Assignment 1B – Machine Learning Sem 1 2021

Nicholas Dempsey (N10224378) / Matt Kopec (n10229230)

2021

Table of Contents

[Question 1: 3](#_Toc71495382)

[Problem 2. Person Re-Identification 10](#_Toc71495383)

[1) Developing and evaluating a non-deep learning method for Person Re-Identification 10](#_Toc71495384)

[2) Develop and evaluate a deep learning-based method for person re-identification. 12](#_Toc71495385)

[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. 16](#_Toc71495386)

[Appendix 19](#_Toc71495387)

[Problem 1 19](#_Toc71495388)

[**Appendix 1: Basic** VGG No Batch normalization or Spatial Dropout: 19](#_Toc71495389)

[Appendix 2: Basic VGG w/ Spatial Dropout and Batch Normalization: 19](#_Toc71495390)

[Appendix 3: Data Augmentation: 19](#_Toc71495391)

[Appendix 4: Initial Fine tuning (No freezing): 20](#_Toc71495392)

[Appendix 5: Fine Tuning Freezing: 20](#_Toc71495393)

[Appendix 6: Pretrain Freezing and Data Augmentation: 20](#_Toc71495394)

[Appendix 7: Data Augmentation (without horizontal flip): 21](#_Toc71495395)

[Problem 2 22](#_Toc71495396)

[Data Sets 22](#_Toc71495397)

[Figure 4 - PCA 22](#_Toc71495398)

[Figure 5 – CMC Histogram 23](#_Toc71495399)

[Figure 6 – CMC Curve 24](#_Toc71495400)

[Figure 7 – Percentage CMC Curve 25](#_Toc71495401)

[Figure 8 - Siamese Pair’s 25](#_Toc71495402)

[Figure 9 – Creating Base model 27](#_Toc71495403)

[Figure 10 – Creating Siamese Network 28](#_Toc71495404)

[Figure 11 – Siamese Network Model 28](#_Toc71495405)

[Figure 12 – Training Siamese Network 29](#_Toc71495406)

[Figure 12 – Siamese Network Results 29](#_Toc71495407)

[Figure 13 – Training Siamese Network results 30](#_Toc71495408)

[Figure 14 – Scatter Plot 32](#_Toc71495409)

[Figure 15 – CMC 33](#_Toc71495410)

[Figure 16 CMC Curve Siamese Network 34](#_Toc71495411)

[Figure 17 CMC Curve Siamese Network Percentage 35](#_Toc71495412)

[Code Q1 36](#_Toc71495413)

[Code Q2 36](#_Toc71495414)

Question 1:

**Section 1.1 Network Design:**

The model architecture chosen to train was a VGG model. VGG (named after creators, Visual Geometry Group) is a convolutional neural network utilizing ‘blocks’ which contain convolution layers followed by pooling. Our model is a three staged VGG, meaning there are three convolution and pooling blocks followed by multiple dense blocks and an output layer. The main beneficiary of choosing VGG-3 architecture how simple the model is to implement and how less computationally intensive it is to train (on 3 staged models like ours). However, as more stages/blocks are added to a VGG model the computation time becomes out of scope for the purpose of this report (e.g. VGG-19 has approximately 144 million parameters).

Another downside to VGG architecture is the potential to easily overfit due to the enormous number of trainable parameters. A simple VGG-3 model may account for ~340 000 parameters while the same model utilizing spatial dropout and batch normalization can account for ~2 400 000 parameters both of which are extremely high with respect to the simple architecture.

It is also important to note that for a fair comparison between models it is required that the same model be used. Therefore, the VGG-3 architecture used is the same as the pretrained model imported for existing model fine tuning.

**Section 2.1 Augmentation:**

The training data provided for Question 1 is extremely small compared to the validation set provided, at a 1:10 ratio (1,000 training samples, 10,000 testing samples). With a small training set like this, data augmentation may be used to increase the training pool via artificially creating new samples from pre-existing data. The artificial data is created through adding preprocessing layers before the model’s learning layers which will perform a series of transformations on randomly selected samples. In this case, the transformations defined in our model are a horizontal reflection, random rotation, and random translation all of which will be reasoned in the following subsections. Utilizing the newly augmented data alongside the pre-existing training dataset **can** allow for a more robust and precise model at the cost of an increase in training time (negligible in this case) – due to the increase in training data.

**Section 2.2 Random Flip:**

The first data augmentation process is the random flip. Like all augmentations, this will choose a random sample to flip/reflect along the horizontal axis (Y axis). This will create an inverted representation, essentially creating behind view of the same image. This seems improbable since all images are taken facing front of the house; however, it may provide the model a different view to help define features for the specific class. The Random Flip was to be chosen to flip horizontal as the vertical flip will result in an overturned variant which is irrelevant to the current scenario of street view house numbers. However, it is important to note that the flip may create conflicting data where the numbers may be classified as a different ordering of the same numbers. (e.g. A flipped 12 may hold similarities to 21).

**Section 2.3 Random Rotation:**

The second augmentation chosen was a random image rotation. This will choose a random sample and rotate it positive or negative of the inputted ‘factor’ parameter. The value chosen for the ‘factor’ parameter was 0.05 which equates to a degree range of . This factor was chosen as it allows for a ~60-degree range which allows for training against SVHN (Street View House Numbers) which are either misaligned or taken from an angled camera shot. If a too large angle was chosen, then the augmented data may produce outliers or unrelated data. (e.g. 90-degrees rotation will cause the numbers to be completely rotated, unlike the rest of the dataset).

**Section 2.4 Random Translation:**

The final data augmentation method is image translation. Translation is a simple transformation method to allow for training against uncentered captures, partial captures, and provide varying number locations. While adding more variance to train against, this may allow for a more robust model by enabling the model to train against a partially captured street number. The input parameters are for the width and height factors (percent) to determine the upper and lower bound shifts of width and height. These will be kept low as the input data is only of 32x32 pixels and a too high variable can cause large variation or even possibly translate the SVHN off the 32x32 pixel space.

**Section 3.1 Pretrained Model:**

The pretrained model chosen was the small VGG-3 (3 stage) model trained on the CIFAR data, taken from the pretrained models supplied on blackboard. The primary reasoning for using this model was so it held the same network design was applied across the whole of Question 1, to allow for accurate comparison against the different approaches. The smaller VGG-3 model was chosen as the added filters from the large model seemed unnecessary for relatively small and noncomplex images. The images have been interpreted as non-complex due to the nature of numbers being much less complex and varying than other images e.g. Animals, Objects etc. The larger model could also possibly lead to an increased rate in overfitting, considering the trainable parameters will be increased.

The pretrained model trained on the CIFAR dataset was used as it holds more relevance for the current task. Although the CIFAR dataset contains various objects unrelated to the Question 1 problem space it holds more value than using a model pretrained on the MNIST fashion dataset.

In terms of tuning parameters, the first 7 layers of the model will be frozen which will cover the first convolution block (including the max pooling associated with it). This was chosen as the first convolution block will be most likely extracting universal to images (such as edges etc) and will not provide much performance increase. This will also reduce the computational load instead of retraining every layer again. Data augmentation will also be used to tune the pretrained model.

**Section 4.1 Computational Limitations:**

In terms of computational load, the chosen model architecture is decently basic and therefore does not have much of a computational load to fit and train. Although the model is basic, since it is VGG architecture it will still generate a large number of trainable parameters which can greatly increase performance time. The training parameters chosen for all models to be trained where a batch size of 64 with the number of epochs of 25. A batch size of 64 is a low value and allows for higher ability to generalize which means that the model will be lest strict on feature extraction and therefore run quicker. With a low batch size, the model is potentially open to underfitting and will be considered when creating and testing Question 1 models.

An epoch size of 25 has been tested and allows for enough training time for a model to converge or come close to it. While being a suitable amount of training time, it also makes the model runnable within a reasonable amount of time considering the computational limitations of having average processing power available.

**Section 5.0 Model Comparison:  
Section 5.1 VGG-3 Trained from Scratch Model:**

Chart, line chart, histogram

Description automatically generatedThe basic 3-stage VGG model chosen underwent two variations, initially a basic three stage VGG model and a 3-stage VGG utilizing Batch Normalization and Spatial Dropout. The first iteration model (basic VGG model) was improved upon as the threat of overfitting was prevalent as show in the below graphics.

From the diagram, it is shown that the model has a decent validation test accuracy of ~74% and an amazing training accuracy of ~98%. With a training accuracy so high, it is extremely likely that overfitting has occurred and conclusion is proven true with the supporting graphics of accuracy and loss functions. The training history of loss functions show a large disparity between the training loss and validation loss which immediately indicated overfitting or underfitting. In this case, with a validation categorical entropy loss being a lot greater than the training loss indicates overfitting. Overfitting is also prevalent in the history of accuracies where training accuracy improves but the validation accuracy plateaus and even decreases. The confusion matrix also shows a large overfitting, especially on classes 0 and 1 which hold majority of the validation data samples and high accuracies. These can be compared to class 2 which is highly classified as class 4 due to the overfitting on classes 0 and 1.

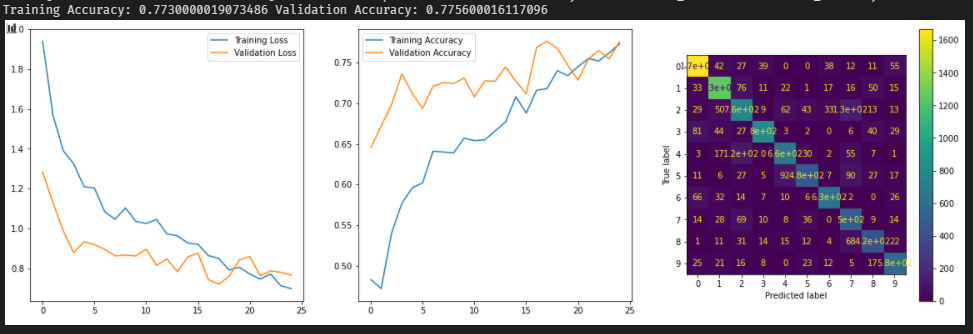
Chart, line chart

Description automatically generated­Below is the improved 3 staged Basic VGG model utilizing Spatial Dropout and Batch Normalization.­­

Immediately, the model has a slightly higher validation accuracy of ~74.5% while having a lower training accuracy of 85% which is an improvement. The training accuracy is highly performing but not to the extent of the last model which can indicate that the improved model is not overfitting. Again, through observation of the loss function and accuracies it is shown to have much less deviation between training and validation trends when compared to the Basic VGG model – a basic indication to a better fit model. The final training and validation loss function outputs show much closer and values with validation loss still being higher than the training loss. This means that overfitting is still present however extremely smaller. The reduction in overfitting can be attributed mainly to the use of Spatial Dropout which drops complete feature maps randomly subsampled from the previously layer, acting as a ‘thinner’ for the model. Observation of the accuracy graph shows the training and validation accuracy to be much closer with the training data performing better, as expected. While some classes are still being misclassified a decent amount, the overall accuracy of other classes has slightly increased, excluding class 5 and 2. One factor attributing to this is batch normalization, which normalizes the inputs of layers via rescaling and recentering acting as a small regularization.

Overall, the improved model performs better and is more robust, avoiding the issues of overfitting and will be chosen as the final iteration of our ‘train from scratch’ model.

**Section 5.2 VGG-3 Data Augmentation:**

Using the same last iteration VGG model previously analysed data augmentation was undertaken to potentially improve the robustness and possibly accuracy of the model (Augmentation methods and selection reasoning was previously evaluated in Section 2.1). The results of data augmentation will be evaluated with the following diagram holding graphs of the training history’s accuracies and loss function.

At first glance, the model has improved by ~3% from the last which is a good improvement but not drastic. A point of worry would be the training accuracy which has dropped by approximately ~10%. This is a major decrease from previous models and having a training and validation accuracy so close could indicate a possible underfitting. Underfitting is observable from the accuracy graphs, showing training accuracy lower than the validation for the majority of the training process. It is also observable from the loss function graph where having a validation loss lower than training highlights underfitting and the extend is dependent on the disparity between then. For example, at the start of the models training underfitting is high while coming close to the same by the end of training. As previously stated, by the end of training the model’s fit is almost perfect by only having a small difference between validation and training loss functions. From the trend of the graphs, underfitting may be resolved as the model continues to train if more training time (epochs) is allocated.

Although this may fix the issue of an underfitting model, it is better to explore the possible causes of this issue to eliminate underfitting before it begins. It is evident that the underfitting is a result from the data augmentation as the model being augmented did not follow the same trends during training. Although the transformation processes in the data augmentation layers were justified there is a specific transformation which may be causing this issue. This transformation is the ‘RandomFlip’, where it will flip a random image along the horizontal axis. After further analysis and evaluation of the data this may prove unnecessary and provide the model with complex representations which will never reoccur in the dataset. This holds true as the SVHN dataset contains images all from the same viewpoint (Street View) and a flip variant will not occur. An image variant which has been flipped will not occur unless an extreme outlier is present where the number have been purposefully flipped. Removal of this pre-processing layer may resolve the issue of the model struggling to find a good fit (underfit). (See Appendix 7 for removal of flip).

**Section 5.3 VGG-3 Fine Tuning:**

The final model being evaluated will be the ‘Fine tuning of existing model.’ This model holds the same architecture as the BasicVGG model used for the ‘train from scratch’ and ‘data augmentation’ sections. The parameters chosen for tuning have been discussed in the previous section (Section 3.1) and will not be further discussed. The results of fine tuning the CIFAR VGG model are in the following graph.

Chart, line chart

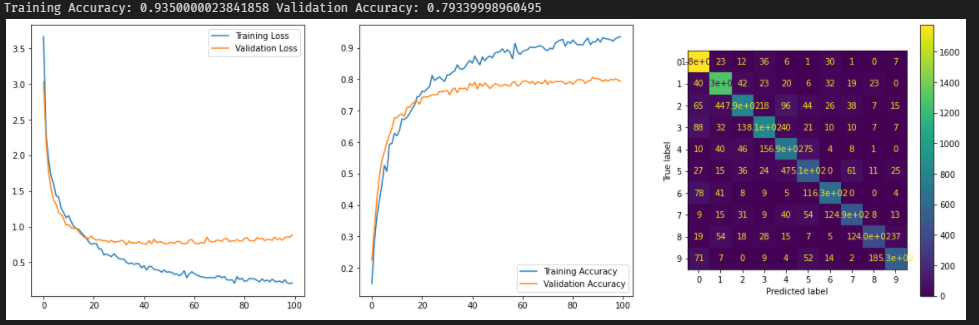
Description automatically generated

Firstly, the model is performing quite well with the validation dataset yielding approximately 82% accuracy however has a 98% accuracy on the training set. Again, this could be due to overfitting but not an extremely large amount like the initial basic VGG model. Disregarding the potential overfitting, this is the best performing model as of yet yielding approximately 4% more accuracy against the validation dataset.

To possibly eliminate any overfitting data augmentation will be applied using the existing pretrained model. The parameters for data augmentation will be the same as the augmentation model in part 2 of the question. With the data augmentation applied the results are displayed below. Chart

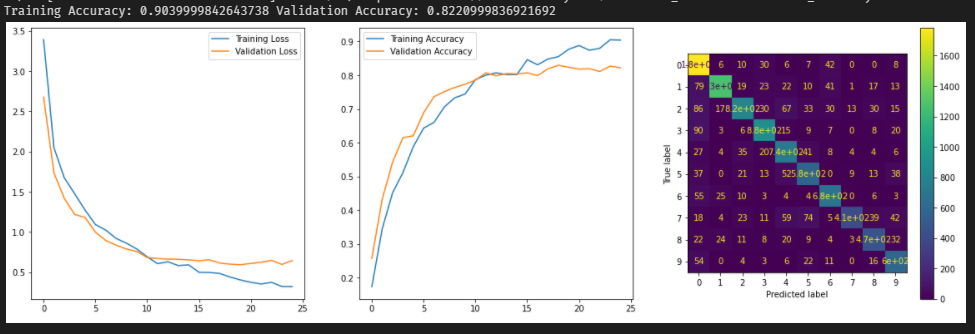
Description automatically generated

In terms of accuracy, this model is mediocre. Its performing fine but not nearly as well as the non-data augmented model. However, in terms of fitting this model excels extraordinarily well with a very slight variance between training and validation loss. This model has a good fit and perhaps with an increase in training time (via increasing epochs) it may produce worthwhile results. Below is the same model run with the four times the number of epochs (100).



An increase of approximately 5% validation accuracy is visible, which is quite a good accuracy increase but could be improved further. The misclassifications are much lower than the previous model, especially on classes 1 and 2.

A final adjustment to the model will be made, where the data augmentation parameters will be altered to remove the ‘Random Flip’ feature. As previously discussed, this provides the model with inadequate test samples as the scenario of a ‘flipped’ house number from street view is not achievable. Therefore, the same data augmented pretrained model will be ran with adjustments to the augmentation.



The validation result is much the same of the initial fine tune, however the training loss is slightly closer to the validation loss and the training accuracy has been reduced. While training accuracy being reduced isn’t always great in this case it indicates that the overfitting has been reduced and can be seen in the confusion matrix. In general, misclassifications have been reduced significantly and previously underperforming classes such as 2 and 3 have much smaller error rates. While being smaller, misclassifications are also more spread instead of being concentrated on a specific class (was class 1 and 2).

In conclusion, the final fine tune system is the fine-tuned model using the tweaked data augmentation. This is due to the high validation and training accuracy while also maintaining high performance on all classes.

**Section 5.4 Comparison Conclusion:**

In conclusion, the final version of the fine-tuned model yields the best results in terms of validation at approximately 82% while holding a training validation rate of 90.4%. Although raw performance is not a complete determinate for a model, this model also has a good fit to the data while also having a good accuracy on all classes. The misclassifications are also well-rounded, having a large spread instead of the misclassifications being concentrated around a particular class(s). This evidently means that overfitting has been minimized and all classes have been considered when classifying. Overfitting is still slightly present as seen in classification of class 0 is predicted more than other classes however, this can be seen across multiple models indicating that the dataset highly favours class 0.

# Problem 2. Person Re-Identification

## Developing and evaluating a non-deep learning method for Person Re-Identification

The reason that we chose PCA is because it reduces dimensions and reduces times. It also is using feature representations which can enhance the model’s performance. The cons of PCA is that is incapable of learning nonlinear feature representation.

Develop and evaluate a non-deep learning method 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.

When looking at this question the first problem was getting the data in a readable format for this we used Path creating a object array of all the images in that folder in python. We could of stuck with this array the whole time in python but we decided to make .mat files for the data. This saves us time in the future when running the python script over and over again and also makes the data very close to the example’s

As you can see from lines 33-48. This is our function that takes a path and a name as a input and then reads through that directory and looks for .jpg files this makes sure that there is only valid jpg photos in the data sources. At the end it then exports to a .mat as x and y.

def readImage(*path*, *name*):

    folder = Path(path).rglob('\*.jpg')

    files = [x *for* x in folder]

    ImageArray = []

    yarray = []

*for* x in files:

        img = cv2.imread(str(x))

*# img = img.flatten('C')*

        ImageArray.append(img)

        test = re.search("[0-9]{4}", str(x))

        yarray.append(int(test[0]))

    obj\_arr = numpy.zeros((2,), *dtype*=numpy.object)

    obj\_arr[0] = ImageArray

    obj\_arr[1] = numpy.transpose([yarray])

    scipy.io.savemat('./Data/Q2/matfiles' + name + '.mat',

*mdict*={'x': obj\_arr[0], 'y': obj\_arr[1]})

readImage('./Data/Q2/Testing/Gallery', 'Gallery')

readImage('./Data/Q2/Testing/Probe', 'Probe')

readImage('./Data/Q2/training', 'Training')

After this is complete, we make variables for training, galley and probe data from the mat lab files

Training = scipy.io.loadmat('./Data/Q2/matfilesTraining.mat')

Probe = scipy.io.loadmat('./Data/Q2/matfilesProbe.mat')

Gallery = scipy.io.loadmat('./Data/Q2/matfilesGallery.mat')

After loading the data into our python script, I make Org data points that I can refer back to later in the code this is for q2 so I can shape the deep learning model to how it is meant to be shaped for the deep learning algorithms. I also divide all the x’s by 255.0 this is for data normalisation.

training\_x\_Org = Training['x'].astype("float32") / 255.0

training\_x = training\_x\_Org

training\_y\_Org = Training['y'].astype("float32")

training\_y = training\_y\_Org

testing\_Probe\_x\_Org = Probe['x'].astype("float32") / 255.0

testing\_Probe\_x = testing\_Probe\_x\_Org

testing\_Probe\_y\_Org = Probe['y'].astype("float32")

testing\_Probe\_y = testing\_Probe\_y\_Org

testing\_Gallery\_x\_Org = Gallery['x'].astype("float32") / 255.0

testing\_Gallery\_x = testing\_Gallery\_x\_Org

testing\_Gallery\_y\_Org = Gallery['y'].astype("float32")

testing\_Gallery\_y = testing\_Gallery\_y\_Org

After this we create the figures for our 3 data sets (Figures 1-3). After doing this we have to flatten and reshape the arrays to be able to fit into PCA we do this in line 100-103 in the code.

Flatten\_trainx = training\_x.flatten().reshape(5933, 24576)

Flatten\_testingGalleryx = testing\_Gallery\_x.flatten().reshape(301, 24576)

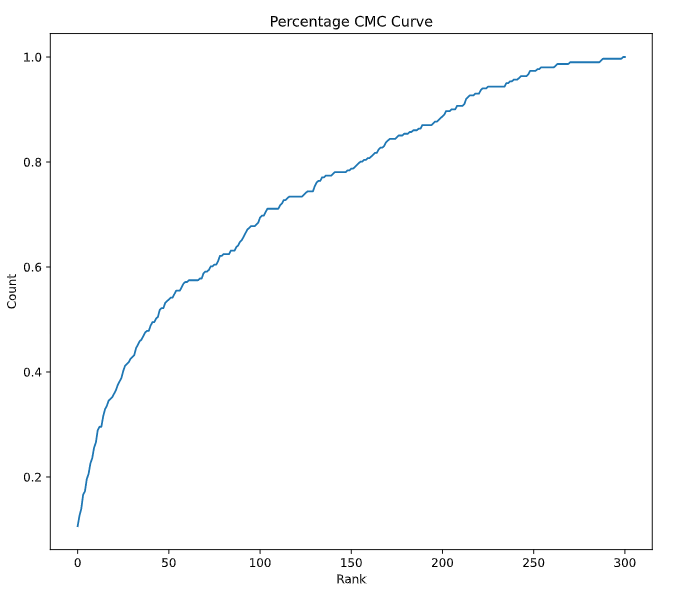
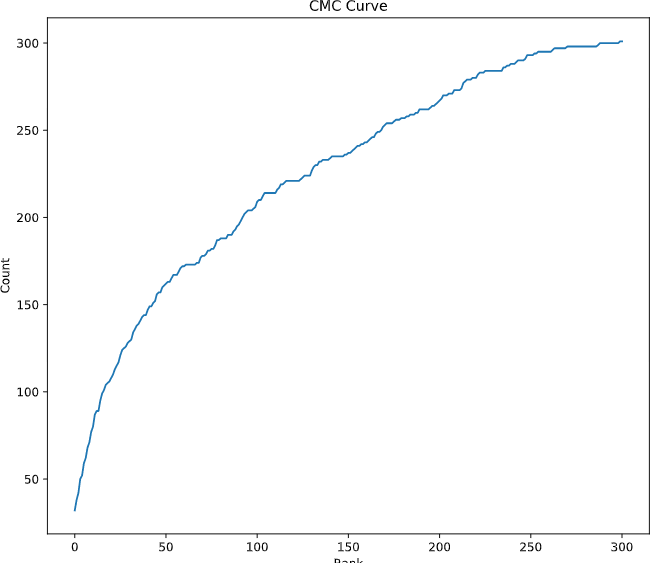
Flatten\_testingProbex = testing\_Probe\_x.flatten().reshape(301, 24576)

print("Done")

Once this has been completed, we move onto fitting the data into PCA. We transform Flatten\_trainx, Flatten\_testingGalleryx & Flatten\_testingProbex. After this has been complete we construct the data and make sure it is correct and the PCA did not change it or do anything funny you can see this in figure 4. After this we create dis variable that will measure the distance from the probe and the gallery images.

What we are doing with CMC is we are comparing each probe to all the gallery samples and we are obtaining the distance that is between them. Then we are ranking the gallery samples in order of similarity. the closes gallery samples is the best match and it continues. Once we have completed this we plot a cumulative match characteristic curve as you can see in figure 6 & 7. We can look at this graph and we can now see how often a rank of n of better is achieved.

As you can see in the graphs below it takes about 160 to get a accuracy of 80% and from their it gradual goes up to 99% when it reaches 300. As you can see with the non deep learning CMC the curve goes up gradually and has no major points were it improves allot. This curve only gets to around 90% accuracy around 225 img in.



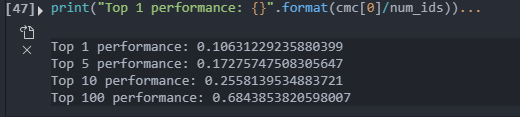
As you can see below we have printed of the top 1, 5, 10 and 100 CMC graph performance

print("Top 1 performance: {}".format(cmc[0]/num\_ids))

print("Top 5 performance: {}".format(cmc[4]/num\_ids))

print("Top 10 performance: {}".format(cmc[9]/num\_ids))

print("Top 100 performance: {}".format(cmc[99]/num\_ids))



## Develop and evaluate a deep learning-based method for person re-identification.

We decided to use the Siamese Networks for the deep learning model. The reason that we chose this model is because it compares two things. They make and contain two identical branches with the exact same weights as the inputs that have been passed into it. The pros of the Siamese network is that it gives a more robust to class imbalance. Nice to an ensemble with the best classifier, learning from sematic similarity. The downside for the Siamese networks is that it requires more training time than normal networks and that is does not output probabilities. We decided to go with this network because it the network has allot of pros and not allot of cons. Plus I have a decent computer and the network takes only around 10 min to train.

We first shape the data to fit into the Siamese network as seen below.

deepLearning\_x\_train = training\_x\_Org.reshape(training\_x\_Org.shape[0], 128, 64, 3)

deepLearning\_y\_train = training\_y\_Org.reshape(training\_y\_Org.shape[0], 1)

deepLearning\_x\_test = testing\_Gallery\_x\_Org.reshape(testing\_Gallery\_x\_Org.shape[0], 128, 64, 3)

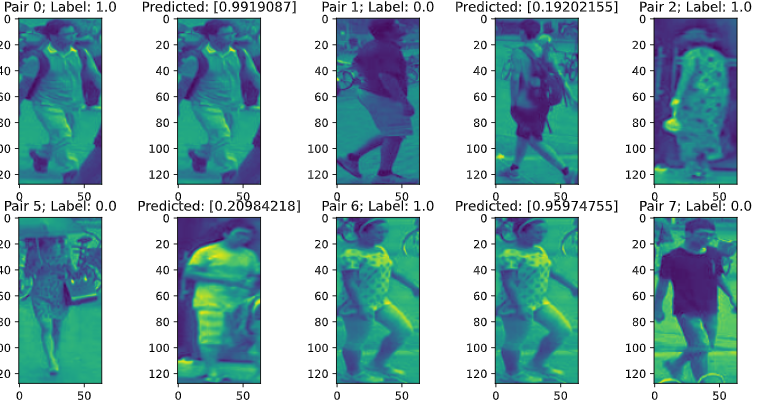
deepLearning\_y\_test = testing\_Gallery\_y\_Org.reshape(testing\_Gallery\_y\_Org.shape[0], 1)

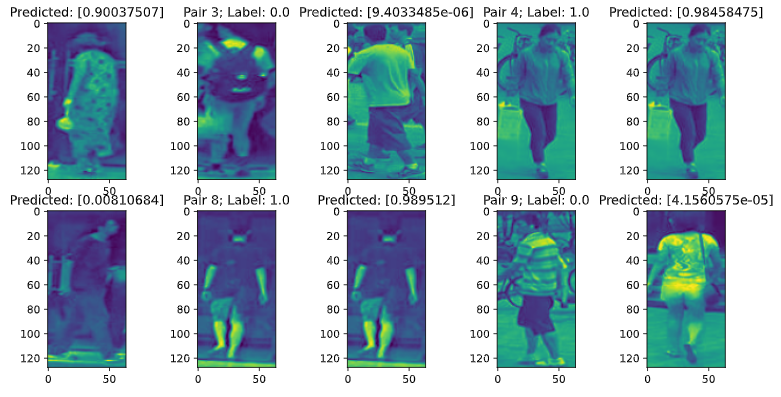
Then what we do is we create pairs for the Siamese network. After this we create a base network. (Figure 9) Then we create the 2 inputs for the Siamese network. This is for the network to compare the gallery and the probe images for validation & testing. After this we layer the 2 inputs embending\_a and embending\_B.

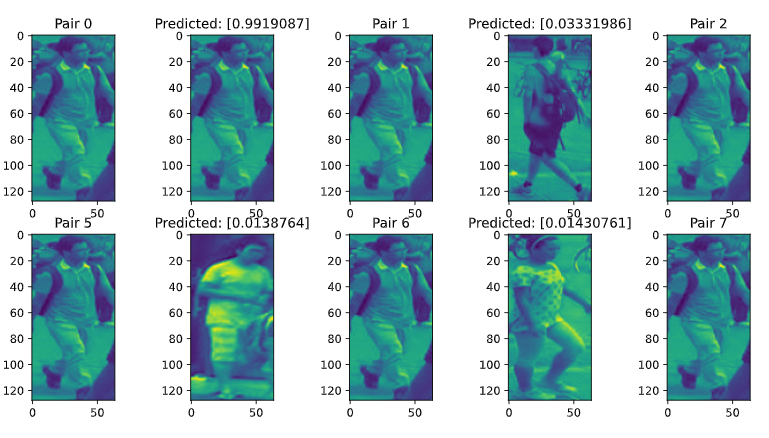
We create the model for the Simese Network with both the input layers and the output layer. As can be seen in figure 10 and for a visual model you can look at figure 11. The next thing we do is train the model this takes around 10 minutes each time we train the model with 10 epoch. As you can see in Figure 12. This is training the model with the training data then validating it with the probe images and gallery images. Our final results are:

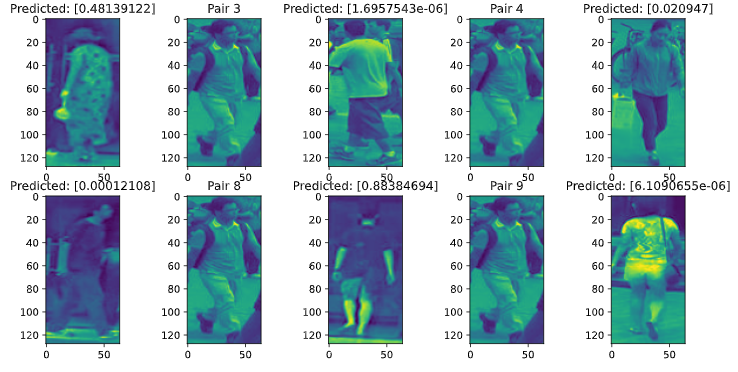
Loss: 0.1848, Accuracy: 0.9309, Val Loss: 0.1668. Val Accuracy: 0.9308

After we are done training our model. We print out the first 10 results and plot them on a graph this can be seen in figure 12. As you can see below you have the pair number the Label this means is if the images matches or the image does not match, and the prediction based of the network. As you can see from the figures below our model predicts around 8/10 of the images that was printed









We created a CMC curve for the Siamease network so that we could compare it to the non deep learning approach. This is done in lines 359-390

embeddingGallery = base\_network.predict(testing\_Gallery\_x\_Org)

embeddingProbe = base\_network.predict(testing\_Probe\_x\_Org)

dist = distance.cdist(embeddingProbe,

                      embeddingGallery, 'euclidean')

num\_ids = len(numpy.unique(testing\_Gallery\_y))

ranked\_histogram = numpy.zeros(num\_ids)

*for* i in range(len(testing\_Gallery\_y)):

*#print("True: {}".format(testing\_Gallery\_y[i]))*

    order = numpy.argsort(dist[i])

    ranked = testing\_Gallery\_y[order]

    ranked\_result = numpy.where(ranked == testing\_Probe\_y[i])[0][0]

    ranked\_histogram[ranked\_result] += 1

*# print(ranked\_result)*

print(ranked\_histogram)

plt.plot(ranked\_histogram)

cmc = numpy.zeros(num\_ids)

*for* i in range(num\_ids):

    cmc[i] = numpy.sum(ranked\_histogram[:(i + 1)])

fig = plt.figure(*figsize*=[20, 8])

ax = fig.add\_subplot(1, 2, 1)

ax.plot(cmc)

ax.set\_xlabel('Rank')

ax.set\_ylabel('Count')

ax.set\_title('CMC Curve')

ax = fig.add\_subplot(1, 2, 2)

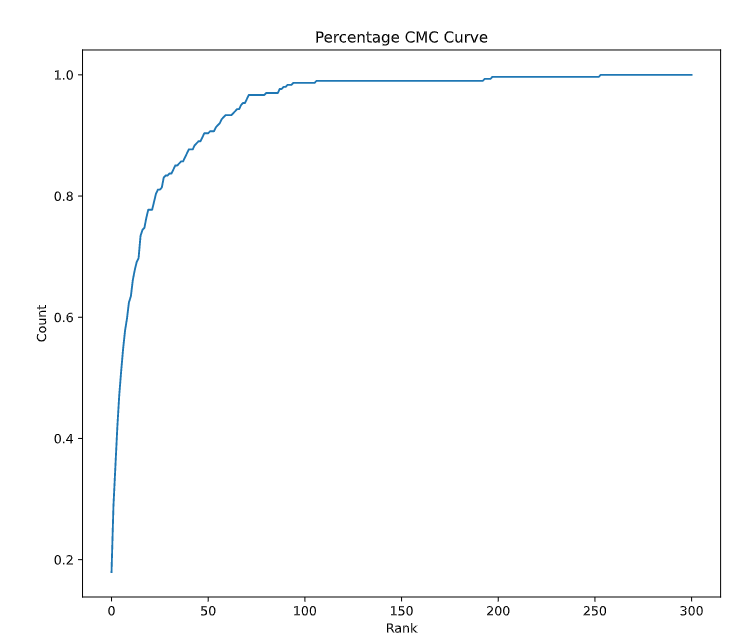
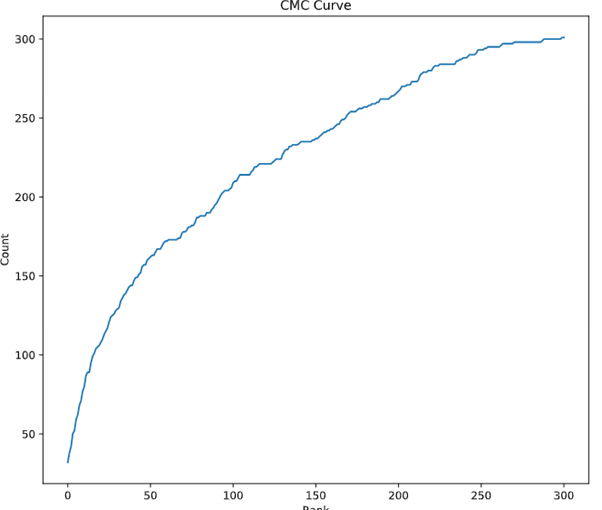
ax.plot(cmc/num\_ids)

ax.set\_xlabel('Rank')

ax.set\_ylabel('Count')

ax.set\_title('Percentage CMC Curve')

As you can see below from our CMC curves for the Siamese Network it gets to 80% accuracy around 25 images in. As you can see this CMC curve has a very steep incline at the start and when it gets to around 90% it goes up only a little bit at the end.



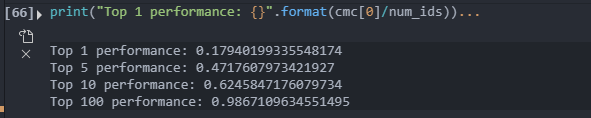
What this does is create a histogram of the CMC curve and then it creates the CMC curves which u can see in figures 15-17. Next we can print of the top 1, 5, 10, 100

print("Top 1 performance: {}".format(cmc[0]/num\_ids))

print("Top 5 performance: {}".format(cmc[4]/num\_ids))

print("Top 10 performance: {}".format(cmc[9]/num\_ids))

print("Top 100 performance: {}".format(cmc[99]/num\_ids))



## 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.

PCA vs Siamese Network

As we can see from our to CMC Curves one for PCA and the other for the Siamese Network. They are completely different. If you are looking at the percentage CMC Curve the PCA model gradual goes up and has no dramatic increase of it accuracy. This gives us a nice curve across all the data. It takes about 150 images to get to 80% accuracy.

With the Siamese Network you can see that the graph only takes around 25 images that have a probability that is lower than 80%. Then it has a little curve. When the accuracy gets to 90%. It does not improve that much more over the others this is because getting 100% accuracy is very hard to achieve / is impossible with current technology. U will also notice that the CMC curve increase allot at the very start to get.

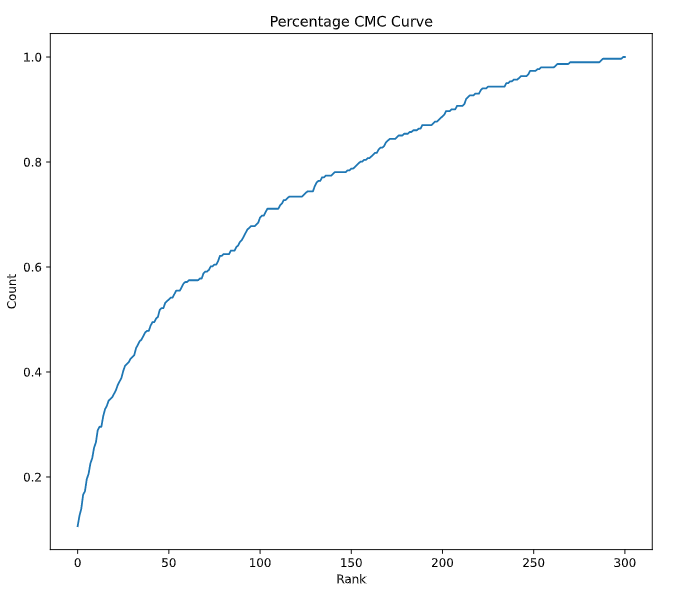
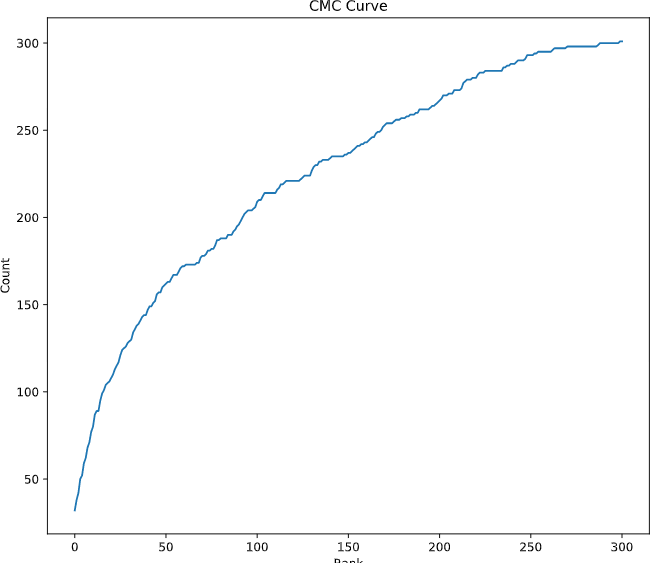
There are no instances in our model where the non-deep learning methods is working better then the deep learning method. You could make the argument that the deep learning network could be learning to fast, and it might get data wrong by assuming more things quicker. The other thing that could happen is the Siamese Network you might get overfitting if you train it too much or too little.

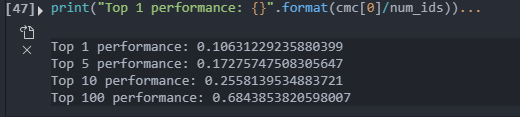
As you can see the top 1 performance of the PCA CMC is 0.1063 and the top 1 performance for the Siamese network is 0.1794. Going on to top 5 performance PCA CMC is 0.10631 and the top 5 performance for the Siamese Network is 0.47176. Going on to top 10 performance PCA CMC is 0.25581 and the top 5 performance for the Siamese Network is 0.62458. Going on to top 100 performance PCA CMC is 0.684385 and the top 5 performance for the Siamese Network is 0.98671.

As u can see in all the performance results the Siamese Network is preforming better than the PCA CMC.

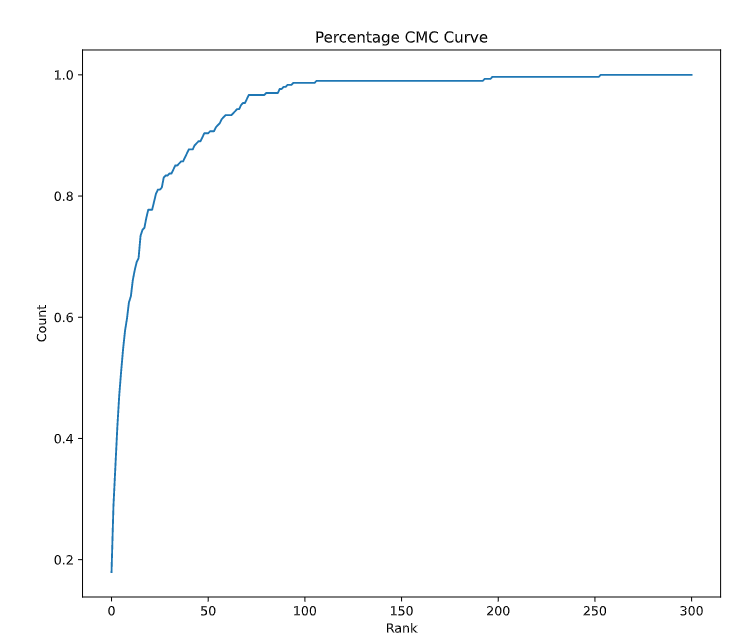
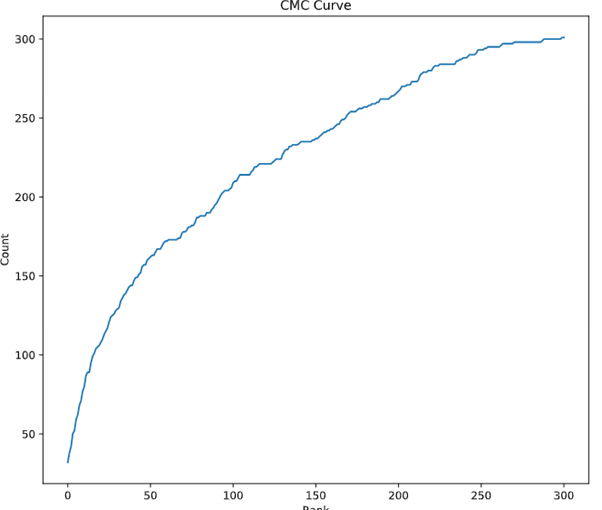
The strengths on the PCA is that it is harder to over fit and is faster to run than the Siamese Network. The strength of Siamese Network is that it is more accurate then PCA and gets betters results, but this causes it to take a considerably longer than time from PCA around 8 more minutes. To Train and validate the model. One of the benefits of Siamese network compares to things. It takes the compared data, and it will train the model of that and make it so that the make the network predict if something is a match and when something is not a match and gives data back on that. Another downside of Siamese network is that has its limitations on what it can say to things are the same or not. For some applications this is enough but others it is not, and things are too similar. Also, it does not offer any information in any realisable sort of way. We could improve this by doing Constructive loss or by doing Triplet loss but this would cause the model to take longer to train.

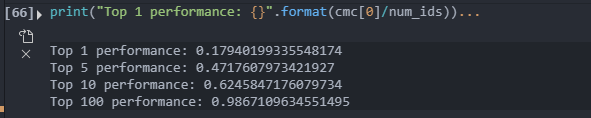
PCA CMC Curve





Siamese Network CMC Curve

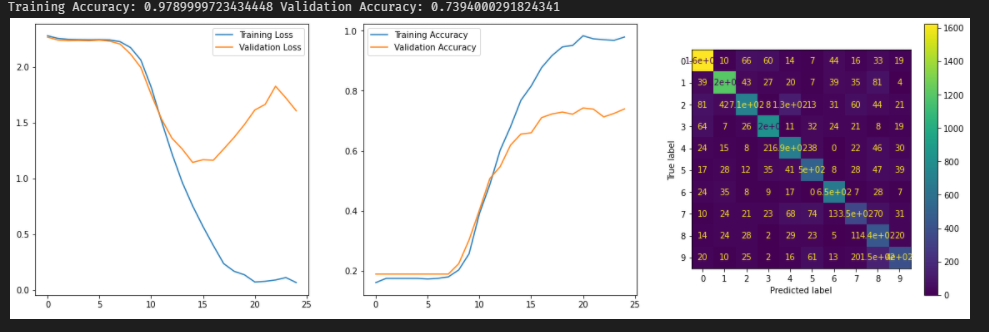




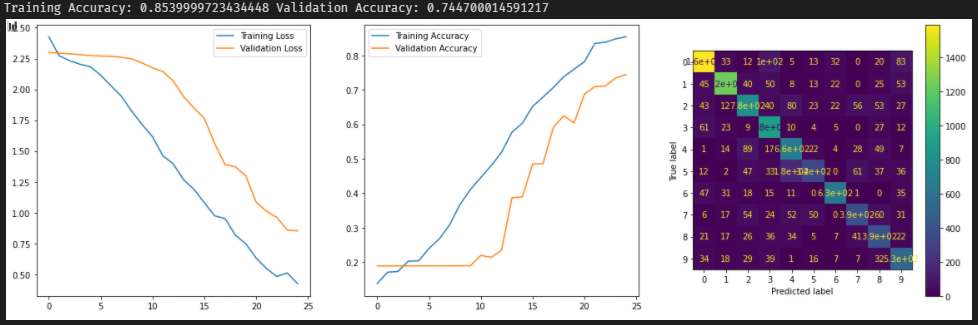
# Appendix

## Problem 1

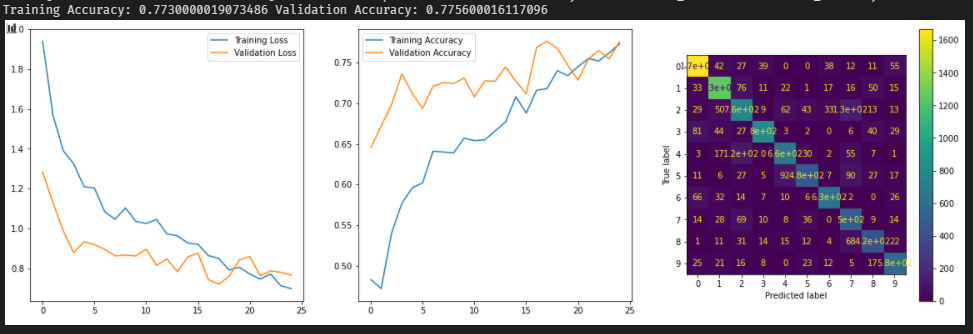
### **Appendix 1: Basic** VGG No Batch normalization or Spatial Dropout:



### Appendix 2: Basic VGG w/ Spatial Dropout and Batch Normalization:

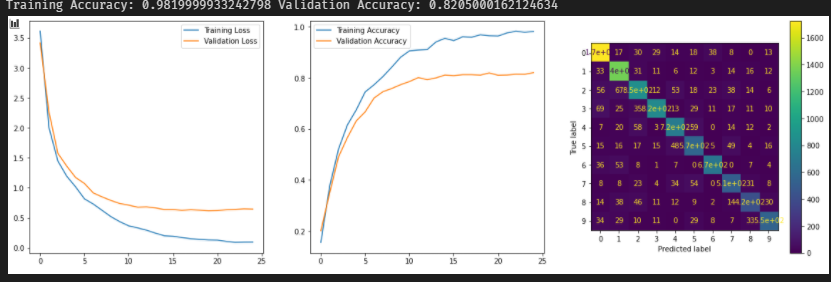


### Appendix 3: Data Augmentation:

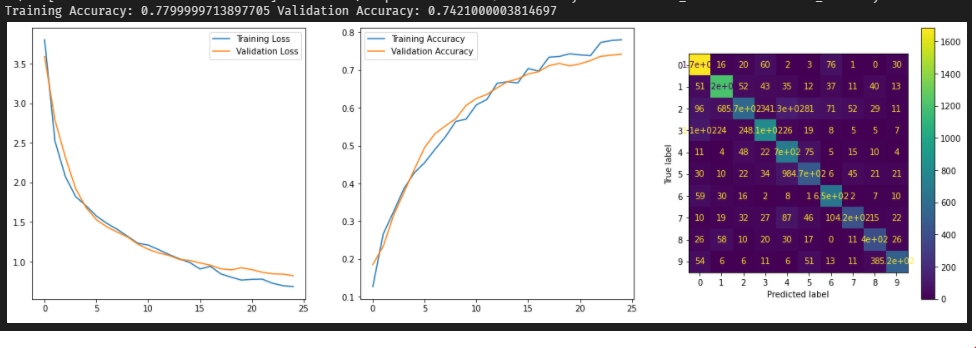


### Appendix 4: Initial Fine tuning (No freezing):

### Appendix 5: Fine Tuning Freezing:



### Appendix 6: Pretrain Freezing and Data Augmentation:

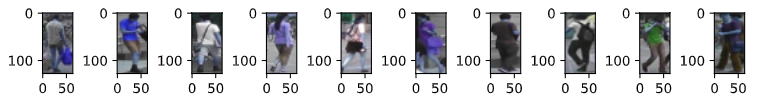


### Appendix 7: Data Augmentation (without horizontal flip):

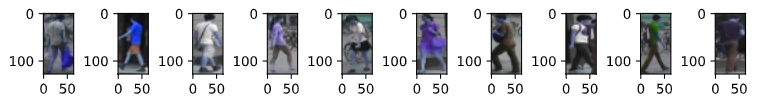
## Problem 2

### Data Sets

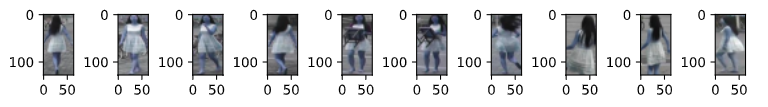
#### Figure 1 - Testing Gallery



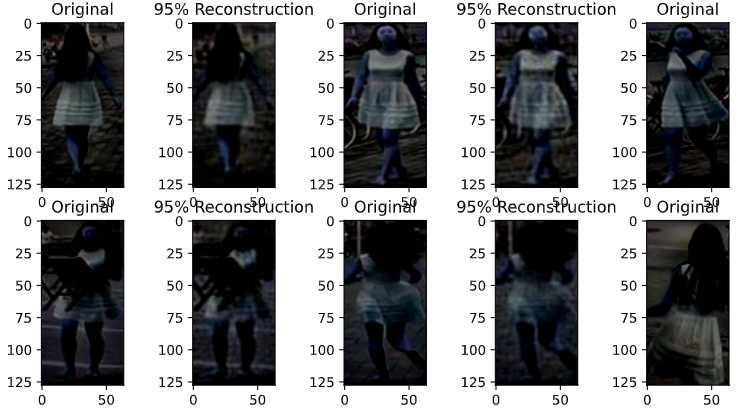
#### Figure 2 - Testing Probe

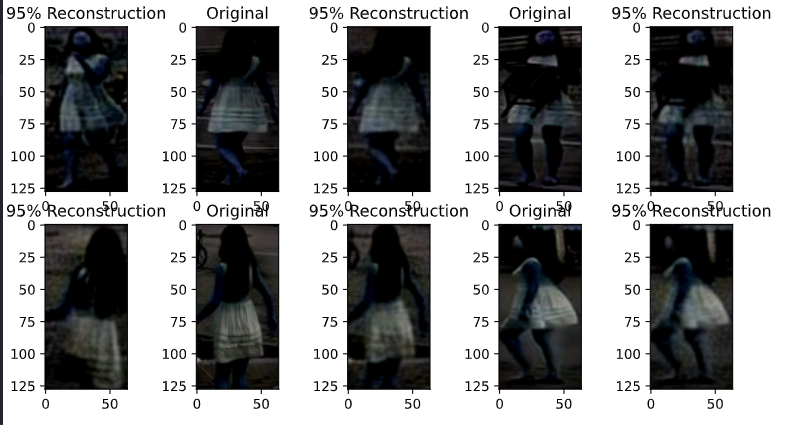


#### Figure 3 - Training

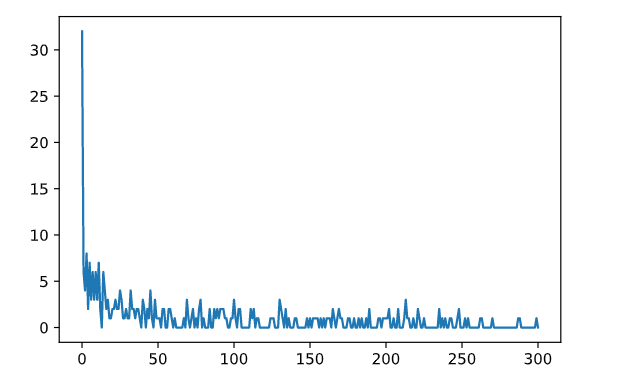


### Figure 4 - PCA

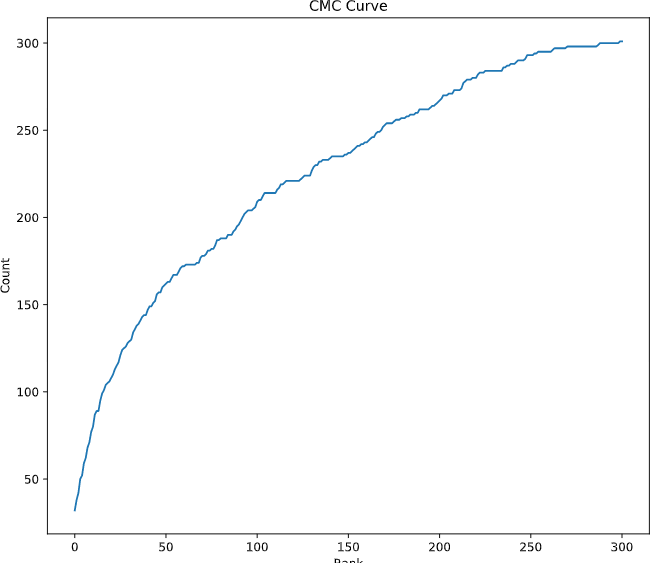




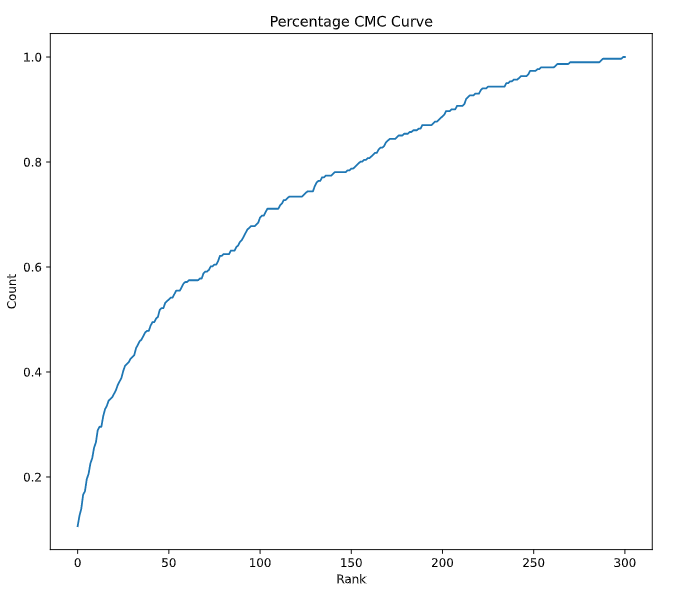
### Figure 5 – CMC Histogram



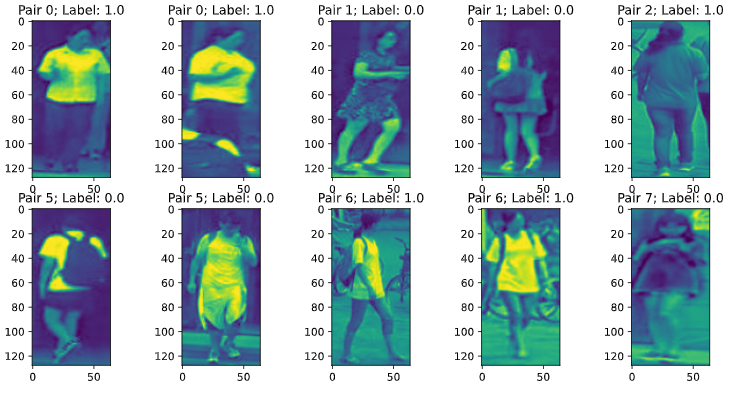
### Figure 6 – CMC Curve

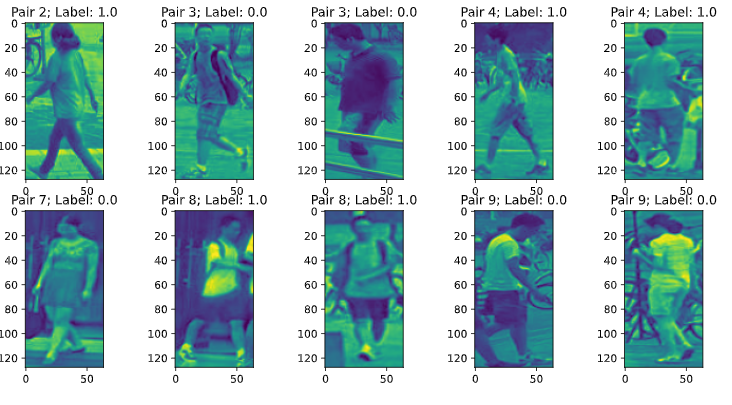


### Figure 7 – Percentage CMC Curve

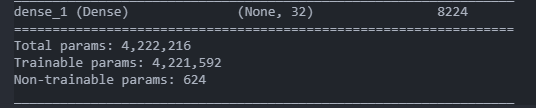
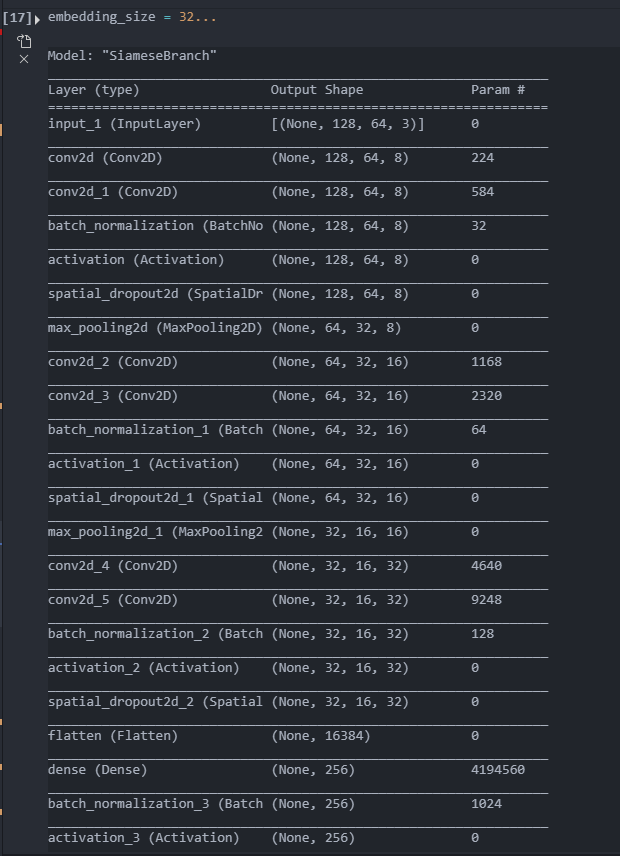


### Figure 8 - Siamese Pair’s

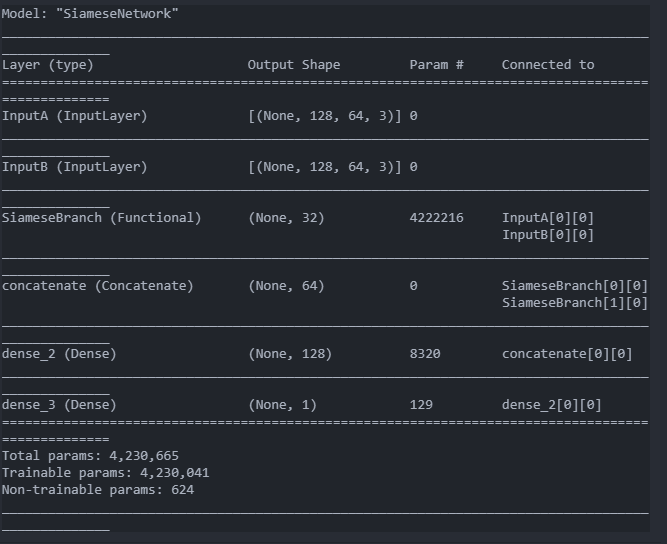




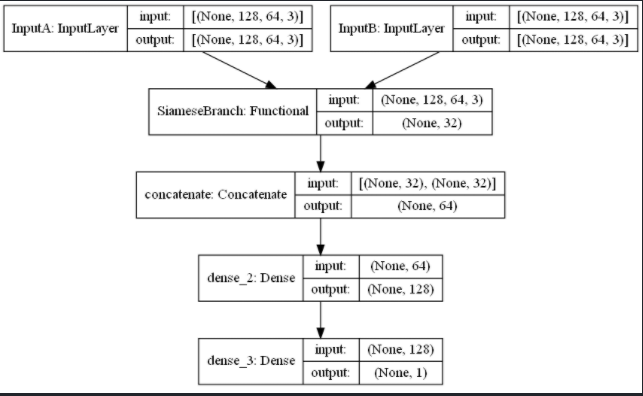
### Figure 9 – Creating Base model



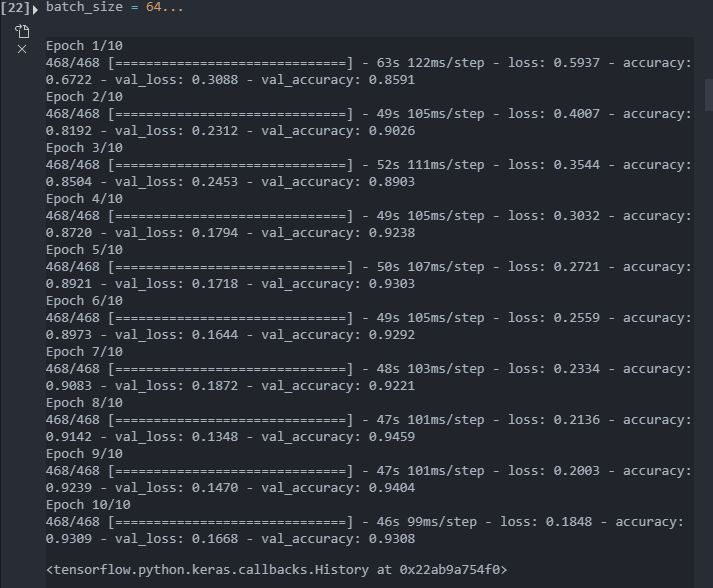
### Figure 10 – Creating Siamese Network



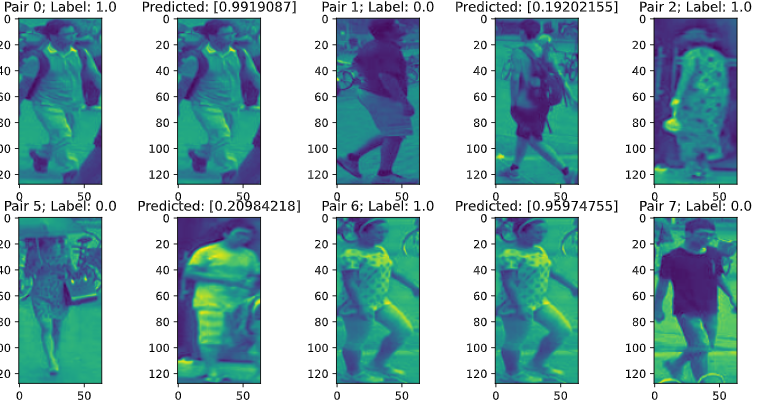
### Figure 11 – Siamese Network Model

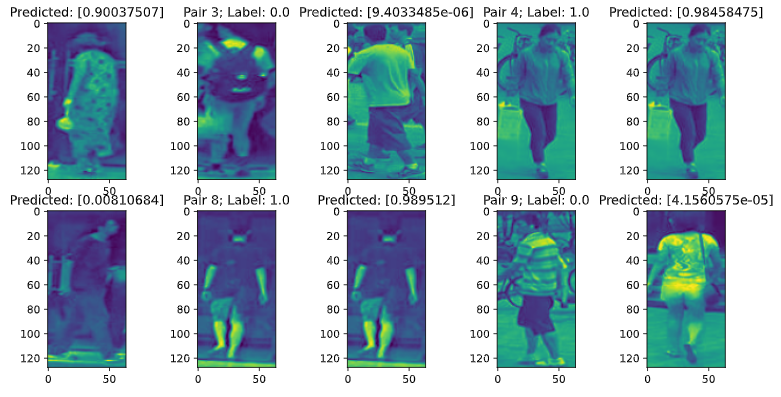


### Figure 12 – Training Siamese Network

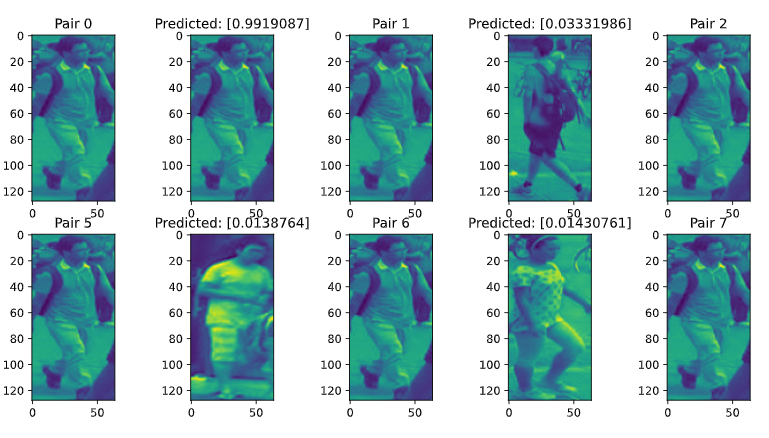


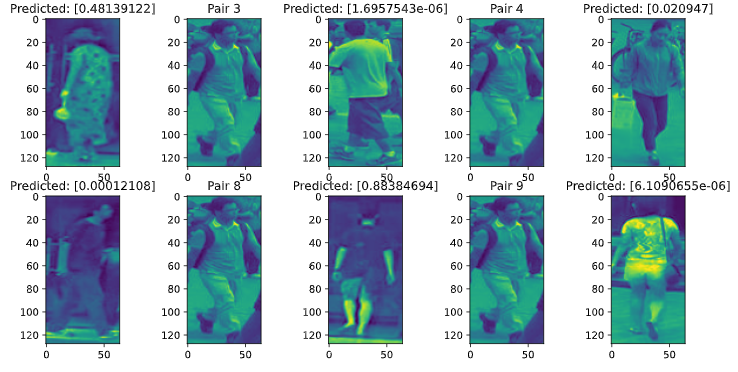
### Figure 12 – Siamese Network Results



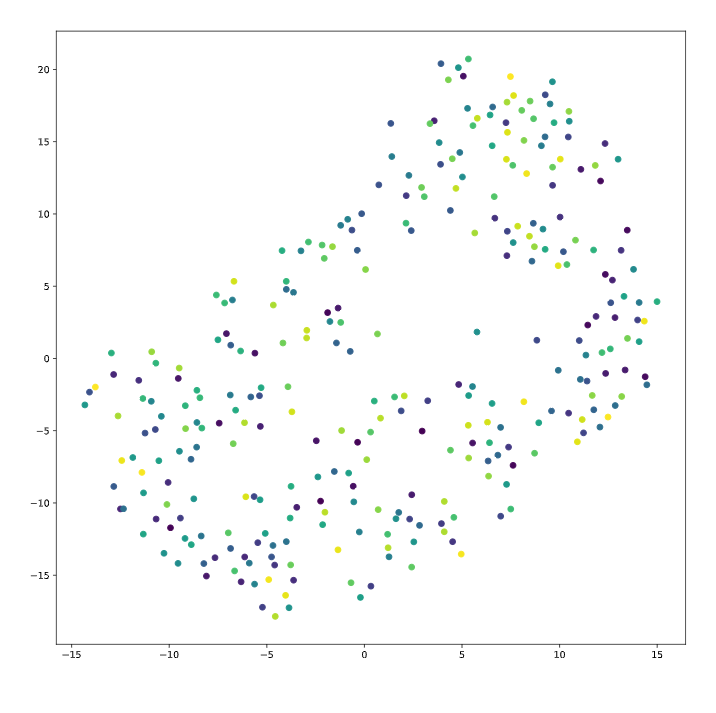


### Figure 13 – Training Siamese Network results

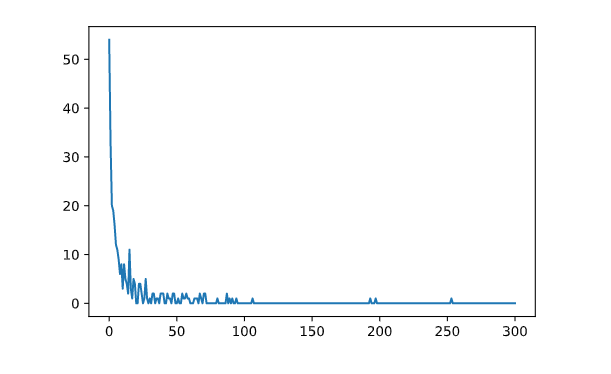




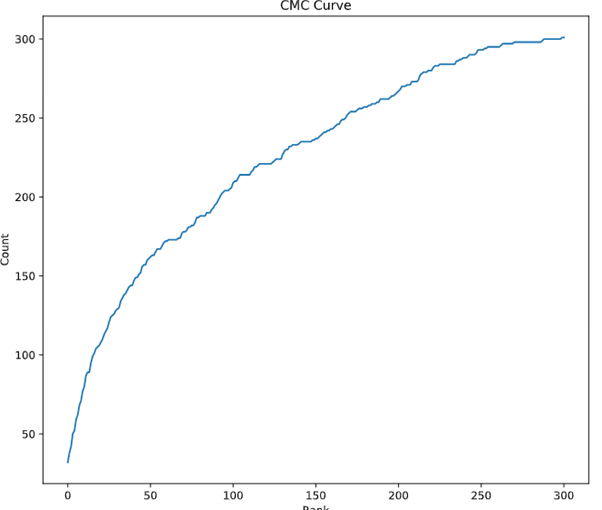
### Figure 14 – Scatter Plot



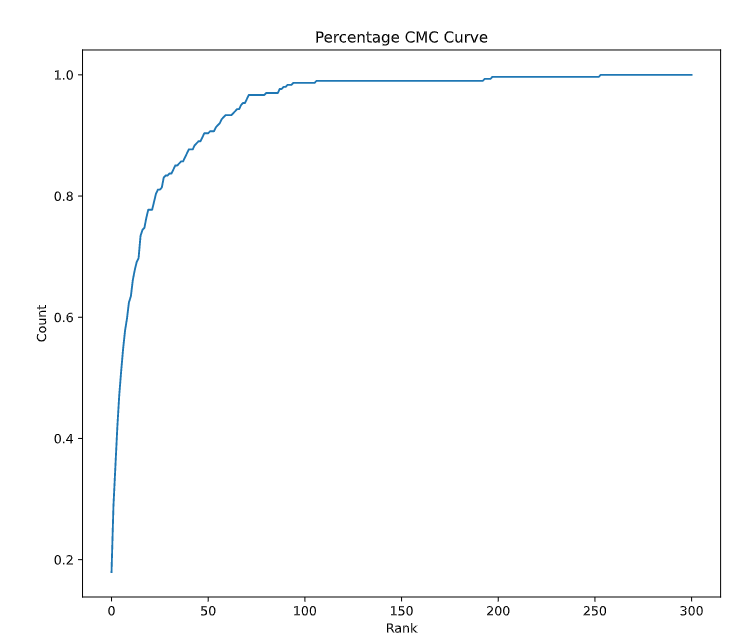
### Figure 15 – CMC



### Figure 16 CMC Curve Siamese Network



### Figure 17 CMC Curve Siamese Network Percentage



## Code Q1

## Code Q2

*# %%*

*from* pathlib *import* Path

*from* cv2 *import* cv2

*import* tensorflow *as* tf

*from* tensorflow *import* keras

*from* tensorflow.keras *import* layers

*from* tensorboard *import* notebook

*from* sklearn.metrics *import* confusion\_matrix, ConfusionMatrixDisplay, plot\_confusion\_matrix

*import* matplotlib.pyplot *as* plt

*import* scipy.io

*import* numpy

*from* sklearn *import* decomposition

*from* sklearn *import* discriminant\_analysis

*import* re

*from* sklearn.manifold *import* TSNE

*from* sklearn.neighbors *import* KNeighborsClassifier

*from* pathlib *import* Path

*import* os

*import* matplotlib.pyplot *as* plt

*import* random

*from* tensorflow.python.client *import* device\_lib

*from* scipy.spatial *import* distance

*import* sys

*import* numpy *as* np

*from* datetime *import* datetime

*#%%*

tf.test.is\_built\_with\_cuda()

*#%%*

physical\_devices = tf.config.list\_physical\_devices('GPU')

print("Num GPUs:", len(physical\_devices))

*# %%*

def readImage(*path*, *name*):

    folder = Path(path).rglob('\*.jpg')

    files = [x *for* x in folder]

    ImageArray = []

    yarray = []

*for* x in files:

        img = cv2.imread(str(x))

*# img = img.flatten('C')*

        ImageArray.append(img)

        test = re.search("[0-9]{4}", str(x))

        yarray.append(int(test[0]))

    obj\_arr = numpy.zeros((2,), *dtype*=numpy.object)

    obj\_arr[0] = ImageArray

    obj\_arr[1] = numpy.transpose([yarray])

    scipy.io.savemat('./Data/Q2/matfiles' + name + '.mat',

*mdict*={'x': obj\_arr[0], 'y': obj\_arr[1]})

def createFig(*imgArray*):

*# print(imgArray)*

    fig = plt.figure(*figsize*=(50, 50))

*for* i in range(10):

        ax = fig.add\_subplot(50, 50, i+1)

        ax.imshow(imgArray[i, :, :, :])

*# ax.imshow(imgArray[:,:,:,i])*

def eval\_model(*model*, *X\_train*, *Y\_train*, *X\_test*, *Y\_test*):

    fig = plt.figure(*figsize*=[25, 8])

    ax = fig.add\_subplot(1, 2, 1)

    conf = plot\_confusion\_matrix(model, X\_train, Y\_train, *normalize*='true', *ax*=ax)

    conf.ax\_.set\_title('Training Set Performance');

    ax = fig.add\_subplot(1, 2, 2)

    conf = plot\_confusion\_matrix(model, X\_test, Y\_test, *normalize*='true', *ax*=ax)

    conf.ax\_.set\_title('Test Set Performance');

    pred = model.predict(X\_test)

    print('Test Accuracy: ' + str(sum(pred == Y\_test)/len(Y\_test)))

*# %% Create 4D arrays for iamge vectors and for Y values. Then making mat files for the data to be stored*

readImage('./Data/Q2/Testing/Gallery', 'Gallery')

readImage('./Data/Q2/Testing/Probe', 'Probe')

readImage('./Data/Q2/training', 'Training')

Training = scipy.io.loadmat('./Data/Q2/matfilesTraining.mat')

Probe = scipy.io.loadmat('./Data/Q2/matfilesProbe.mat')

Gallery = scipy.io.loadmat('./Data/Q2/matfilesGallery.mat')

*#%%*

training\_x\_Org = Training['x'].astype("float32") / 255.0

training\_x = training\_x\_Org

training\_y\_Org = Training['y'].astype("float32")

training\_y = training\_y\_Org

testing\_Probe\_x\_Org = Probe['x'].astype("float32") / 255.0

testing\_Probe\_x = testing\_Probe\_x\_Org

testing\_Probe\_y\_Org = Probe['y'].astype("float32")

testing\_Probe\_y = testing\_Probe\_y\_Org

testing\_Gallery\_x\_Org = Gallery['x'].astype("float32") / 255.0

testing\_Gallery\_x = testing\_Gallery\_x\_Org

testing\_Gallery\_y\_Org = Gallery['y'].astype("float32")

testing\_Gallery\_y = testing\_Gallery\_y\_Org

*# %% Create Figures*

createFig(testing\_Gallery\_x)

createFig(testing\_Probe\_x)

createFig(training\_x)

*# %%*

*# Flatten Arrays*

Flatten\_trainx = training\_x.flatten().reshape(5933, 24576)

Flatten\_testingGalleryx = testing\_Gallery\_x.flatten().reshape(301, 24576)

Flatten\_testingProbex = testing\_Probe\_x.flatten().reshape(301, 24576)

print("Done")

*# %%*

pca = decomposition.PCA()

pca.fit(Flatten\_trainx)

transformed = pca.transform(Flatten\_trainx)

transformed\_test\_Gallery = pca.transform(Flatten\_testingGalleryx)

transformed\_test\_Probe = pca.transform(Flatten\_testingProbex)

cumulative\_sum = numpy.cumsum(pca.explained\_variance\_ratio\_, *axis*=0)

top95 = numpy.where(cumulative\_sum > 0.95)[0][0]

fig = plt.figure(*figsize*=[20, 5])

count = 0

*for* i in range(5):

*for* j in range(2):

        ax = fig.add\_subplot(2, 10, count\*2 + 1)

        ax.imshow(numpy.reshape(

            Flatten\_trainx[count, :] - pca.mean\_, (128, 64, 3)))

        ax.set\_title('Original')

        ax = fig.add\_subplot(2, 10, count\*2 + 2)

        pca.mean\_

        ax.imshow(numpy.reshape(pca.components\_[0:top95, :].transpose().dot(

            numpy.reshape(transformed[count, 0:top95], (-1, 1))), (128, 64, 3)))

        ax.set\_title('95% Reconstruction')

        count += 1

*#%%*

*#transformed // testing*

*#transformed\_test\_Gallery // Test probe*

*#transformed\_test\_Probe // Test Probe*

*# training\_y*

*# testing\_Probe\_y*

*# testing\_Gallery\_y*

*#testing\_Probe\_y.shape*

print(transformed\_test\_Probe)

*#%%*

dist = distance.cdist(transformed\_test\_Probe, transformed\_test\_Gallery, 'euclidean')

num\_ids = len(numpy.unique(testing\_Gallery\_y))

ranked\_histogram = numpy.zeros(num\_ids)

*for* i in range(len(testing\_Gallery\_y)):

*#print("True: {}".format(testing\_Gallery\_y[i]))*

    order = numpy.argsort(dist[i])

    ranked = testing\_Gallery\_y[order]

    ranked\_result = numpy.where(ranked == testing\_Probe\_y[i])[0][0]

    ranked\_histogram[ranked\_result] += 1

*#print(ranked\_result)*

*# print(ranked\_histogram)*

plt.plot(ranked\_histogram)

*#%%*

cmc = numpy.zeros(num\_ids)

*for* i in range(num\_ids):

    cmc[i] = numpy.sum(ranked\_histogram[:(i + 1)])

fig = plt.figure(*figsize*=[20, 8])

ax = fig.add\_subplot(1, 2, 1)

ax.plot(cmc)

ax.set\_xlabel('Rank')

ax.set\_ylabel('Count')

ax.set\_title('CMC Curve')

ax = fig.add\_subplot(1, 2, 2)

ax.plot(cmc/num\_ids)

ax.set\_xlabel('Rank')

ax.set\_ylabel('Count')

ax.set\_title('Percentage CMC Curve')

*#%%*

print("Top 1 performance: {}".format(cmc[0]/num\_ids))

print("Top 5 performance: {}".format(cmc[4]/num\_ids))

print("Top 10 performance: {}".format(cmc[9]/num\_ids))

print("Top 100 performance: {}".format(cmc[99]/num\_ids))

*# %% Deep learning  Method Helper functions*

def GetSiameseData(*imgs*, *labels*, *batch\_size*):

    image\_a = numpy.zeros((batch\_size, numpy.shape(imgs)[1], numpy.shape(imgs)[2], numpy.shape(imgs)[3]));

    image\_b = numpy.zeros((batch\_size, numpy.shape(imgs)[1], numpy.shape(imgs)[2], numpy.shape(imgs)[3]));

    label = numpy.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 PairGenerator(*imgs*, *labels*, *batch\_size*):

*while* True:

        [image\_a, image\_b], label = GetSiameseData(imgs, labels, batch\_size)

*yield* [image\_a, image\_b], label

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

*#%%*

deepLearning\_x\_train = training\_x\_Org.reshape(training\_x\_Org.shape[0], 128, 64, 3)

deepLearning\_y\_train = training\_y\_Org.reshape(training\_y\_Org.shape[0], 1)

deepLearning\_x\_test = testing\_Gallery\_x\_Org.reshape(testing\_Gallery\_x\_Org.shape[0], 128, 64, 3)

deepLearning\_y\_test = testing\_Gallery\_y\_Org.reshape(testing\_Gallery\_y\_Org.shape[0], 1)

*#%%*

test = PairGenerator(deepLearning\_x\_train, deepLearning\_y\_train, 20)

x, y = next(test)

print(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,:,:,0])

    ax.set\_title('Pair ' + str(i) +'; Label: ' + str(y[i]))

    ax = fig.add\_subplot(2, 10, i\*2 + 2)

    ax.imshow(x[1][i,:,:,0])

    ax.set\_title('Pair ' + str(i) +'; Label: ' + str(y[i]))

*#%%*

embedding\_size = 32

dummy\_input = keras.Input((128, 64, 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')

base\_network.summary()

*#%%*

input\_a = keras.Input((128, 64, 3), *name*='InputA')

input\_b = keras.Input((128, 64, 3), *name*='InputB')

embedding\_a = base\_network(input\_a)

embedding\_b = base\_network(input\_b)

*# %%*

combined = layers.concatenate([embedding\_a, embedding\_b])

combined = layers.Dense(128, *activation*='relu')(combined)

output = layers.Dense(1, *activation*='sigmoid')(combined)

siamese\_network = keras.Model([input\_a, input\_b], output, *name*='SiameseNetwork')

siamese\_network.summary()

*# %%*

keras.utils.plot\_model(siamese\_network, *show\_shapes*=True)

*#%%*

siamese\_network.compile(*loss*='binary\_crossentropy', *optimizer*=keras.optimizers.RMSprop(), *metrics*=['accuracy'])

*# %%*

batch\_size = 64

training\_gen = PairGenerator(deepLearning\_x\_train, deepLearning\_y\_train, batch\_size)

siamese\_test\_x, siamese\_test\_y = GetSiameseData(deepLearning\_x\_test, deepLearning\_y\_test, 10000)

siamese\_network.fit(training\_gen, *steps\_per\_epoch* = 30000 // batch\_size, *epochs*=10, *validation\_data* = (siamese\_test\_x, siamese\_test\_y))

*#%%*

x, y = GetSiameseData(deepLearning\_x\_test, deepLearning\_y\_test, 10)

res = siamese\_network.predict(x)

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,:,:,0])

    ax.set\_title('Pair ' + str(i) +'; Label: ' + str(y[i]))

    ax = fig.add\_subplot(2, 10, i\*2 + 2)

    ax.imshow(x[1][i,:,:,0])

    ax.set\_title('Predicted: ' + str(res[i]))

*# %%*

*for* i in range(10):

    x[0][i,:] = x[0][0,:]

res = siamese\_network.predict(x)

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,:,:,0])

    ax.set\_title('Pair ' + str(i))

    ax = fig.add\_subplot(2, 10, i\*2 + 2)

    ax.imshow(x[1][i,:,:,0])

    ax.set\_title('Predicted: ' + str(res[i]))

*# %%*

print(deepLearning\_y\_test)

embeddings = base\_network.predict(deepLearning\_x\_test)

tsne\_embeddings = TSNE(*random\_state*=4).fit\_transform(embeddings)

fig = plt.figure(*figsize*=[12, 12])

ax = fig.add\_subplot(1, 1, 1)

ax.scatter(tsne\_embeddings[:,0], tsne\_embeddings[:,1], *c* = deepLearning\_y\_test.flatten());

*# %% CNN*

dummy\_input = keras.Input((128, 64, 3))

vgg\_network = vgg\_net(dummy\_input, [8, 16, 32], [256], 0.2, 0)

embedding\_layer = layers.Dense(embedding\_size, *activation*=None)(vgg\_network)

output\_layer = layers.Dense(10, *activation*=None, *name*='feature\_extractor')(embedding\_layer)

vgg\_network = keras.Model(dummy\_input, output\_layer, *name*='SimpleVGGNetwork')

vgg\_network.summary()

*#%%*

*#%%*

embeddings = base\_network.predict(deepLearning\_x\_test)

tsne\_embeddings = TSNE(*random\_state*=4).fit\_transform(embeddings)

fig = plt.figure(*figsize*=[12, 12])

ax = fig.add\_subplot(1, 1, 1)

ax.scatter(tsne\_embeddings[:,0], tsne\_embeddings[:,1], *c* = deepLearning\_y\_test.flatten());

*#%%*

vgg\_network.compile(*loss*=keras.losses.SparseCategoricalCrossentropy(*from\_logits*=True),

*optimizer*=keras.optimizers.RMSprop(),

*metrics*=['accuracy'])

history = vgg\_network.fit(testing\_Gallery\_x\_Org, testing\_Gallery\_y\_Org,

*batch\_size*=64,

*epochs*=10,

*validation\_data* = (deepLearning\_x\_test, deepLearning\_y\_test))

*# %%*

intermediate\_layer\_model = keras.Model(*inputs*=vgg\_network.input,

*outputs*=vgg\_network.get\_layer('feature\_extractor').output)

embeddings = intermediate\_layer\_model(deepLearning\_x\_test)

tsne\_embeddings = TSNE(*random\_state*=4).fit\_transform(embeddings)

fig = plt.figure(*figsize*=[12, 12])

ax = fig.add\_subplot(1, 1, 1)

ax.scatter(tsne\_embeddings[:,0], tsne\_embeddings[:,1], *c* = deepLearning\_y\_test.flatten());

*# %%*

*# %%*

embeddingGallery = base\_network.predict(testing\_Gallery\_x\_Org)

embeddingProbe = base\_network.predict(testing\_Probe\_x\_Org)

dist = distance.cdist(embeddingProbe,

                      embeddingGallery, 'euclidean')

num\_ids = len(numpy.unique(testing\_Gallery\_y))

ranked\_histogram = numpy.zeros(num\_ids)

*for* i in range(len(testing\_Gallery\_y)):

*#print("True: {}".format(testing\_Gallery\_y[i]))*

    order = numpy.argsort(dist[i])

    ranked = testing\_Gallery\_y[order]

    ranked\_result = numpy.where(ranked == testing\_Probe\_y[i])[0][0]

    ranked\_histogram[ranked\_result] += 1

*# print(ranked\_result)*

print(ranked\_histogram)

plt.plot(ranked\_histogram)

cmc = numpy.zeros(num\_ids)

*for* i in range(num\_ids):

    cmc[i] = numpy.sum(ranked\_histogram[:(i + 1)])

fig = plt.figure(*figsize*=[20, 8])

ax = fig.add\_subplot(1, 2, 1)

ax.plot(cmc)

ax.set\_xlabel('Rank')

ax.set\_ylabel('Count')

ax.set\_title('CMC Curve')

ax = fig.add\_subplot(1, 2, 2)

ax.plot(cmc/num\_ids)

ax.set\_xlabel('Rank')

ax.set\_ylabel('Count')

ax.set\_title('Percentage CMC Curve')

*#%%*

print("Top 1 performance: {}".format(cmc[0]/num\_ids))

print("Top 5 performance: {}".format(cmc[4]/num\_ids))

print("Top 10 performance: {}".format(cmc[9]/num\_ids))

print("Top 100 performance: {}".format(cmc[99]/num\_ids))

*# %%*