracy'], label='val_accuracy')
    ax.legend()
    ax.set_title('Training Performance')


    ax = fig.add_subplot(1, 3, 2)
    pred = model.predict(x_train);
    indexes = tf.argmax(pred, axis=1)
    cm = confusion_matrix(y_train, indexes)
    c = ConfusionMatrixDisplay(cm, display_labels=range(10))
    c.plot(ax = ax)
    ax.set_title('Training')

    ax = fig.add_subplot(1, 3, 3)
    pred = model.predict(x_test);
    indexes = tf.argmax(pred, axis=1)
    cm = confusion_matrix(y_test, indexes)
    c = ConfusionMatrixDisplay(cm, display_labels=range(10))
    c.plot(ax = ax)
    ax.set_title('Testing')
```

[110]:
```
# batch_size = 128;
# num_epochs = 10;

(x_train, y_train), (x_test, y_test) = keras.datasets.fashion_mnist.load_data()
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1).astype('float32') / 255.0
y_train = y_train.reshape(y_train.shape[0], 1)
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1).astype('float32') / 255.0
y_test = y_test.reshape(y_test.shape[0], 1)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/train-labels-idx1-ubyte.gz
32768/29515 [=================================] - 0s 0us/step
40960/29515 [=========================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/train-images-idx3-ubyte.gz
26427392/26421880 [==============================] - 0s 0us/step
26435584/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/t10k-labels-idx1-ubyte.gz
16384/5148 [==============================================================================
==========================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/t10k-images-idx3-ubyte.gz
4423680/4422102 [==============================] - 0s 0us/step
4431872/4422102 [==============================] - 0s 0us/step
```

[113]:
```
tic = time.perf_counter()
```

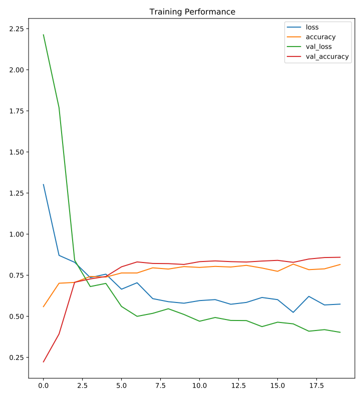
```
inputs = keras.Input(shape=(28, 28, 1, ), name='img')
x = vgg_net(inputs, [8, 16], [256], 0.2, 0.2)
output = layers.Dense(10)(x)
model_cnn = keras.Model(inputs=inputs, outputs=output, name='simple_vgg')
model_cnn.compile(loss=keras.losses.
 ↪SparseCategoricalCrossentropy(from_logits=True),
              optimizer=keras.optimizers.RMSprop(),
              metrics=['accuracy'])
train_and_eval(model_cnn, x_train, y_train, x_test, y_test,␣
 ↪'vgg_2stage_FashionMNIST_small.h5', batch_size, num_epochs)

toc = time.perf_counter()

timer = str(datetime.timedelta(seconds=(toc - tic)))
print(f'Timer: {timer} seconds')
```

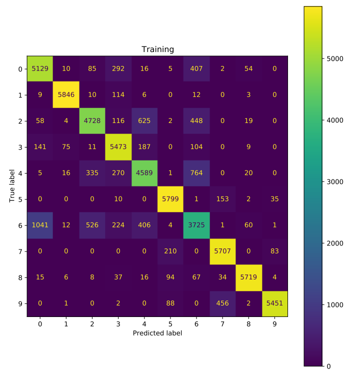
```
Epoch 1/20
100/100 [==============================] - ETA: 0s - loss: 1.3018 - accuracy:
0.5587
Epoch 1: val_loss improved from inf to 2.21273, saving model to
vgg_2stage_FashionMNIST_small.h5
100/100 [==============================] - 12s 103ms/step - loss: 1.3018 -
accuracy: 0.5587 - val_loss: 2.2127 - val_accuracy: 0.2228
Epoch 2/20
 98/100 [============================>.] - ETA: 0s - loss: 0.8742 - accuracy:
0.7015
Epoch 2: val_loss improved from 2.21273 to 1.76923, saving model to
vgg_2stage_FashionMNIST_small.h5
100/100 [==============================] - 9s 95ms/step - loss: 0.8707 -
accuracy: 0.7013 - val_loss: 1.7692 - val_accuracy: 0.3923
Epoch 3/20
 99/100 [============================>.] - ETA: 0s - loss: 0.8294 - accuracy:
0.7071
Epoch 3: val_loss improved from 1.76923 to 0.84148, saving model to
vgg_2stage_FashionMNIST_small.h5
100/100 [==============================] - 7s 71ms/step - loss: 0.8286 -
accuracy: 0.7063 - val_loss: 0.8415 - val_accuracy: 0.7071
Epoch 4/20
 99/100 [============================>.] - ETA: 0s - loss: 0.7319 - accuracy:
0.7424
Epoch 4: val_loss improved from 0.84148 to 0.68105, saving model to
vgg_2stage_FashionMNIST_small.h5
100/100 [==============================] - 7s 72ms/step - loss: 0.7354 -
accuracy: 0.7425 - val_loss: 0.6811 - val_accuracy: 0.7273
Epoch 5/20
100/100 [==============================] - ETA: 0s - loss: 0.7565 - accuracy:
0.7387
```

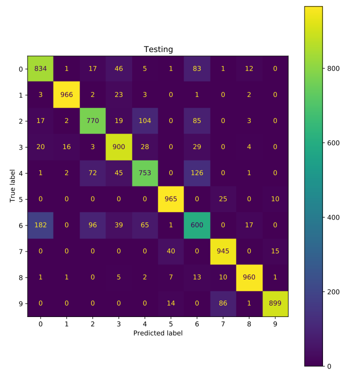
```
Epoch 5: val_loss did not improve from 0.68105
100/100 [==============================] - 7s 70ms/step - loss: 0.7565 -
accuracy: 0.7387 - val_loss: 0.7003 - val_accuracy: 0.7423
Epoch 6/20
 98/100 [============================>.] - ETA: 0s - loss: 0.6595 - accuracy:
0.7666
Epoch 6: val_loss improved from 0.68105 to 0.56000, saving model to
vgg_2stage_FashionMNIST_small.h5
100/100 [==============================] - 7s 72ms/step - loss: 0.6649 -
accuracy: 0.7638 - val_loss: 0.5600 - val_accuracy: 0.8011
Epoch 7/20
 99/100 [============================>.] - ETA: 0s - loss: 0.7060 - accuracy:
0.7626
Epoch 7: val_loss improved from 0.56000 to 0.49983, saving model to
vgg_2stage_FashionMNIST_small.h5
100/100 [==============================] - 7s 71ms/step - loss: 0.7043 -
accuracy: 0.7638 - val_loss: 0.4998 - val_accuracy: 0.8309
Epoch 8/20
 98/100 [============================>.] - ETA: 0s - loss: 0.6052 - accuracy:
0.7959
Epoch 8: val_loss did not improve from 0.49983
100/100 [==============================] - 7s 71ms/step - loss: 0.6075 -
accuracy: 0.7950 - val_loss: 0.5180 - val_accuracy: 0.8216
Epoch 9/20
 99/100 [============================>.] - ETA: 0s - loss: 0.5845 - accuracy:
0.7879
Epoch 9: val_loss did not improve from 0.49983
100/100 [==============================] - 7s 70ms/step - loss: 0.5889 -
accuracy: 0.7875 - val_loss: 0.5459 - val_accuracy: 0.8206
Epoch 10/20
 98/100 [============================>.] - ETA: 0s - loss: 0.5797 - accuracy:
0.8010
Epoch 10: val_loss did not improve from 0.49983
100/100 [==============================] - 7s 72ms/step - loss: 0.5797 -
accuracy: 0.8025 - val_loss: 0.5113 - val_accuracy: 0.8155
Epoch 11/20
 99/100 [============================>.] - ETA: 0s - loss: 0.5870 - accuracy:
0.7992
Epoch 11: val_loss improved from 0.49983 to 0.47007, saving model to
vgg_2stage_FashionMNIST_small.h5
100/100 [==============================] - 7s 72ms/step - loss: 0.5955 -
accuracy: 0.7975 - val_loss: 0.4701 - val_accuracy: 0.8324
Epoch 12/20
100/100 [==============================] - ETA: 0s - loss: 0.6016 - accuracy:
0.8037
Epoch 12: val_loss did not improve from 0.47007
100/100 [==============================] - 7s 72ms/step - loss: 0.6016 -
accuracy: 0.8037 - val_loss: 0.4928 - val_accuracy: 0.8371
```

```
Epoch 13/20
 98/100 [============================>.] - ETA: 0s - loss: 0.5755 - accuracy:
0.7997
Epoch 13: val_loss did not improve from 0.47007
100/100 [==============================] - 7s 71ms/step - loss: 0.5733 -
accuracy: 0.8000 - val_loss: 0.4749 - val_accuracy: 0.8326
Epoch 14/20
 99/100 [============================>.] - ETA: 0s - loss: 0.5858 - accuracy:
0.8106
Epoch 14: val_loss did not improve from 0.47007
100/100 [==============================] - 7s 70ms/step - loss: 0.5845 -
accuracy: 0.8100 - val_loss: 0.4740 - val_accuracy: 0.8303
Epoch 15/20
100/100 [==============================] - ETA: 0s - loss: 0.6146 - accuracy:
0.7937
Epoch 15: val_loss improved from 0.47007 to 0.43728, saving model to
vgg_2stage_FashionMNIST_small.h5
100/100 [==============================] - 7s 71ms/step - loss: 0.6146 -
accuracy: 0.7937 - val_loss: 0.4373 - val_accuracy: 0.8362
Epoch 16/20
 99/100 [============================>.] - ETA: 0s - loss: 0.6063 - accuracy:
0.7715
Epoch 16: val_loss did not improve from 0.43728
100/100 [==============================] - 7s 71ms/step - loss: 0.6014 -
accuracy: 0.7738 - val_loss: 0.4643 - val_accuracy: 0.8405
Epoch 17/20
100/100 [==============================] - ETA: 0s - loss: 0.5239 - accuracy:
0.8175
Epoch 17: val_loss did not improve from 0.43728
100/100 [==============================] - 7s 70ms/step - loss: 0.5239 -
accuracy: 0.8175 - val_loss: 0.4541 - val_accuracy: 0.8284
Epoch 18/20
 98/100 [============================>.] - ETA: 0s - loss: 0.6179 - accuracy:
0.7832
Epoch 18: val_loss improved from 0.43728 to 0.40940, saving model to
vgg_2stage_FashionMNIST_small.h5
100/100 [==============================] - 7s 71ms/step - loss: 0.6214 -
accuracy: 0.7837 - val_loss: 0.4094 - val_accuracy: 0.8486
Epoch 19/20
 99/100 [============================>.] - ETA: 0s - loss: 0.5712 - accuracy:
0.7879
Epoch 19: val_loss did not improve from 0.40940
100/100 [==============================] - 7s 71ms/step - loss: 0.5691 -
accuracy: 0.7887 - val_loss: 0.4189 - val_accuracy: 0.8575
Epoch 20/20
100/100 [==============================] - ETA: 0s - loss: 0.5740 - accuracy:
0.8150
Epoch 20: val_loss improved from 0.40940 to 0.40278, saving model to
```

```
vgg_2stage_FashionMNIST_small.h5
100/100 [==============================] - 7s 72ms/step - loss: 0.5740 -
accuracy: 0.8150 - val_loss: 0.4028 - val_accuracy: 0.8592
Timer: 0:03:41.041538 seconds
```







3.2.4 Re-Train Using Our Data

[114]:
```python
## Re-train model using our data
tic = time.perf_counter()
model_cnn.compile(loss=keras.losses.
 ↪SparseCategoricalCrossentropy(from_logits=True),
              optimizer=keras.optimizers.Adam(),
              metrics=['accuracy'])
history = model_cnn.fit(test, test_y,
                    batch_size=batch_size,
                    steps_per_epoch = steps_per_epoch,
                    epochs=num_epochs,
                    validation_data=(train, train_y))
toc = time.perf_counter()

timer = str(datetime.timedelta(seconds=(toc - tic)))
print(f'Training Timer: {timer} seconds')
```

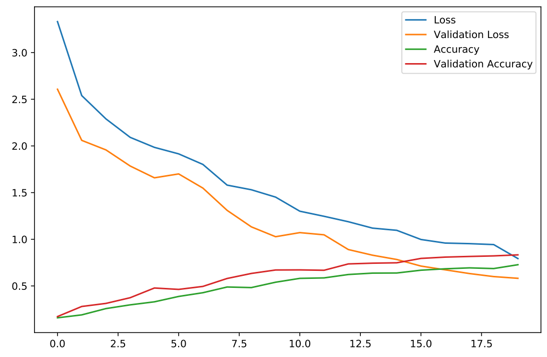
```
Epoch 1/20
100/100 [==============================] - 4s 26ms/step - loss: 3.3318 -
accuracy: 0.1587 - val_loss: 2.6074 - val_accuracy: 0.1720
Epoch 2/20
100/100 [==============================] - 2s 24ms/step - loss: 2.5394 -
accuracy: 0.1900 - val_loss: 2.0597 - val_accuracy: 0.2800
Epoch 3/20
100/100 [==============================] - 2s 24ms/step - loss: 2.2895 -
accuracy: 0.2575 - val_loss: 1.9574 - val_accuracy: 0.3130
```

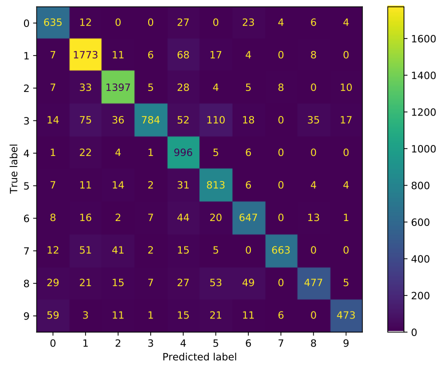
```
Epoch 4/20
100/100 [==============================] - 2s 24ms/step - loss: 2.0919 -
accuracy: 0.2975 - val_loss: 1.7848 - val_accuracy: 0.3740
Epoch 5/20
100/100 [==============================] - 2s 24ms/step - loss: 1.9846 -
accuracy: 0.3300 - val_loss: 1.6585 - val_accuracy: 0.4780
Epoch 6/20
100/100 [==============================] - 2s 24ms/step - loss: 1.9150 -
accuracy: 0.3875 - val_loss: 1.7002 - val_accuracy: 0.4630
Epoch 7/20
100/100 [==============================] - 2s 23ms/step - loss: 1.8019 -
accuracy: 0.4275 - val_loss: 1.5484 - val_accuracy: 0.4950
Epoch 8/20
100/100 [==============================] - 2s 24ms/step - loss: 1.5802 -
accuracy: 0.4888 - val_loss: 1.3097 - val_accuracy: 0.5800
Epoch 9/20
100/100 [==============================] - 2s 24ms/step - loss: 1.5305 -
accuracy: 0.4825 - val_loss: 1.1330 - val_accuracy: 0.6340
Epoch 10/20
100/100 [==============================] - 2s 23ms/step - loss: 1.4518 -
accuracy: 0.5400 - val_loss: 1.0275 - val_accuracy: 0.6710
Epoch 11/20
100/100 [==============================] - 2s 24ms/step - loss: 1.3007 -
accuracy: 0.5813 - val_loss: 1.0713 - val_accuracy: 0.6720
Epoch 12/20
100/100 [==============================] - 2s 24ms/step - loss: 1.2462 -
accuracy: 0.5863 - val_loss: 1.0478 - val_accuracy: 0.6680
Epoch 13/20
100/100 [==============================] - 2s 23ms/step - loss: 1.1881 -
accuracy: 0.6225 - val_loss: 0.8901 - val_accuracy: 0.7360
Epoch 14/20
100/100 [==============================] - 2s 24ms/step - loss: 1.1195 -
accuracy: 0.6375 - val_loss: 0.8294 - val_accuracy: 0.7440
Epoch 15/20
100/100 [==============================] - 2s 24ms/step - loss: 1.0960 -
accuracy: 0.6388 - val_loss: 0.7834 - val_accuracy: 0.7480
Epoch 16/20
100/100 [==============================] - 2s 23ms/step - loss: 0.9976 -
accuracy: 0.6687 - val_loss: 0.7129 - val_accuracy: 0.7950
Epoch 17/20
100/100 [==============================] - 2s 23ms/step - loss: 0.9594 -
accuracy: 0.6837 - val_loss: 0.6727 - val_accuracy: 0.8090
Epoch 18/20
100/100 [==============================] - 2s 24ms/step - loss: 0.9532 -
accuracy: 0.6938 - val_loss: 0.6328 - val_accuracy: 0.8160
Epoch 19/20
100/100 [==============================] - 2s 24ms/step - loss: 0.9432 -
accuracy: 0.6862 - val_loss: 0.6003 - val_accuracy: 0.8220
```

```
Epoch 20/20
100/100 [==============================] - 2s 23ms/step - loss: 0.7932 -
accuracy: 0.7275 - val_loss: 0.5818 - val_accuracy: 0.8330
Training Timer: 0:00:48.615545 seconds
```

[115]:
```
tic = time.perf_counter()
eval_model(model_cnn, history.history, train, train_y, test, test_y)
toc = time.perf_counter()

timer = str(datetime.timedelta(seconds=(toc - tic)))
print(f'Inference Timer: {timer} seconds')
```

```
              precision    recall  f1-score   support

           0       0.82      0.89      0.85       711
           1       0.88      0.94      0.91      1894
           2       0.91      0.93      0.92      1497
           3       0.96      0.69      0.80      1141
           4       0.76      0.96      0.85      1035
           5       0.78      0.91      0.84       892
           6       0.84      0.85      0.85       758
           7       0.97      0.84      0.90       789
           8       0.88      0.70      0.78       683
           9       0.92      0.79      0.85       600

    accuracy                           0.87     10000
   macro avg       0.87      0.85      0.86     10000
weighted avg       0.87      0.87      0.86     10000

Inference Timer: 0:00:03.962496 seconds
```





3.3 3. Compare the performance of the four, considering the accuracy, training time and inference time (i.e. time taken to evaluate the models), using the provided test set.

```
[ ]: # Written in report
```

---

4 Problem 2. Person Re-Identification.

Using this data, you are to: 1. Develop and a non-deep learning method using one of the dimension reduction meth- ods covered in Week 6 for person re-identification. The method should be evaluated on the test set by considering Top-1, Top-5 and Top-10 performance. A CMC (cumulative match characteristic) curve should also be provided. 2. Develop and evaluate a deep learning based method for person re-identification, using one of the methods covered in Week 7. The method should be evaluated on the test set by considering Top-1, Top-5 and Top-10 performance. A CMC (cumulative match characteristic) curve should also be provided. 3. Compare the performance of the two methods. Are there instances where the non-deep learning method works better? Comment on the respective strengths and weaknesses of the two approaches.

You have been provided with sample python code to:

- load all images, and to transform them into a vectorised representation for non-deep learning methods;
- resize images and transform the images to greyscale, which you may or may not wish to use;
- build image pairs or triplets for use with metric learning methods; plot a CMC curve given a set of distances between gallery and probe samples.

```
[2]: #
# Utility functions for CAB420, Assignment 1B, Q2
# Author: Simon Denman (s.denman@qut.edu.au)
#

import matplotlib.pyplot as plt     # for plotting
import numpy as np                  # for reshaping, array manipulation
import cv2                          # for image loading and colour conversion
import tensorflow as tf             # for bulk image resize
import os
import glob
import random


# get the subject ID from the filename. A sample filename is␣
 ↪0001_c1s1_001051_00.jpg, the first
# four characters are the subject ID
#   fn:      the filename to parse
```

```
#
#   returns: first four characters of the filename converted to an int
#
def get_subject_id_from_filename(fn):
    return int(fn[0:4])

# load the images in a directory
#   base_path: path to the data
#
#   returns:   numpy arrays of size (samples, width, height, channels), and␣
↪size (samples) for
#              images and thier labels
def load_directory(base_path):

    # find all images in the directory
    files = glob.glob(os.path.join(base_path, '*.jpg'))
    x = []
    y = []

    # loop through the images, loading them and extracting the subject ID
    for f in files:
        x.append(cv2.cvtColor(cv2.imread(f), cv2.COLOR_BGR2RGB) / 255.0)
        y.append(get_subject_id_from_filename(os.path.basename(f)))

    return np.array(x), np.array(y)

# load the data
#   base_path: path to the data, within the directory that this points to there␣
↪should be a 'Training'
#              and 'Testing' directory
#
#   returns:   loaded data
#
def load_data(base_path):

    # load training data
    train_X, train_Y = load_directory(os.path.join(base_path, 'Training'))

    # load gallery data from the test set
    gallery_X, gallery_Y = load_directory(os.path.join(base_path, 'Testing',␣
↪'Gallery'))

    # load probe data from the test set
    probe_X, probe_Y = load_directory(os.path.join(base_path, 'Testing',␣
↪'Probe'))

    return train_X, train_Y, gallery_X, gallery_Y, probe_X, probe_Y
```

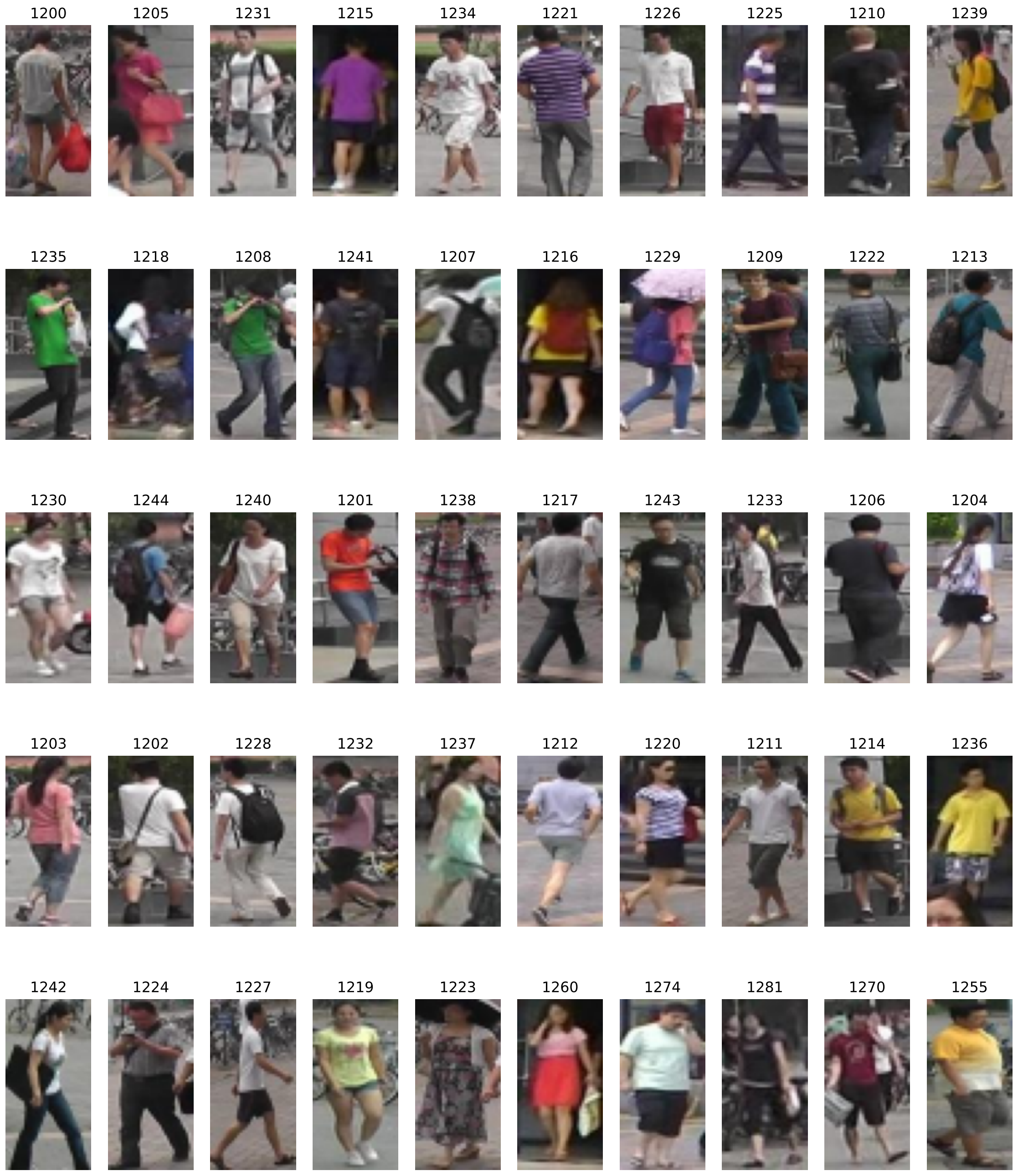
```
# Plot some images and their labels. Will plot the first 50 samples in a 10x5␣
↪grid
#  x: array of images, of shape (samples, width, height, channels)
#  y: labels of the images
#
def plot_images(x, y):
    fig = plt.figure(figsize=[15, 18])
    for i in range(50):
        ax = fig.add_subplot(5, 10, i + 1)
        ax.imshow(x[i,:], cmap=plt.get_cmap('Greys'))
        ax.set_title(y[i])
        ax.axis('off')

# vectorise an array of images, such that the shape is changed from {samples,␣
↪width, height, channels} to
# (samples, width * height * channels)
#   images: array of images to vectorise
#
#   returns: vectorised array of images
#
def vectorise(images):
    # use numpy's reshape to vectorise the data
    return np.reshape(images, [len(images), -1])

# Resize an array of images
#  images:   array of images, of shape (samples, width, height, channels)
#  new_size: tuple of the new size, (new_width, new_height)
#
#  returns:  resized array of images, (samples, new_width, new_height, channels)
#
def resize(images, new_size):
    # tensorflow has an image resize funtion that can do this in bulk
    # note the conversion back to numpy after the resize
    return tf.image.resize(images, new_size).numpy()

# Convert images to grayscale
#   images:  array of colour images to convert, of size (samples, width,␣
↪height, 3)
#
#   returns: array of converted images, of size (samples, width, height, 1)
#
def convert_to_grayscale(images):
    # storage for converted images
    gray = []
    # loop through images
    for i in range(len(images)):
        # convert each image using openCV
        gray.append(cv2.cvtColor(images[i,:], cv2.COLOR_RGB2GRAY))
    # pack converted list as an array and return
    return np.array(gray)

# Create a batch of siamese data. Pairs will be evenly balanced, such that␣
 ↪there is an
# equal number of positive and negative pairs
#   imgs:       images to use to generate data, of shape (samples, width,␣
 ↪height, channels)
#   labels:     labels for the data, of shape (samples)
#   batch_size: number of pairs to generate
#
#   returns:    image pairs and labels to indicate if the pairs are the same,␣
 ↪or different
#
def get_siamese_data(imgs, labels, batch_size):

    image_a = np.zeros((batch_size, np.shape(imgs)[1], np.shape(imgs)[2], np.
 ↪shape(imgs)[3]));
    image_b = np.zeros((batch_size, np.shape(imgs)[1], np.shape(imgs)[2], np.
 ↪shape(imgs)[3]));
    label = np.zeros(batch_size);

    for i in range(batch_size):

        if (i % 2 == 0):
            idx1 = random.randint(0, len(imgs) - 1)
            idx2 = random.randint(0, len(imgs) - 1)
            l = 1
            while (labels[idx1] != labels[idx2]):
                idx2 = random.randint(0, len(imgs) - 1)

        else:
            idx1 = random.randint(0, len(imgs) - 1)
            idx2 = random.randint(0, len(imgs) - 1)
            l = 0
            while (labels[idx1] == labels[idx2]):
                idx2 = random.randint(0, len(imgs) - 1)

        image_a[i, :, :, :] = imgs[idx1,:,:,:]
        image_b[i, :, :, :] = imgs[idx2,:,:,:]
        label[i] = l

    return [image_a, image_b], label

# Generator to continually produce batches of Siamese Data
#   imgs:       images to use to generate data, of shape (samples, width,␣
↪height, channels)
#   labels:     labels for the data, of shape (samples)
#   batch_size: number of pairs to generate
#
#   yields:     image pairs and labels to indicate if the pairs are the same,␣
↪or different
#
def pair_generator(imgs, labels, batch_size):
    while True:
        [image_a, image_b], label = get_siamese_data(imgs, labels, batch_size)
        yield [image_a, image_b], label

# Plot the first 10 pairs of a batch, good sanity check for pair generation
#  x: images in the pairs
#  y: labels of the pairs
#
def plot_pairs(x, y):
    fig = plt.figure(figsize=[25, 6])
    for i in range(10):
        ax = fig.add_subplot(2, 10, i*2 + 1)
        ax.imshow(x[0][i,:], cmap=plt.get_cmap('Greys'))
        ax.set_title('Pair ' + str(i) +'; Label: ' + str(y[i]))

        ax = fig.add_subplot(2, 10, i*2 + 2)
        ax.imshow(x[1][i,:], cmap=plt.get_cmap('Greys'))
        ax.set_title('Pair ' + str(i) +'; Label: ' + str(y[i]))

# Create a batch of triplet data.
#   imgs:       images to use to generate data, of shape (samples, width,␣
↪height, channels)
#   labels:     labels for the data, of shape (samples)
#   batch_size: number of triplets to generate
#
#   returns:    triplet of the requested batch size
#
def get_triplet_data(imgs, labels, batch_size):

    image_a = np.zeros((batch_size, np.shape(imgs)[1], np.shape(imgs)[2], np.
↪shape(imgs)[3]));
    image_b = np.zeros((batch_size, np.shape(imgs)[1], np.shape(imgs)[2], np.
↪shape(imgs)[3]));
    image_c = np.zeros((batch_size, np.shape(imgs)[1], np.shape(imgs)[2], np.
↪shape(imgs)[3]));

    for i in range(batch_size):
        idx1 = random.randint(0, len(imgs) - 1)
        idx2 = random.randint(0, len(imgs) - 1)
        idx3 = random.randint(0, len(imgs) - 1)

        while (labels[idx1] != labels[idx2]):
            idx2 = random.randint(0, len(imgs) - 1)

        while (labels[idx1] == labels[idx3]):
            idx3 = random.randint(0, len(imgs) - 1)

        image_a[i, :, :, :] = imgs[idx1,:,:,:]
        image_b[i, :, :, :] = imgs[idx2,:,:,:]
        image_c[i, :, :, :] = imgs[idx3,:,:,:]

    return [image_a, image_b, image_c]

# Generator to continually produce batches of Triplet Data
#   imgs:       images to use to generate data, of shape (samples, width,␣
 ↪height, channels)
#   labels:     labels for the data, of shape (samples)
#   batch_size: number of pairs to generate
#
#   yields:     triplet of the requested batch size
#
def triplet_generator(imgs, labels, batch_size):
    while True:
        [image_a, image_b, image_c] = get_triplet_data(imgs, labels, batch_size)
        yield [image_a, image_b, image_c], None


# Plot the first 9 triplets of a batch, good sanity check for triplet generation
#  x: images in the triplets
#
def plot_triplets(x):
    fig = plt.figure(figsize=[25, 10])
    for i in range(9):
        ax = fig.add_subplot(3, 9, i*3 + 1)
        ax.imshow(x[0][i,:], cmap=plt.get_cmap('Greys'))
        ax.set_title('Triple ' + str(i) + ': Anchor')

        ax = fig.add_subplot(3, 9, i*3 + 2)
        ax.imshow(x[1][i,:], cmap=plt.get_cmap('Greys'))
        ax.set_title('Triple ' + str(i) + ': Positive')

        ax = fig.add_subplot(3, 9, i*3 + 3)
        ax.imshow(x[2][i,:], cmap=plt.get_cmap('Greys'))
        ax.set_title('Triple ' + str(i) + ': Negative')


# Compute a ranked histogram, which can be used to generate a CMC curve. This␣
 ↪function will loop
# through all probe samples. For each probe sample it will:
#  - Compare the sample to all gallery samples to get a distance between the␣
 ↪probe sample and
#    all gallery samples. In this case it is the L1 distance
#  - Sort the gallery samples by how close they are to the probe samples
#  - Find the location of the true match
#  - Update a ranked histogram based on this
# The ranked histogram will show how many samples are matched at each rank. For␣
 ↪example,
# ranked_histogram[0] will record how many samples are matched at Rank-1.
#
# This implementation assumes that there is only one sample per ID in the␣
 ↪gallery set
#
# NOTE: L1 distance, used here, may not be appropriate for all problems.␣
 ↪Consider the nature
# of your problem and what distance measure you should use
#
#   gallery_feat: features for the gallery data, of shape (gallery_samples,␣
 ↪num_features)
#   gallery_Y:    IDs of the gallery samples, of shape (gallery_samples,)
#   probe_feat:   features for the probe data, of shape (probe_samples,␣
 ↪num_features)
#   probe_Y:      IDs of the probe samples, of shape (probe_samples,)
#   verbose:      bool to indicate if debug information shoudl be printed. Be␣
 ↪careful using this with
#                 large feature sets, and/or lots of samples
#
#   returns:      ranked histogram matching the probe samples to the gallery
#
def get_ranked_histogram_l1_distance(gallery_feat, gallery_Y, probe_feat,␣
 ↪probe_Y, verbose = False):

    # storage for ranked histogram
    # length equal to number of unique subjects in the gallery
    ranked_histogram = np.zeros(len(np.unique(gallery_Y)))

    # loop over all samples in the probe set
    for i in range(len(probe_Y)):
        # get the true ID of this sample
        true_ID = probe_Y[i]
        if verbose:
            print('Searching for ID %d' % (true_ID))

        # get the distance between the current probe and the whole gallery, L1␣
↪distance here. Note that L1
        # may not always be the best choice, so consider your distance metric␣
↪given your problem
        dist = np.linalg.norm(gallery_feat - probe_feat[i,:], axis=1, ord=1)
        if verbose:
            print(dist)

        # get the sorted order of the distances
        a = np.argsort(dist)
        # apply the order to the gallery IDs, such that the first ID in the␣
↪list is the closest, the second
        # ID is the second closest, and so on
        ranked = gallery_Y[a]
        if verbose:
            print('Ranked IDs for query:')
            print(a)

        # find the location of the True Match in the ranked list
        ranked_result = np.where(ranked == true_ID)[0][0]
        if verbose:
            print(ranked_result)

        # store the ranking result in the histogram
        ranked_histogram[ranked_result] += 1
        if verbose:
            print('')

    if verbose:
        print(ranked_histogram)

    return ranked_histogram

# Convert a ranked histogram to a CMC. This simply involves computing the␣
↪cumulative sum over the histogram
# and dividing it by the length of the histogram
#   ranked_hist: ranked histogram to convert to a CMC
#
#   returns:     CMC curve
#
def ranked_hist_to_CMC(ranked_hist):

    cmc = np.zeros(len(ranked_hist))
    for i in range(len(ranked_hist)):
        cmc[i] = np.sum(ranked_hist[:(i + 1)])

    return (cmc / len(ranked_hist))

# plot a CMC
#   cmc: cmc data to plot
#
def plot_cmc(cmc):
    fig = plt.figure(figsize=[10, 8])
    ax = fig.add_subplot(1, 1, 1)
    ax.plot(range(1, len(cmc)+1), cmc)
    ax.set_xlabel('Rank')
    ax.set_ylabel('Count')
    ax.set_ylim([0, 1.0])
    ax.set_title('CMC Curve')
```

[3]:
```
## Load the data
train_X, train_Y, gallery_X, gallery_Y, probe_X, probe_Y = load_data('/content/
 ↪drive/My Drive/Assignment 1B/CAB420_Assessment_1B_Data/Q2/Q2')

## Save original values
train_X_org = train_X
train_Y_org = train_Y
gallery_X_org = gallery_X
gallery_Y_org = gallery_Y
probe_X_org = probe_X
probe_Y_org = probe_Y
```

[4]:
```
## Resize the data
train_X_small = resize(train_X, (64, 32))
gallery_X_small = resize(gallery_X, (64, 32))
probe_X_small = resize(probe_X, (64, 32))

print(train_X_small.shape)
print(gallery_X_small.shape)
print(probe_X_small.shape)
```

```
(5933, 64, 32, 3)
(301, 64, 32, 3)
(301, 64, 32, 3)
```

[ ]:
```
# plot some resized images
plot_images(gallery_X_small, gallery_Y)
```



4.1 1. Develop and a non-deep learning method using one of the dimension reduction methods covered in Week 6 for person re-identification.

The method should be evaluated on the test set by considering Top-1, Top-5 and Top-10 performance. A CMC (cumulative match characteristic) curve should also be provided.

[17]:
```python
## Vectorise data
train_X_vec = vectorise(train_X_small)
gallery_X_vec = vectorise(gallery_X_small)
probe_X_vec = vectorise(probe_X_small)
```

4.1.1 Train PCA Network

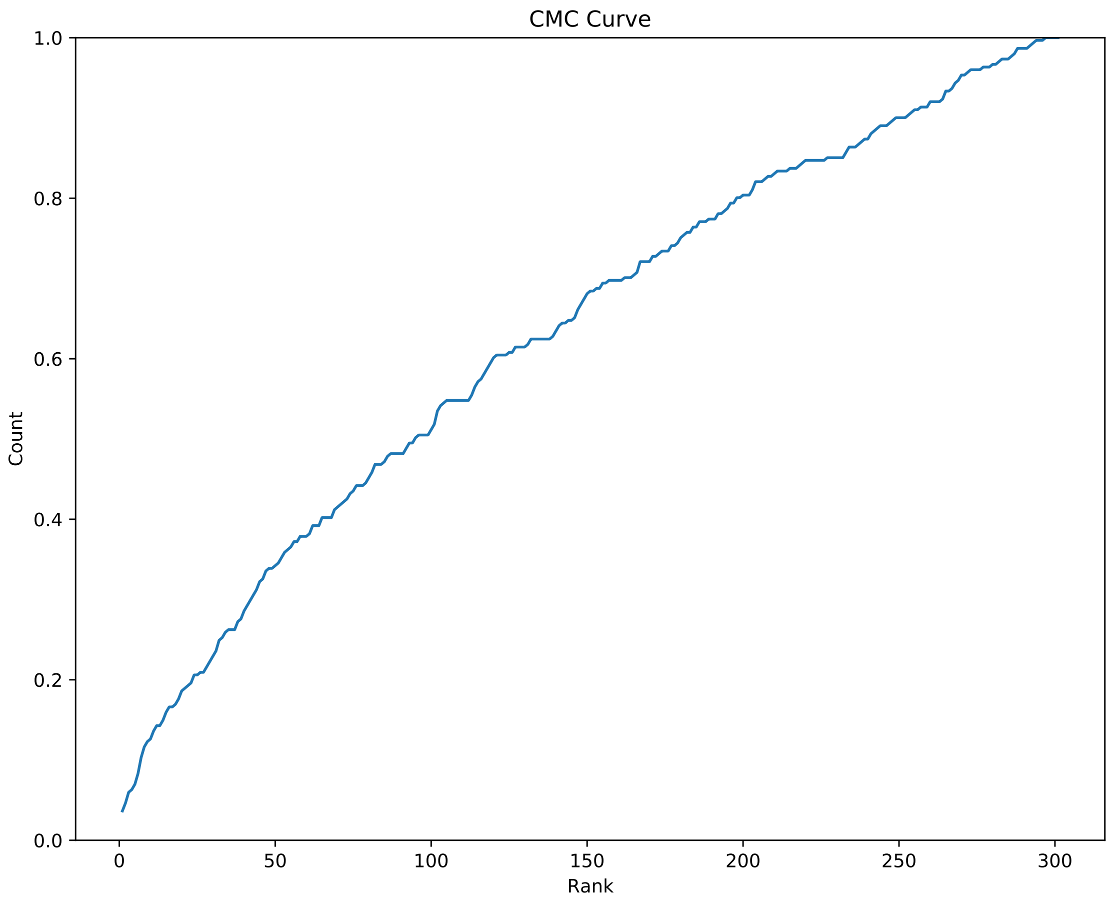
[28]:
```python
tic = time.perf_counter()

pca = decomposition.PCA()
pca.fit(train_X_vec)

transformed = pca.transform(train_X_vec)
transformed_gallery = pca.transform(gallery_X_vec)
transformed_probe = pca.transform(probe_X_vec)

cumulative_sum = np.cumsum(pca.explained_variance_ratio_, axis=0)

toc = time.perf_counter()

timer = str(datetime.timedelta(seconds=(toc - tic)))
print(f'Training Timer: {timer} seconds')
```

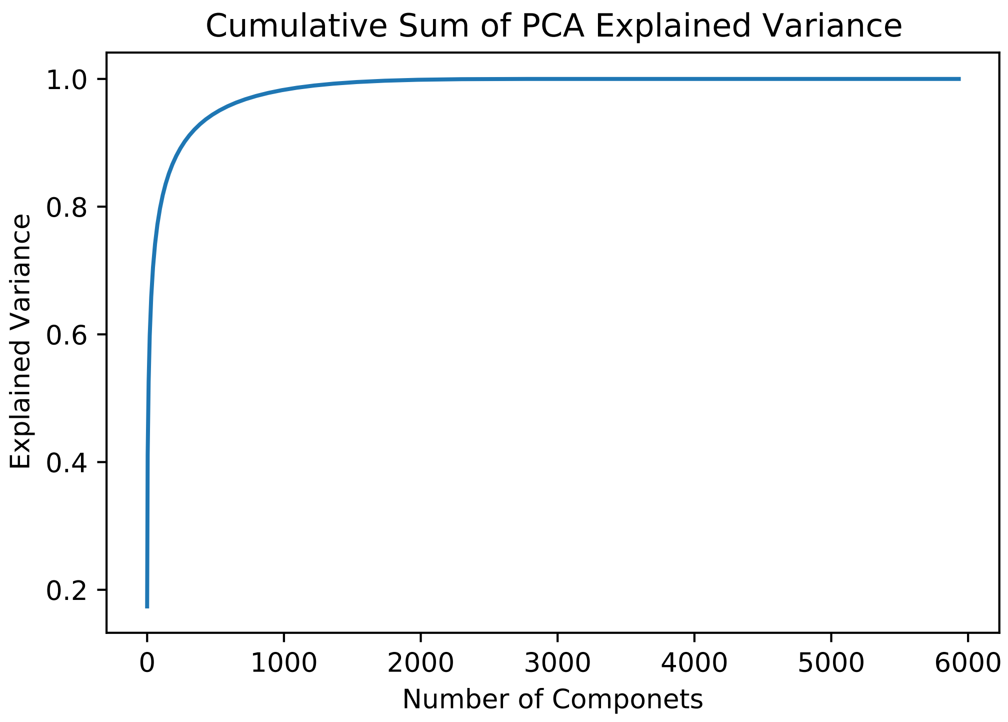
```
Training Timer: 0:01:21.954921 seconds
```

4.1.2 Plot CMC

[29]:
```python
tic = time.perf_counter()

ranked_hist = get_ranked_histogram_l1_distance(transformed_gallery, gallery_Y,␣
 ↪transformed_probe, probe_Y)
cmc = ranked_hist_to_CMC(ranked_hist)
plot_cmc(cmc)

print(pca.explained_variance_ratio_)

print('Samples in Rank-1: %d ' % (ranked_hist[0]))
print('Samples in Rank-1: %d  ' % (ranked_hist[4]))
print('Samples in Rank-1: %d ' % (ranked_hist[9]))

cumulative_sum = np.cumsum(pca.explained_variance_ratio_, axis=0)
fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax.plot(cumulative_sum)
ax.set_title('Cumulative Sum of PCA Explained Variance')
ax.set_ylabel('Explained Variance')
ax.set_xlabel('Number of Componets')

toc = time.perf_counter()

timer = str(datetime.timedelta(seconds=(toc - tic)))
print(f'Inference Timer: {timer} seconds')
```

```
[1.7394124e-01 1.1590346e-01 6.3110031e-02 … 3.6700174e-12 3.5085324e-12
 1.7095068e-13]
Samples in Rank-1: 11
Samples in Rank-1: 2
Samples in Rank-1: 1
Inference Timer: 0:00:01.189961 seconds
```





4.1.3 Train LDA Network

[30]:
```
tic = time.perf_counter()

pca_coeff_for_lda = 1024 - len(np.unique(train_X_vec))

transformed_train_for_lda = transformed[:, 0:pca_coeff_for_lda]
transformed_gallery_for_lda = transformed_gallery[:, 0:pca_coeff_for_lda]
transformed_probe_for_lda = transformed_probe[:, 0:pca_coeff_for_lda]

lda = discriminant_analysis.LinearDiscriminantAnalysis()
lda.fit(np.array(transformed_train_for_lda), np.array(train_Y))
transformed = lda.transform(transformed_train_for_lda)
print(np.shape(transformed))
transformed_gallery = lda.transform(transformed_gallery_for_lda)
transformed_probe = lda.transform(transformed_probe_for_lda)

toc = time.perf_counter()

timer = str(datetime.timedelta(seconds=(toc - tic)))
print(f'Training Timer: {timer} seconds')
```

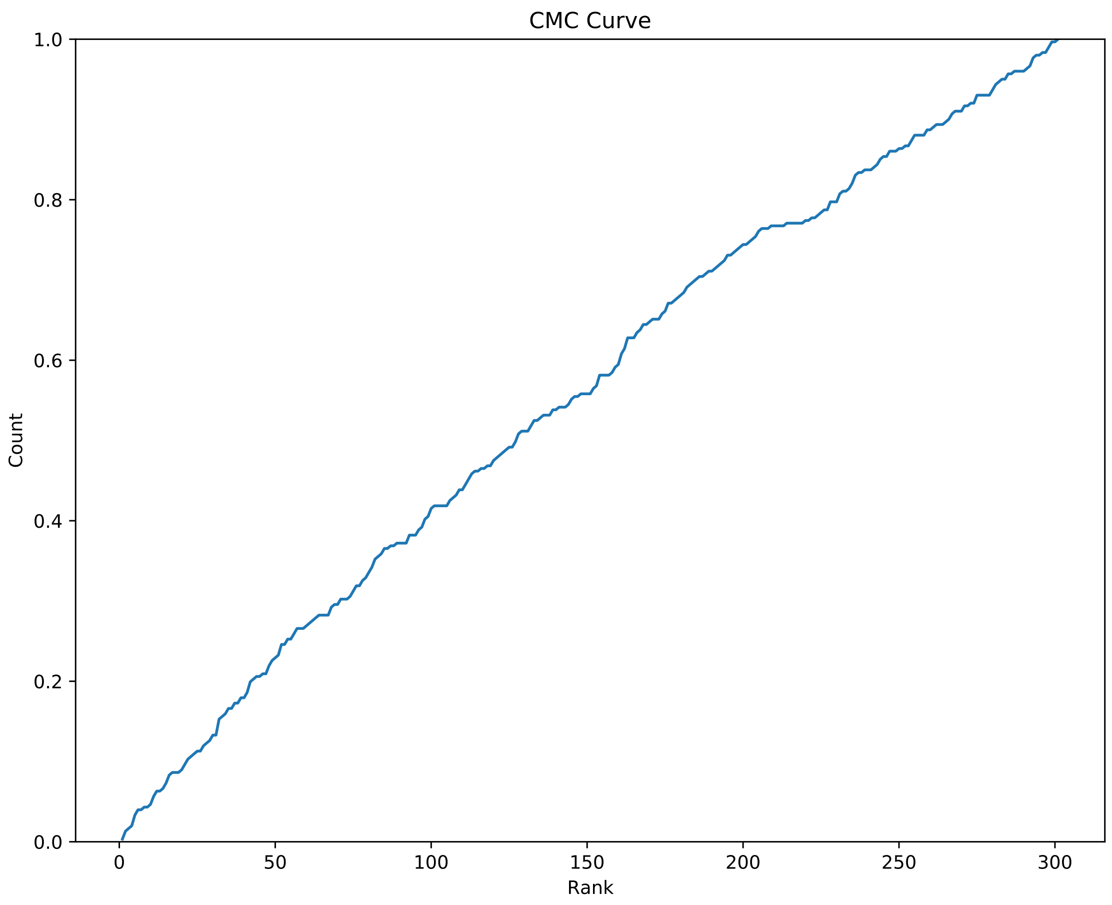
```
(5933, 299)
```

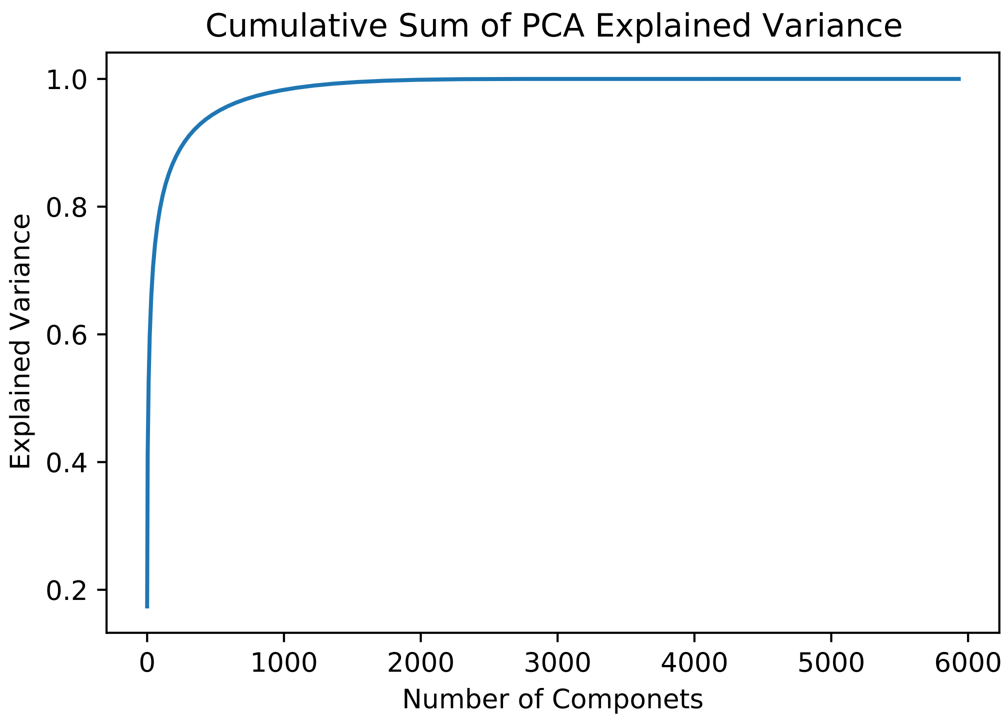
```
Training Timer: 0:01:41.507781 seconds
```

4.1.4 Plot CMC

[32]:
```python
tic = time.perf_counter()

ranked_hist = get_ranked_histogram_l1_distance(transformed_gallery, gallery_Y,␣
 ↪transformed_probe, probe_Y)
cmc = ranked_hist_to_CMC(ranked_hist)
plot_cmc(cmc)

print(pca.explained_variance_ratio_)

print('Samples in Rank-1: %d ' % (ranked_hist[0]))
print('Samples in Rank-5: %d  ' % (ranked_hist[4]))
print('Samples in Rank-10: %d ' % (ranked_hist[9]))

cumulative_sum = np.cumsum(pca.explained_variance_ratio_, axis=0)
fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax.plot(cumulative_sum)
ax.set_title('Cumulative Sum of PCA Explained Variance')
ax.set_ylabel('Explained Variance')
ax.set_xlabel('Number of Componets')

toc = time.perf_counter()

timer = str(datetime.timedelta(seconds=(toc - tic)))
print(f'Inference Timer: {timer} seconds')
```

```
[1.7394124e-01 1.1590346e-01 6.3110031e-02 … 3.6700174e-12 3.5085324e-12
 1.7095068e-13]
Samples in Rank-1: 1
Samples in Rank-5: 4
Samples in Rank-10: 1
Inference Timer: 0:00:00.198185 seconds
```


CMC Curve
Count
Rank
0.0
0.2
0.4
0.6
0.8
1.0
0
50
100
150
200
250
300



4.2 2. Develop and evaluate a deep learning based method for person re-identification,using one of the methods covered in Week 7.

The method should be evaluated on the test set by considering Top-1, Top-5 and Top-10 performance. A CMC (cumulative match characteristic) curve should also be provided.

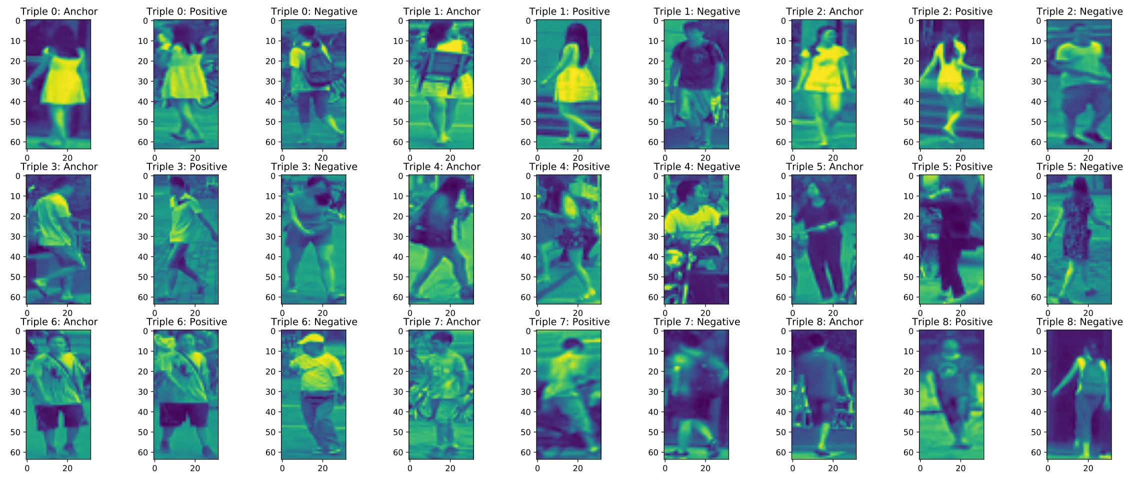
[13]:
```
print(train_X_small.shape)
```

```
(5933, 64, 32, 3)
```

4.2.1 Train Siamese Network

[14]:
```
def GetSiameseData(imgs, labels, batch_size):

    image_a = np.zeros((batch_size, np.shape(imgs)[1], np.shape(imgs)[2], np.
↪shape(imgs)[3]));
    image_b = np.zeros((batch_size, np.shape(imgs)[1], np.shape(imgs)[2], np.
↪shape(imgs)[3]));
    label = np.zeros(batch_size);
    for i in range(batch_size):

        if (i % 2 == 0):
            idx1 = random.randint(0, len(imgs) - 1)
            idx2 = random.randint(0, len(imgs) - 1)
            l = 1
            while (labels[idx1] != labels[idx2]):
                idx2 = random.randint(0, len(imgs) - 1)

        else:
            idx1 = random.randint(0, len(imgs) - 1)
            idx2 = random.randint(0, len(imgs) - 1)
            l = 0
            while (labels[idx1] == labels[idx2]):
                idx2 = random.randint(0, len(imgs) - 1)

        image_a[i, :, :, :] = imgs[idx1,:,:,:]
        image_b[i, :, :, :] = imgs[idx2,:,:,:]
        label[i] = l

    return [image_a, image_b], label

def GetTripletData(imgs, labels, batch_size):

    image_a = np.zeros((batch_size, np.shape(imgs)[1], np.shape(imgs)[2], np.
 ↪shape(imgs)[3]));
    image_b = np.zeros((batch_size, np.shape(imgs)[1], np.shape(imgs)[2], np.
 ↪shape(imgs)[3]));
    image_c = np.zeros((batch_size, np.shape(imgs)[1], np.shape(imgs)[2], np.
 ↪shape(imgs)[3]));

    for i in range(batch_size):

        idx1 = random.randint(0, len(imgs) - 1)
        idx2 = random.randint(0, len(imgs) - 1)
        idx3 = random.randint(0, len(imgs) - 1)

        while (labels[idx1] != labels[idx2]):
            idx2 = random.randint(0, len(imgs) - 1)

        while (labels[idx1] == labels[idx3]):
            idx3 = random.randint(0, len(imgs) - 1)

        image_a[i, :, :, :] = imgs[idx1,:,:,:]
        image_b[i, :, :, :] = imgs[idx2,:,:,:]
        image_c[i, :, :, :] = imgs[idx3,:,:,:]
    return [image_a, image_b, image_c]

def TripleGenerator(imgs, labels, batch_size):
    while True:
        [image_a, image_b, image_c] = GetTripletData(imgs, labels, batch_size)
        yield [image_a, image_b, image_c], None

print(train_X_small.shape)
test = TripleGenerator(train_X_small, train_Y, 9)
x, _ = next(test)

fig = plt.figure(figsize=[25, 10])
for i in range(9):
    ax = fig.add_subplot(3, 9, i*3 + 1)
    ax.imshow(x[0][i,:,:,0])
    ax.set_title('Triple ' + str(i) + ': Anchor')

    ax = fig.add_subplot(3, 9, i*3 + 2)
    ax.imshow(x[1][i,:,:,0])
    ax.set_title('Triple ' + str(i) + ': Positive')

    ax = fig.add_subplot(3, 9, i*3 + 3)
    ax.imshow(x[2][i,:,:,0])
    ax.set_title('Triple ' + str(i) + ': Negative')
```

```
(5933, 64, 32, 3)
```



[33]:
```
def conv_block(inputs, filters, spatial_dropout = 0.0, max_pool = True):

    x = layers.Conv2D(filters=filters, kernel_size=(5,5), padding='same',␣
 ↪activation='relu')(inputs)
```

```
    x = layers.Conv2D(filters=filters, kernel_size=(5,5), padding='same',␣
↪activation=None)(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation('relu')(x)
    if (spatial_dropout > 0.0):
        x = layers.SpatialDropout2D(spatial_dropout)(x)
    if (max_pool == True):
        x = layers.MaxPool2D(pool_size=(2, 2))(x)

    return x

def fc_block(inputs, size, dropout):
    x = layers.Dense(size, activation=None)(inputs)
    x = layers.BatchNormalization()(x)
    x = layers.Activation('relu')(x)
    if (dropout > 0.0):
        x = layers.Dropout(dropout)(x)

    return x

def vgg_net(inputs, filters, fc, spatial_dropout = 0.0, dropout = 0.0):

    x = inputs
    for idx,i in enumerate(filters):
        x = conv_block(x, i, spatial_dropout, not (idx==len(filters) - 1))

    x = layers.Flatten()(x)

    for i in fc:
        x = fc_block(x, i, dropout)

    return x

embedding_size = 32
dummy_input = keras.Input((64, 32, 3))
base_network = vgg_net(dummy_input, [8, 16, 32], [256], 0.2, 0)
embedding_layer = layers.Dense(embedding_size, activation=None)(base_network)
base_network = keras.Model(dummy_input, embedding_layer, name='SiameseBranch')

class TripletLossLayer(layers.Layer):
    def __init__(self, alpha, **kwargs):
        self.alpha = alpha
        super(TripletLossLayer, self).__init__(**kwargs)

    def triplet_loss(self, inputs):
        anchor, positive, negative = inputs
        anchor = K.l2_normalize(anchor, axis=1)
        positive = K.l2_normalize(positive, axis=1)
        negative = K.l2_normalize(negative, axis=1)

        p_dist = K.sum(K.square(anchor-positive), axis=-1)
        n_dist = K.sum(K.square(anchor-negative), axis=-1)
        return K.sum(K.maximum(p_dist - n_dist + self.alpha, 0), axis=0)

    def call(self, inputs):
        loss = self.triplet_loss(inputs)
        self.add_loss(loss)
        return loss

input_anchor = keras.Input((64, 32, 3), name='Anchor')
input_positive = keras.Input((64, 32, 3), name='Positive')
input_negative = keras.Input((64, 32, 3), name='Negative')

embedding_anchor = base_network(input_anchor)
embedding_positive = base_network(input_positive)
embedding_negative = base_network(input_negative)

margin = 1
loss_layer = TripletLossLayer(alpha=margin,␣
 ↪name='triplet_loss_layer')([embedding_anchor, embedding_positive,␣
 ↪embedding_negative])

tic = time.perf_counter()

triplet_network = keras.Model(inputs=[input_anchor, input_positive,␣
 ↪input_negative], outputs=loss_layer)
triplet_network.summary()

triplet_network.compile(optimizer=keras.optimizers.RMSprop())

batch_size = 128
training_gen = TripleGenerator(train_X_small, train_Y, batch_size)
triplet_test_x = GetTripletData(train_X_small, train_Y, 1000)

# triplet_network.fit(training_gen, steps_per_epoch = 60000 // batch_size,␣
 ↪epochs=10, validation_data=(triplet_test_x, None))
history = triplet_network.fit(training_gen, steps_per_epoch = 15, epochs=10,␣
 ↪validation_data=(triplet_test_x, None))

toc = time.perf_counter()

timer = str(datetime.timedelta(seconds=(toc - tic)))
print(f'Training Timer: {timer} seconds')
```

```
Model: "model_1"
________________________________________________________________________________
__________________
 Layer (type)                   Output Shape         Param #     Connected to
================================================================================
==================
 Anchor (InputLayer)            [(None, 64, 32, 3)]  0           []

 Positive (InputLayer)          [(None, 64, 32, 3)]  0           []

 Negative (InputLayer)          [(None, 64, 32, 3)]  0           []

 SiameseBranch (Functional)     (None, 32)           1108616
['Anchor[0][0]',
'Positive[0][0]',
'Negative[0][0]']

 triplet_loss_layer (TripletLos  ()                  0
['SiameseBranch[0][0]',
 sLayer)
'SiameseBranch[1][0]',
'SiameseBranch[2][0]']

================================================================================
==================
Total params: 1,108,616
Trainable params: 1,107,992
Non-trainable params: 624
________________________________________________________________________________
__________________
Epoch 1/10
15/15 [==============================] - 51s 3s/step - loss: 81.3049 - val_loss:
28.6904
Epoch 2/10
15/15 [==============================] - 45s 3s/step - loss: 52.4663 - val_loss:
21.0407
Epoch 3/10
15/15 [==============================] - 40s 3s/step - loss: 46.2893 - val_loss:
17.4199
Epoch 4/10
15/15 [==============================] - 40s 3s/step - loss: 42.2273 - val_loss:
9.9641
Epoch 5/10
15/15 [==============================] - 40s 3s/step - loss: 33.9860 - val_loss:
12.4848
Epoch 6/10
15/15 [==============================] - 40s 3s/step - loss: 34.5784 - val_loss:
7.9971
```

```
Epoch 7/10
15/15 [==============================] - 40s 3s/step - loss: 32.9180 - val_loss:
8.1702
Epoch 8/10
15/15 [==============================] - 40s 3s/step - loss: 30.6055 - val_loss:
5.9969
Epoch 9/10
15/15 [==============================] - 40s 3s/step - loss: 27.7390 - val_loss:
5.6340
Epoch 10/10
15/15 [==============================] - 40s 3s/step - loss: 28.6465 - val_loss:
7.0043
Training Timer: 0:06:58.864229 seconds
```

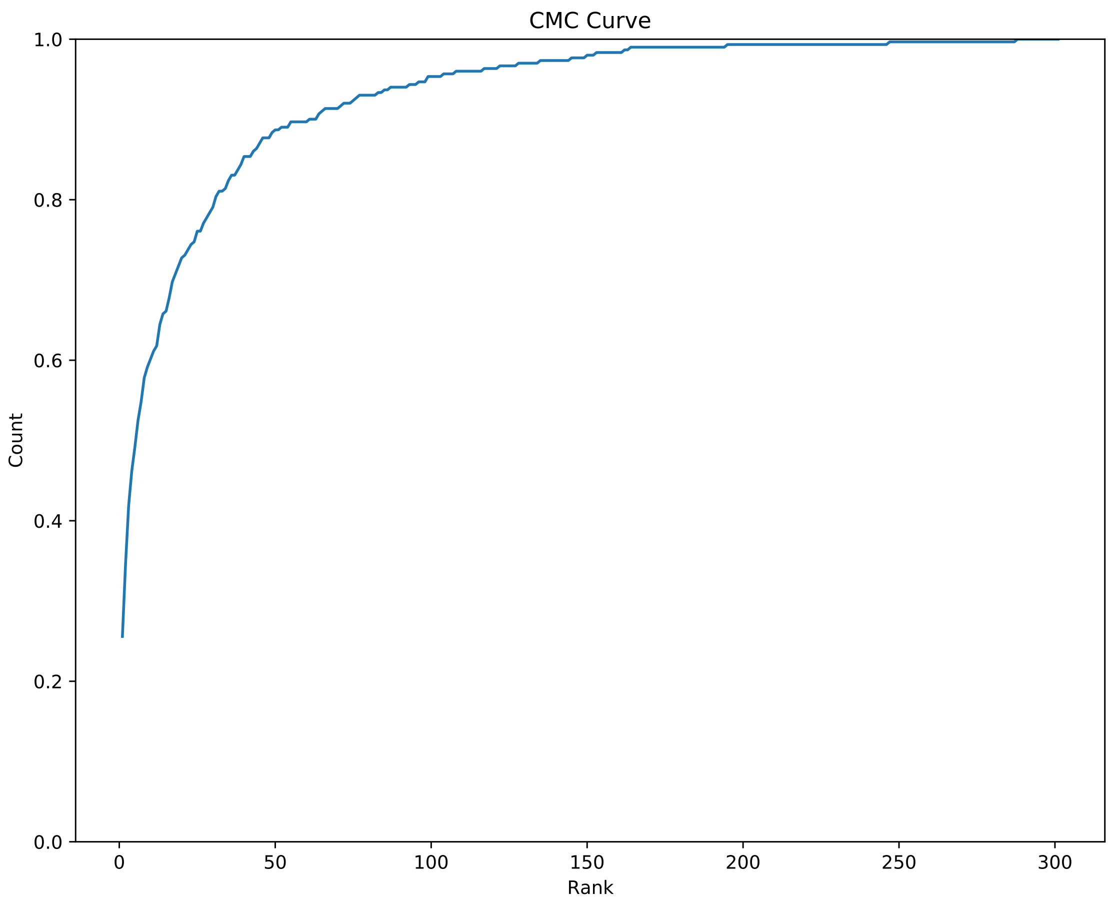
[ ]:
```
# model = triplet_network
```

4.2.2 Plot CMC

[39]:
```
embeddings_gallery = base_network.predict(gallery_X_small)
embeddings_probe = base_network.predict(probe_X_small)
```

[40]:
```
tic = time.perf_counter()

ranked_hist = get_ranked_histogram_l1_distance(embeddings_gallery, gallery_Y,␣
 ↪embeddings_probe, probe_Y)
cmc = ranked_hist_to_CMC(ranked_hist)
plot_cmc(cmc)

print(pca.explained_variance_ratio_)

print('Samples in Rank-1: %d ' % (ranked_hist[0]))
print('Samples in Rank-5: %d  ' % (ranked_hist[4]))
print('Samples in Rank-10: %d ' % (ranked_hist[9]))

toc = time.perf_counter()

timer = str(datetime.timedelta(seconds=(toc - tic)))
print(f'Inference Timer: {timer} seconds')
```

```
[1.7394124e-01 1.1590346e-01 6.3110031e-02 … 3.6700174e-12 3.5085324e-12
 1.7095068e-13]
Samples in Rank-1: 77
Samples in Rank-5: 9
Samples in Rank-10: 3
Inference Timer: 0:00:00.059419 seconds
```



4.3 3. Compare the performance of the two methods.

Are there instances where the non-deep learning method works better? Comment on the respective strengths and weaknesses of the two approaches.

```
[ ]: # Written in report
```

---

5 Export as PDF

```
[ ]: !wget -nc https://raw.githubusercontent.com/brpy/colab-pdf/master/colab_pdf.py
     from colab_pdf import colab_pdf
     colab_pdf('n10477659 Final Assignment_1B.ipynb')
```

```
--2022-05-15 03:43:29--  https://raw.githubusercontent.com/brpy/colab-
pdf/master/colab_pdf.py
Resolving raw.githubusercontent.com (raw.githubusercontent.com)…
185.199.108.133, 185.199.109.133, 185.199.110.133, …
Connecting to raw.githubusercontent.com
(raw.githubusercontent.com)|185.199.108.133|:443… connected.
HTTP request sent, awaiting response… 200 OK
Length: 1864 (1.8K) [text/plain]
Saving to: ‘colab_pdf.py’

colab_pdf.py        100%[===================>]   1.82K  --.-KB/s    in 0s

2022-05-15 03:43:30 (33.2 MB/s) - ‘colab_pdf.py’ saved [1864/1864]


WARNING: apt does not have a stable CLI interface. Use with caution in scripts.


WARNING: apt does not have a stable CLI interface. Use with caution in scripts.

Extracting templates from packages: 100%
```