Machine Learning – Designing a Neural Network

Caleb Scott Bedekamp (BRDCAL003)

Taariq Daniels (DNLTAA002)

Due: 24/06/2020

Introduction

The field of *Artificial intelligence* (A.I) revolves around the study in which machines are provided with the means to exhibit reasoning. Such means are related to the self-conscious human mind. A.I seeks to simulate such a mind and equip it to machines, such that they can effectively operate in a variety of scenarios (potentially unknown) without a human explicitly telling it what to do (through direct coding, etc).

Machine learning is subcategory of A.I and involves providing a machine with the means to learn and apply what it has learned (its 'experience') to real-life applications. Machine Learning, on its own, is vast and provides many creative ways of going about it. It is present in everyday technology and continuously improves society's way of life by offering better security (detection of fraud, facial recognition), assisting in making medical diagnosis, etc.

One type of model of machine learning is a *neural network.* From a medical perspective, *a neural network* of a brain describes vast layers of connections between brain cells (known as neurons). A (artificial) neural network attempts to mimic the workings of a brain by utilising computer algorithms. Doing this, the artificial network can recognise patterns through a learning process and eventually apply what it has learn to situations that is *similar* (but never-before seen) to that which it already learnt from.

Aim

The aim of this project is to apply machine learning and design a neural network with the objective of achieving reliable *facial recognition*.

A particular type of neural network will be explored and designed using existing coding libraries that have been created for such a purpose. The neural network will be trained from scratch until ready for testing.

Literature Review

The basis of artificial neural networks was first conceived as early as 1943 by Warren McCulloch (a neurophysiologist) and Walter Pitts (a logician), who, together, defined the first mathematical model of a neural network [1]. They defined the mapping of the inner workings of the brain and the artificial neural networks we have used till this day.

Since then, there have been number of innovations made in in the field. While many of these innovations are relevant to the context of this report, for the sake of being concise, we'll only review the literature that describes a Convolutional Neural Network (CNN), which conveniently encapsulates the prior innovations.

The literature that was reviewed was a research paper titled 'MobileNets: Efficient Convolutional Neural Networks for Mobile VisionApplications' [2]. The paper provided a great deal of insight and formed the basis of the model used in this project.

This paper describes using a class of CNN (known as MobileNet) to perform mobile vision applications (object detection, facial recognition, etc.). It provides details, regarding the shortcomings of a traditional CNN and how their proposed architecture (MobileNet) can alleviate these shortcomings, using a technique called *separable depth wise convolution*. The characteristics

each element of the architecture is explained in detail and compared with alternative techniques. Metrics used to describe model performance are in the form computational efficiency and accuracy, which are generated from multiple experiments before being compared with other existing models.

Method

As hinted at earlier, there are several ways to go about designing a neural network; each method carrying its own advantages and disadvantages. In our case, we have selected to design a *convolutional Siamese neural network.*

Network architecture

The network itself will be a convolutional neural network (CNN). The class of the CNN architecture is known as *MobileNet.*

A vanilla neural network; namely a multilayer perceptron network (MLP); is a class of artificial networks. It consists of multiple fully connected layers and is capable of distinguishing data that may not conform to a linear decision boundary.

A CNN is the natural evolution of the MLP. Unlike the MLP, the CNN is not fully connected (is sparsely connected), which results in a reduction of weighting parameters and offers improved training efficiency and effectiveness. The CNN is also capable of retaining spatial information, which is extremally useful for feature recognition.

The CNN architecture, MobileNet, differs from a standard CNN in that it makes use of depth wise separable convolution, as opposed to only standard convolution layers. The advantage of the MobileNet architecture is a reduction in model size and computation [2].

A neural network consists of several 'layers'-each layer can offer a variety of operations. A main characteristic of a convolution neural network is the usage of 'filters' (also known as kernels) that perform the convolution function. The different operations of each layer are dictated by the type of filter used.

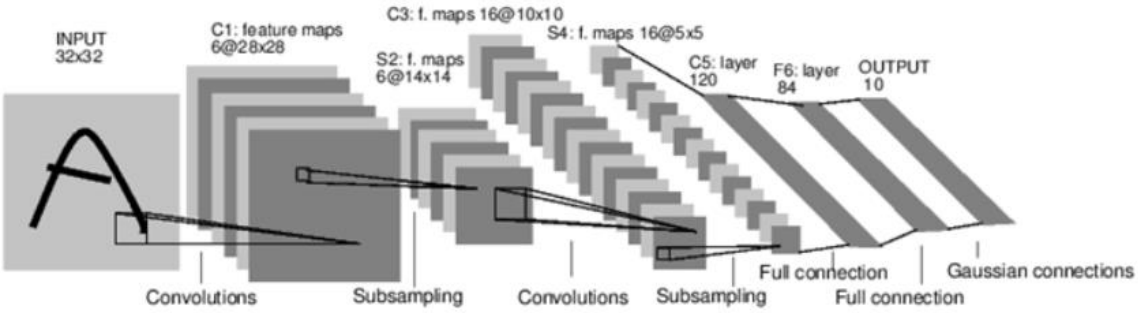


Figure 1: A generic fully connected convolutional neural network (LeCun et al., 1998) [3]

The layers that comprise our CNN (in no particular order) are as follows:

Input layer (Input); the first layer of our network. It accepts a feature vector in the form of a $60 \times 3 \times 224 \times 224$ tensor where 60 is the batch size.

Convolutional layer (Conv); the convolutional layer uses filters to detect features on an image. Each filter has different 'weighting' values that correspond to different features and are of different dimensions. These filters are applied and 'slide' across the inputted image. At each unique location

on the image, element-wise multiplication and summation occurs between the filter matrix and receptive region (area that the filter is over). Based on the output of this operation, we can determine the likelihood that a particular feature is present in a given location. Multiple filters (with different weights and sizes) are used to further identify features in an image.

Within the early layers of the CNN, lower-level features, such as simple lines or curves, are detected. The further and deeper one goes into network, the more higher-level features, such as eyes or mouths, are identified.

As stated earlier, the *MobileNet* CNN architecture uses depth wise separable convolution. A standard convolution involves using a kernel of the form: $\{w \times h \times n \times m\}$ where

- w and h are the width and length of the kernel, respectively and
- n and m are the number of input and output channels (the depth), respectively.

An input image (e.g. size: $12 \times 12 \times 3$) subjected to standard convolution by a filter (e.g. size: $5 \times 5 \times 3 \times 100$) will produce an output of size $8 \times 8 \times 100$ (assuming a filter stride of 1 and zero padding).

Depth wise separable convolution performs the same function as standard convolution but goes about it in two separate processes; namely depth wise convolution (DW) and pointwise convolution (PW).

DW convolution involves using a $Z \times$ kernels of the form: $\{w \times h\}$ where Z is the number of layers/channels of the input image. Each kernel (e.g. 3 kernels) perform a convolution function on a single layer of the image (e.g. size: 12×12) producing 3 layers each of size 8×8 that stack forming an $8 \times 8 \times 3$ output. [4]

Following the DW convolutional layer, is the PW convolutional layer, which performs point wise convolution on its input. PW convolution involves subjecting the input to a kernel of the form: $\{1 \times 1 \times n \times m\}$. This operation takes an input (e.g. size: $8 \times 8 \times 3$) and subjects it to kernel (e.g. size $1 \times 1 \times 3 \times 100$) to produce an output if size: $8 \times 8 \times 100$. [4]

This means the output of the depth wise separable convolution is identical to that of the standard convolution operation. The advantage in performing DW+PW convolution over standard convolution lies in significant lower number of computations done by separable convolution, resulting in greater model efficiency.

The first layer (following the input layer) is a standard convolution layer; each subsequent convolution layer is a depth wise separable layer.

Multiple convolutional layers are used in our network and between each convolutional layer lie 'hidden layers' that perform unique functions to aid training processes.

Pooling layer (Pool); The pooling layer is responsible for reducing instances of overfitting by the model and increasing model robustness, by down sampling the layer's input. Down sampling refers to a reduction in resolution (size) of input, which reduces the complexity of the model while still preserving important features of the input. The reduction in model complexity results in a lower output variance, increasing model robustness (i.e. improving the model's ability to generalise). [5]

The pooling function we used is known as *Average Pooling* and it differs from the convolutional layer in that it is designed to extract the average presence of features from an image. The pooling filter is placed over the given image and outputs the average value of the receptive area.

Rectified Linear Unit layer (ReLU); The ReLU layer is found in between each convolutional layer and its function itself is very simple; it applies the function $\max(0, x)$ to the input x, effectively mapping any negative inputs to 0. This activation function helps introduce non-linearities into the model. By making the model output non-linearly dependent on input features, it can better distinguish between data that may not conform linearly. Additionally, ReLU reduces model variance by introducing sparsity into the model. It does this by zeroing out some of the filter responses in the net for different inputs. This means, for some inputs, the number of filters affecting the output is reduced.

Batch normalisation (BN); batch normalisation ultimately speeds up the training process and is found at every ReLU layer. It does this by normalising the inputs to the hidden layers, which makes the cost function more symmetrical and reduces the sensitivity of layers (particularly deeper layers) on initial weights. [6]

Normalisation essentially scales the layer input values, such that their activation values' mean and standard deviation is zero and one, respectively. This translates to a contour plot (i.e. the loss curve) that is more symmetrical and easier to traverse. This allows for a larger *training rate* to be used during the training phase, resulting in a quicker convergence onto the cost function's minimum with less chance overshooting the minimum (which is a typical consequence of using a large learning rate). [6]

Batch normalisation refers to the process of normalisation being applied to a group of samples (e.g.3 image samples constitute a single batch), where the mean and standard deviation is taken of the batch's activation values.

Fully connected layer; the final layer of our network. The fully connected layer is comprised of multiple 'sublayers', where each layer is further comprised of neurons/activation functions. Unlike the preceding layers, as the name implies, this layer is 'fully connected' (i.e. each neuron in a layer is connected to every other neuron in the successive layer). The fully connected layer acts as our classifier; based on the preceding layer outputs, the fully connected layer results in a 1000 point vector which represents the input image. Since the network does not act as a classifier in itself but rather generates a corresponding vector for an image input, the Softmax layer indicated in Table 1 below has been omitted in our design.

The Siamese network method utilises 2 identical networks that operate in parallel for its evaluation process; this means a single network structure is utilised, and a pair of inputs are fed into the network at a time. The Siamese network ultimately compares the differences/similarities between the input pair and outputs a 'grade (value)' describing the dissimilarity. In our case, the input will be an image pair and the value describing similarity or dissimilarity is the distance between the output 1000-point vectors. Images that are similar will have a lower grade value than images that are dissimilar. The Siamese network is advantageous over a traditional (single) network in that it can be trained to produce unique vectors to represent unique faces and when applying the network, any face can be added to the database to compare against without any additional training necessary. For example, in a security application, when registering a new face to be accepted, a series of images of that face could be saved. Then, when tested with a new face, the network would simply compare the new vector produced with the vector produced by the original registration image to determine whether or not the faces match.

Table 1: MobileNet architecture (Howard, Zhu, Chen, Kalenichenko, Wang, Weyand, Andreetto, Adam, 2017) [2]

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Data Structure

Our learning (training and validation) and testing data will be a series of face images of individuals, at various angles. Each image was numbered and placed into a labelled folder according to the individual that the image is of.

A total of 13 different classes (where each class corresponds to a different individual), each containing 51 images (where each image is a picture of a class's corresponding individual at a different angle) make up our training data. A total of 6 classes (of the same directory structure as the training data) make up our validation data and testing data. Due to insufficient data available, we used a portion of the data used for validation, for the final testing as explained below.

Training

The basic principal behind teaching a neural network is providing it with the correct solutions that correspond to a particular input. The correct solution is then compared with the actual output of the model; an optimiser algorithm is used to bring the actual output 'in line' with the correct solution. This process of learning is more formally known as *Supervised Learning*.

By repeating this process over a large enough sample size, the network will gradually infer its own set of 'rules', which will eventually allow for it to discern between different inputs and provide the correct solutions by itself.

A pair of images (that are of the same individual) will be fed into the network. The network will output a value close to zero in this case, as the images are 'similar'. Then, a different pair of images (that are of different individuals) will be provided as the input to the network. Again, the network will output a value but this time it will be closer one, as the images are highly 'dissimilar'. This

process will be repeated over several sample pairs until the network's ability to discern between image pairs (within a set threshold) reaches an adequate level.

Cost Function

In deep learning applications, the cost function determines the degree to which error (the difference between the predicted output and the actual output) is penalised. There are a variety of such functions available; the one the we selected to use is known as the *triplet loss function* and is given as:

$$L(a, p, n) = \max(0, D(a, p) - D(a, n) + margin)$$ [7]

Where L is a function of an *anchor (a), an anchor-positive (p) and an anchor-negative* (n) image.

As loosely described earlier, an iteration of training involves providing the Siamese network with an image pair, which will then output a 'similarity grade'. What was not previously mentioned is the image that serves as the 'correct/desired' model output, against which the image pair is tested. The image that serves this function is known as the *anchor*. The anchor-positive image is an image that belongs to the same class as the anchor (i.e. same individual but different angle); the anchor-negative image is an image that belongs to entirely separate class (i.e. different individual).

For each training iteration, a random image from a random class is selected to serve as an anchor. A random (but different from the anchor) image is selected from the anchor class to serve as the anchor-positive image. The anchor-negative image is then randomly selected from one of the remaining 12 classes. For our application, we initially generated every possible triplet combination and randomised the order in which they were fed into the model. Care was taken to ensure that every no triplet combination was used more than once during the training process.

This is where the loss function falls in.

The triplet loss function (with the aid of an optimiser) serves to minimise the Euclidean distance between the anchor and the anchor-positive $\{D(a, p)\}$ and maximise the distance between the distance between the anchor and the anchor-negative image $\{D(a, n)\}$. Doing so will result in the loss function tending to zero. The *margin* is a hyperparameter that further helps distinguish between positive and negative images.

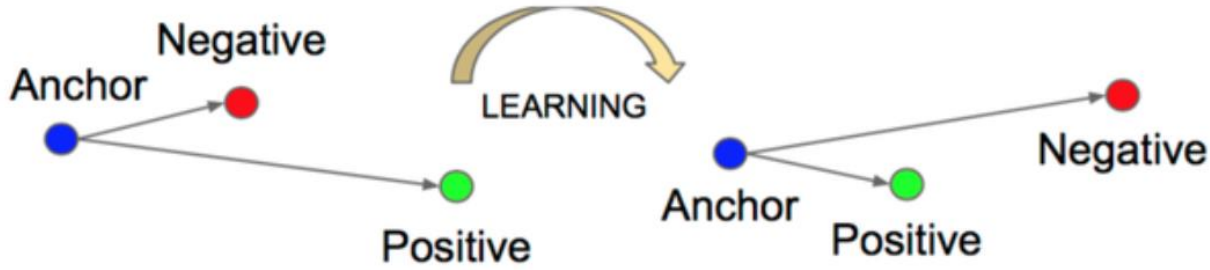


Figure 2: A visual representation of triplet loss function in action. After learning has commenced, the distance between the anchor and the anchor-positive images will shrink; and the distance between the anchor and the anchor-negative images will shrink, resulting in the cost function being minimised. (Reddy, 2019) [8]

Optimiser Algorithm

For the cost function to be minimised, an optimiser algorithm needs to be applied. There are a multitude of algorithms that can be applied to our cost function, which may or may not achieve similar results. For our given cost function, we decided on using a common algorithm known as *Stochastic Gradient Descent.*

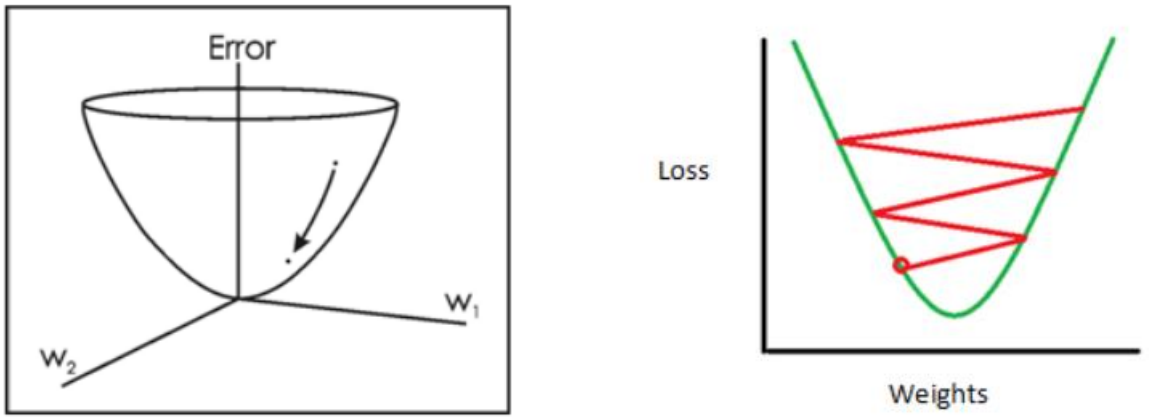


Figure 3: (Left) A 3d visualisation of gradient descent. The objective of the optimiser is finding the weighting values (W1 & W2) that correspond to the trough of the cost/error function. (Right) a visual representation of a large learning rate. The steps size taken (red) between each successive parameter update as it attempts to converge is dependent on the learning rate. (Deshpande, 2016) [9]

As the name somewhat implies, gradient descent involves computing the gradient of the selected cost function at a selected point, with respect to its function parameters, and updating those parameters such that the cost function is minimised. More specifically, it does this by altering the function parameters such that the successive point at which the gradient is computed produces a gentler slope. Throughout the training process, the parameters will be updated and by the end, the optimal parameter values, which result in the convergence of the cost function about zero (i.e. where the loss function is at minimum), will have been determined. The continuous process of updating weights (function parameters) that influence the outputs of inner preceding layers is known as *back propagation*. These optimal function parameters will be applied for when the system is tested and eventually deployed.

The update parameter rule is as follows:

$$\theta_j \coloneqq \theta_j - \eta * \frac{\partial j(\theta)}{\partial \theta_j}$$

Where

- θ_j is a function parameter of the cost function $j(\theta)$
- η is a hyper parameter known as the *learning rate*
    - the learning rate describes the aggressiveness of the gradient descent. A larger η results in larger steps to be taken when moving towards a potential minimum on the cost function; a smaller η results in smaller steps to be taken.

For our case, the function parameters can be seen in the cost function shown previously.

Putting Everything Together

Discussed above were mostly the tools and concepts related to our deep learning application. For the sake of completeness, a more general overview as to how to implement an artificial neural network will be given.

As mentioned earlier, the approach that was taken to train the neural network is known as *Supervised Learning*; a learning scenario where correctly labelled data was made available to the model. What was not mentioned was the validation process that is interweaved in the learning process.

During the training process, weighting parameters are continuously being adjusted (via the cost function and optimiser) to ultimately achieve a reliable model that can carry out the desired operation. After 100 training batches are completed, (i.e. The model has seen 60×100 triplets, where 60 is the batch size), a potential set of optimal weighting values are generated. These weights are then applied, and validation data (comprised of data not previously seen by the model) is then fed into the model in order to evaluate its performance. After this, the model continues training. This process of validation repeated throughout the learning process until the performance of model is within acceptable ranges and we move onto the testing phase. Unfortunately, there are some caveats to this process when using a free application like Google Colab to train the model. These are explained in a later section.

The testing phase is similar to the validation process in that it involves supplying unseen data (separate from validation and training data) to the model in order generate an unbiased performance metric of the model. What distinguishes the two processes from each other is that testing phase is meant to provide a measure as to how the model will perform in the field, while validation is done with the tuning of hyper parameters in mind.

Model Performance Evaluation

The metrics that was used to evaluate the performance of the model is *Accumulated Loss and Evaluation Accuracy.*

Accumulated Loss describes the total loss incurred by the cost function for each batch of samples fed into the model. We want this value to be as close to zero as possible, by the time we complete our final performance evaluations.

Evaluation Accuracy refers to the ratio between the number of correct image matches and total number of samples tested. We want this value to be as close to 1 as possible, by the time we complete our final performance evaluations.

As previously, stated, the model was trained against 13 classes (13 different individuals) of 51 images taken at various angles. Throughout the learning phase, the model's weighting parameters were adjusted, which were validated using the validation data.

The validation data consisted of $(25 \times 50) = 1250$ image pairs. These image pairs were fed into the model and the performance metrics were observed (i.e. the model's ability to discern matching images (same individual) from non-matching pairs (different individuals), was observed). To avoid positively skewing the results, pairs between the same image was avoided.

Training & Validation

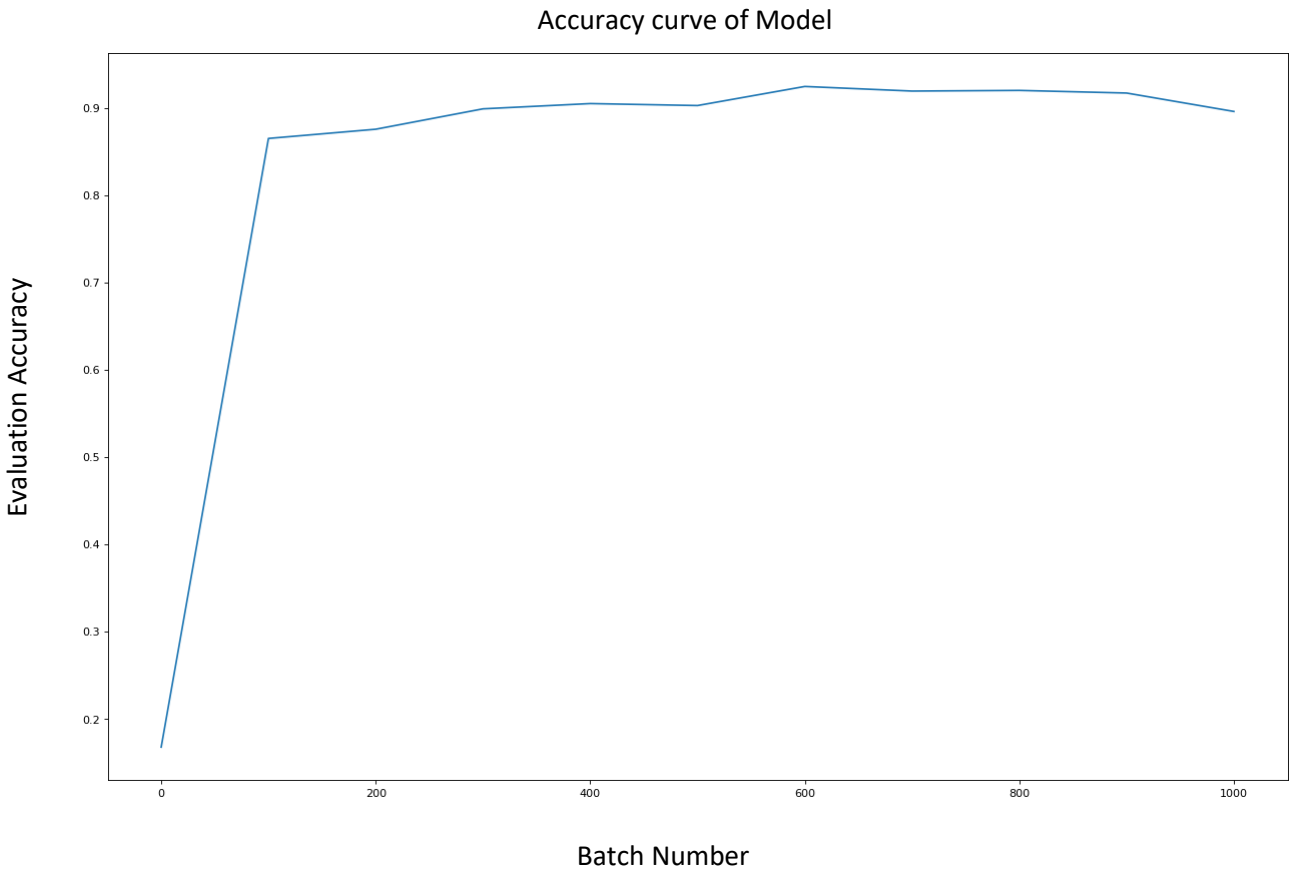


Figure 4: Graph displaying the evaluation accuracy achieved by our model, during the learning phase. More specifically, the graph displays the evaluation accuracy of multiple models that have been subjected to increasing number of data (batches). A single batch consists of 60 triplet pairs.

Referring to Figure: 4, above, observe the continuous increase in accuracy demonstrated by the model as more samples (batches) are fed into the model during the training phase. [For further clarity, a model was saved and validated for every 100 batches used during the training phase.]

However, the model was experiencing cases of overfitting around 600 batches leading to a decrease in validation accuracy. This resulted in the decision to use the model that had been trained on 600 batches during the testing phase.

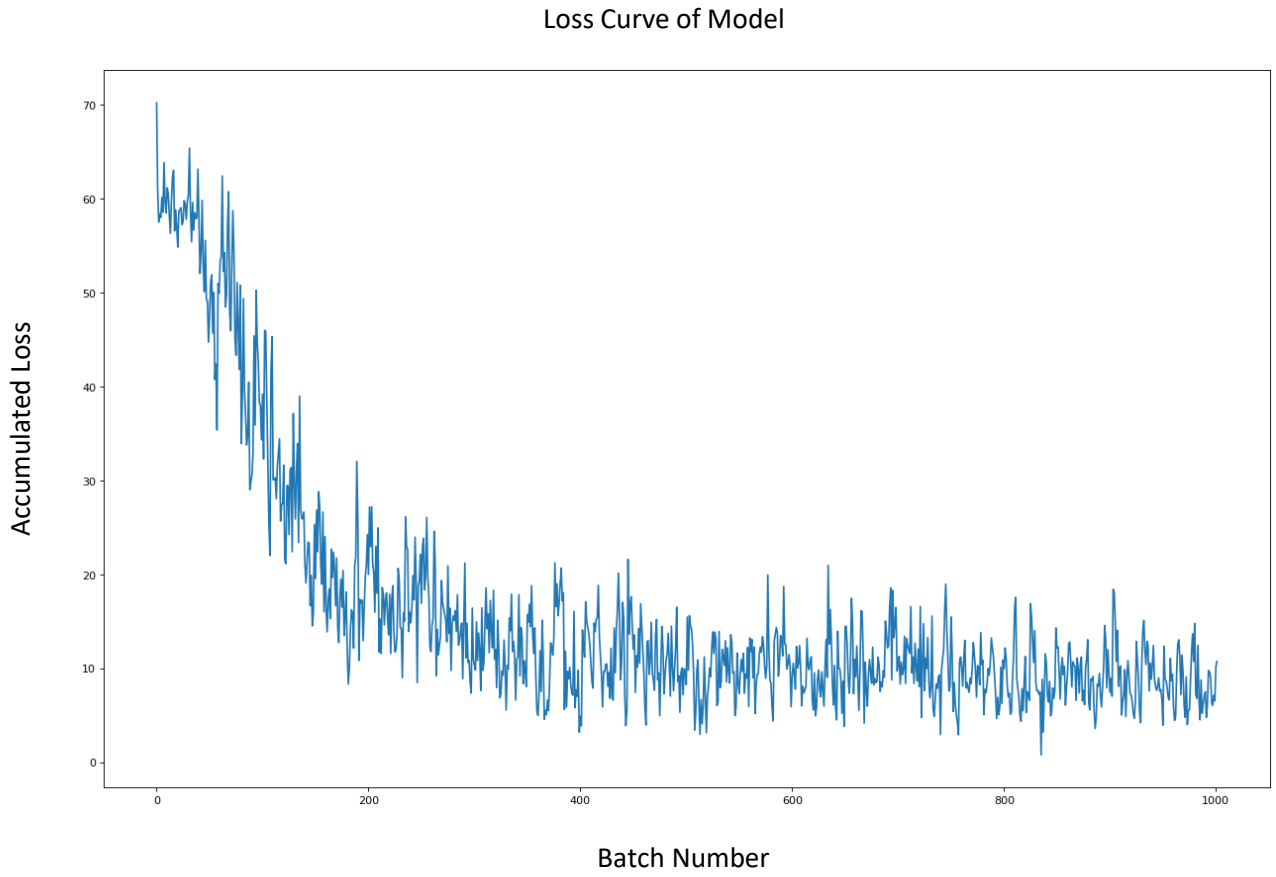


Figure 5: Graph displaying the corresponding loss curve achieved during the learning phase.

Referring to Figure: 5, above, the lost curve is shown to be decreasing as the number of batches seen by the model, increases. The decrease in accumulated loss is initially drastic before levelling out around 600 batches where the model begins overfitting. The noise is attributed to using a relatively small batch size (60 triplets). For reference, at optimal performance the model had seen 36 000 triplets.

The high accuracy/low loss demonstrated, indicated that the model at hand was suitably trained and potentially ready to be deployed into the field. To determine if the model was indeed suitable to be deployed, the trained model was then subjected to the final testing data.

Final Testing

Final testing of the model involved subjecting the trained model to unseen* testing data. *The testing data consisted of $306 \times 306 - 306 = 93330$ image pairs, which *included* the 1250 image pairs used during validation. As previously stated, due to insufficient data available, we used a portion of the data used for validation, for the final testing. The inclusion of seen data will not skew the final accuracy results since they do not affect the model weights or learnable parameters, and

the seen data made up 1.3% of the total testing data. The subtraction of 306 refers to the exclusion of same image pairs.

The final performance results of the model can be seen below.

```
--------------------------------------------
Eval Accuracy:  0.9255228758169934
Correct:  84963
False Positives:  890
False Negatives:  5947
--------------------------------------------
Eval Accuracy:  0.9252383123792153
Correct:  85220
False Positives:  899
False Negatives:  5987
--------------------------------------------
Eval Accuracy:  0.9249880967839674
Correct:  85480
False Positives:  899
False Negatives:  6033
--------------------------------------------
Eval Accuracy:  0.9249768114066309
Correct:  85762
False Positives:  899
False Negatives:  6057
--------------------------------------------
Eval Accuracy:  0.9249118507051943
Correct:  86039
False Positives:  904
False Negatives:  6081
--------------------------------------------
Eval Accuracy:  0.9249330333226187
Correct:  86324
False Positives:  911
False Negatives:  6095
--------------------------------------------
Eval Accuracy:  0.9250074757571874
Correct:  86614
False Positives:  914
False Negatives:  6108
--------------------------------------------
```

Figure 6: An extract of the Evaluation accuracy of the final model.

Referring to Figure: 6, above, the performance metric *Eval Accuracy* indicates that the model is successful in performing facial recognition on (mostly) unseen data, with an Evaluation Accuracy of approximately 93%. This value needs to be slightly reduced however (subtracting 306) since it includes same-image pairs (for which the distance would obviously be zero) resulting in an accuracy of 92.4%

Below are the results of calculating the Euclidean distance between image pairs that are the **same** and between images that are **different**, **before** and **after** training. Due to memory constraints of Google Colab, the distance between only 7650 image pairs were able to be computed.

Table 2: Tabulation of Euclidean distance between image pairs before and after training

	Before Training	**After** Training
average **different** distance	3.1623065069309565e-05	2.7730845892591174
average **same** distance	3.1622864940236096e-05	1.909480437480321

Referring to Table: 2 above, it can be seen that, **before training**, the distance between images of same individuals (*average same distance*) is very small. At first glance, it is natural to assume that this is expected (images of the same individual are similar and therefore closest together). However, looking at the *average different distance* (the distance between image pairs of different individuals), the distance is also very small, which contradicts the initial observation of similar images being nearer to each other. This indicates that untrained model fails to distinguish between image pairs that are similar and image pairs that are different.

After training, we can see how the *average different distance* becomes drastically increased, indicating that the model can identify images pairs that are not of the same individual. The Average same distance also sees an increase after training; this is believed to be the result of clustering of data performed by the model. Despite this, there is a clear boundary between image pairs of different and same individuals, indicating that the model can indeed distinguish between the two cases.

Conclusion

An artificial neural network can perform a tremendous number of useful applications in a wide range of fields. This project demonstrated but one of these applications; facial recognition. The results may not have been perfect, but the limitations of the model design are known and can be improved upon.

Some of these limitations and corresponding improvements include:

- Limited variation in the dataset. Using a dataset comprised of images with more varied backgrounds and of individuals of different sexes, races, etc. may yield a more robust model.
- Adjusting the data structure, such that we have more classes and fewer instances of each class will also improve the model's ability to generalise.
- Hardware/software constraints. Google Colab's memory and time allocations are very limiting. Training and evaluation would have been performed together if Colab's memory allocation had allowed for it. Instead, it was performed separately, as explained in the code appendix.

All things considered, we were able to successfully achieve reliable facial recognition using machine learning libraries and software that was freely available.

References

[1] Palm G. 'Warren McCulloch and Walter Pitts: A Logical Calculus of the Ideas Immanent in Nervous Activity', In*: Palm G., Aertsen A. (eds) Brain Theory.* Springer, Berlin, Heidelberg. DOI: https://doi.org/10.1007/978-3-642-70911-1_14

[2] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, H. Adam, 'MobileNets: Efficient Convolutional Neural Networks for Mobile VisionApplications', April 17, 2017. [Online]. Available: https://arxiv.org/pdf/1704.04861.pdf [Accessed: June 21, 2020].

[3] M. Rizwan, 'LeNet-5 – A Classic CNN Architecture,' Sep 30, 2018. [Online]. Available: https://engmrk.com/lenet-5-a-classic-cnn-architecture/ [Accessed: June 21, 2020].

[4] C. Wang, 'A Basic Introduction to Separable Convolutions,' Aug 14, 2018. [Online]. Available: https://towardsdatascience.com/a-basic-introduction-to-separable-convolutions-b99ec3102728

[5] J. Brownlee, 'A Gentle Introduction to Pooling Layers for Convolutional Neural Networks,' April 22, 2019. [Online]. Available: https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/ [Accessed: June 21, 2020].

[6] CodeEmporium, 'Batch Normalization- EXPLAINED!', *Youtube,* March 9, 2020 [Video file]. Available: https://www.youtube.com/watch?v=DtEq44FTPM4. [Accessed: June 21, 2020].

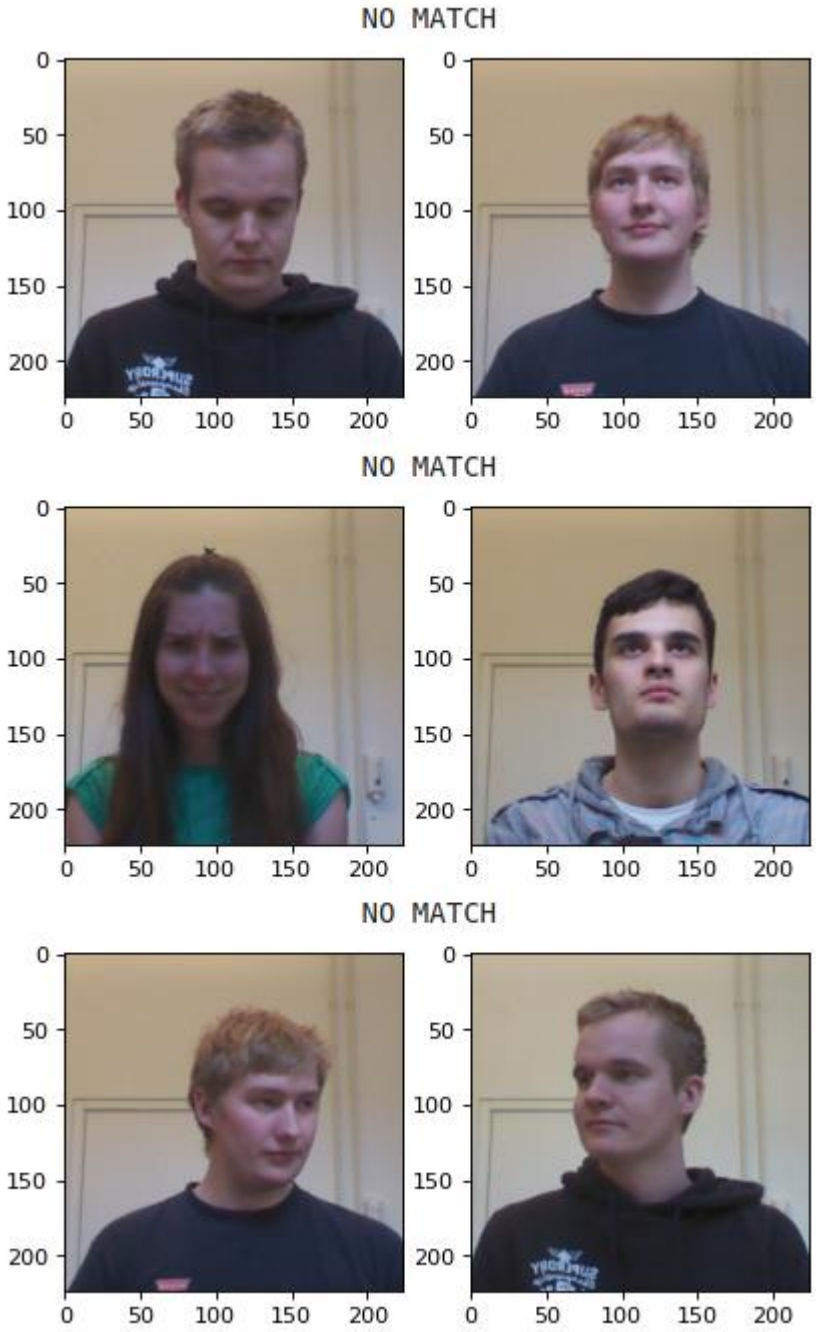
[7] R. Gandhi, 'Siamese Network & Triplet Loss,' May 16, 2018. [Online]. Available: https://towardsdatascience.com/siamese-network-triplet-loss-b4ca82c1aec8 [Accessed: May 30, 2020].

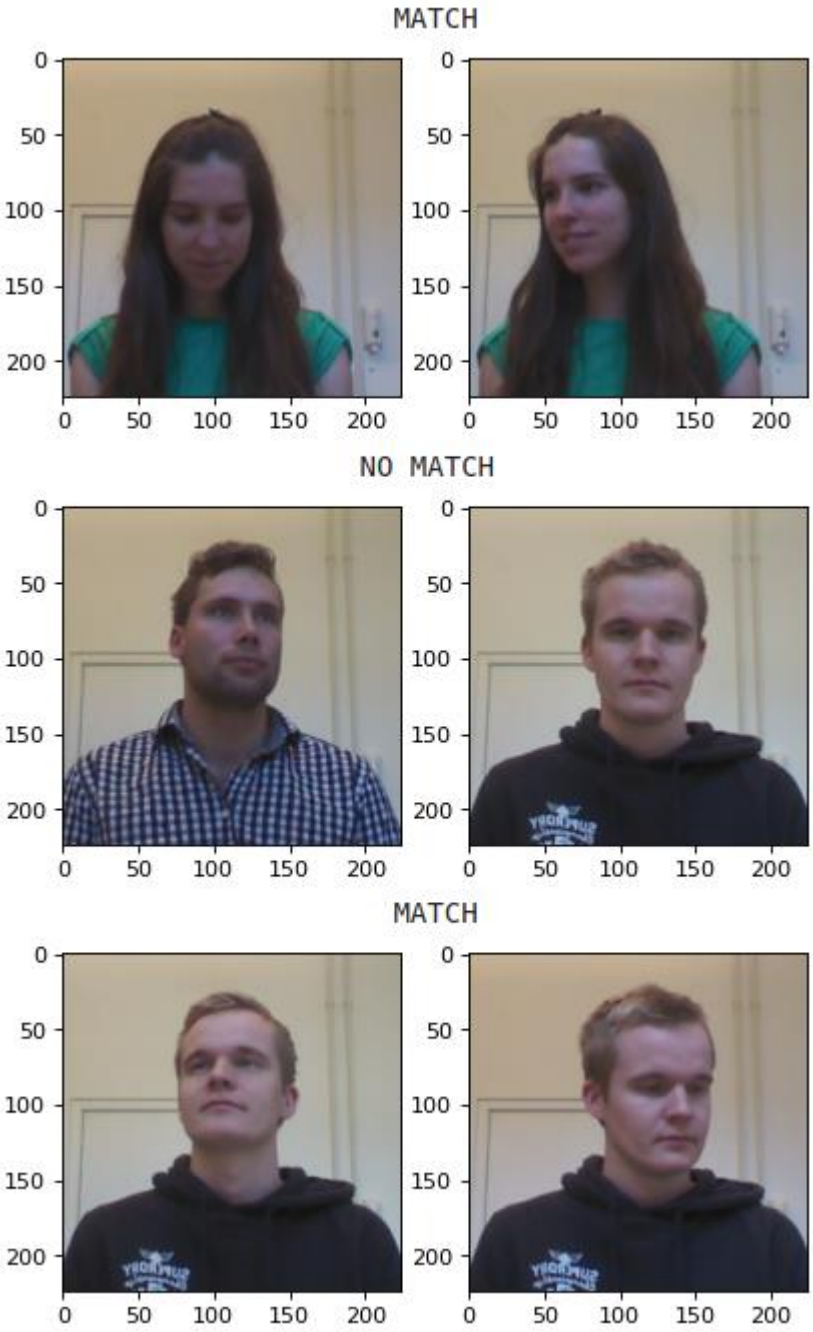
[8] S. Reddy, 'Triplet Loss,' Jan 12, 2019. [Online]. Available: https://medium.com/@susmithreddyvedere/triplet-loss-b9da35be21b8 [Accessed: May 30, 2020].

[9] A. Deshpande, 'A Beginner's Guide To Understanding Convolutional Neural Networks', July 20, 2016. [Online]. Available: https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/ [Accessed: May 30, 2020].

Appendix

A.1-visual demonstration of facial recognition in action





A.2-model code (link to GitHub repository: https://github.com/Caleb-sb/DSP_Siamese_Model)

Necessary Imports:

In [1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

import torchvision
from torchvision import datasets as dset, transforms
import matplotlib.pyplot as plt

import random
from random import randint
from google.colab import drive
from os import path
```

Importing dataset from Google Drive

In [ ]:
```
if (not path.exists('./test')):
  drive.mount('/content/drive')
  !cp -r ./drive/My\ Drive/SiameseFaces/train .
  !cp -r ./drive/My\ Drive/SiameseFaces/test .

  #drive.flush_and_unmount()
  !rm -r ./sample_data
else: print("Dataset imported from Drive")
```

Constants

In [3]:
```
batch_size   =  60  #Divides nicely into 20287800
PER_CLASS    =  51
NUM_CLASSES  =  13
input_size   =  224
EPOCHS       =  1;

#Allowing for GPU use if cuda is available
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
```

```
cuda:0
```

Neural Net Definition:

The net defined below will be incorporated as a Siamese network to facilitate facial recognition. It is a slightly adapted version of MobileNet V1, the structure for which can be found here: https://arxiv.org/abs/1704.04861

In [4]:
```
class Net(nn.Module):
  '''The forward function defines what happens to one input. When evaluating, will call forward twic
e'''
  def __init__(self):
    super(Net, self).__init__();
    # Input channels, output channels, kernel size

    self.conv1  =  nn.Conv2d(3, 32, 3, stride=2, padding=1)
    self.bn1    =  nn.BatchNorm2d(32)
    self.conv2  =  nn.Conv2d(32, 32, 3, stride=1, groups=32, padding=1)
    self.bn2    =  nn.BatchNorm2d(32)
    self.conv3  =  nn.Conv2d(32, 64, 1, stride=1)
    self.bn3    =  nn.BatchNorm2d(64)
    self.conv4  =  nn.Conv2d(64, 64, 3, stride=2, groups=64, padding=1)
    self.bn4    =  nn.BatchNorm2d(64)
    self.conv5  =  nn.Conv2d(64, 128, 1, stride=1)
    self.bn5    =  nn.BatchNorm2d(128)
    self.conv6  =  nn.Conv2d(128, 128, 3, stride=1, groups=128, padding=1)
    self.bn6    =  nn.BatchNorm2d(128)
    self.conv7  =  nn.Conv2d(128, 128, 1, stride=1)
    self.bn7    =  nn.BatchNorm2d(128)
    self.conv8  =  nn.Conv2d(128, 128, 3, stride=2, groups=128, padding=1)
    self.bn8    =  nn.BatchNorm2d(128)
    self.conv9  =  nn.Conv2d(128, 256, 1, stride=1)
    self.bn9    =  nn.BatchNorm2d(256)
    self.conv10 =  nn.Conv2d(256, 256, 3, stride=1, groups=256, padding=1)
    self.bn10   =  nn.BatchNorm2d(256)
    self.conv11 =  nn.Conv2d(256, 256, 1, stride=1)
    self.bn11   =  nn.BatchNorm2d(256)
    self.conv12 =  nn.Conv2d(256, 256, 3, stride=2, groups=256, padding=1)
    self.bn12   =  nn.BatchNorm2d(256)
    self.conv13 =  nn.Conv2d(256, 512, 1, stride=1)
    self.bn13   =  nn.BatchNorm2d(512)
    self.conv14 =  nn.Conv2d(512, 512, 3, stride=1, groups=512, padding=1)
    self.bn14   =  nn.BatchNorm2d(512)
    self.conv15 =  nn.Conv2d(512, 512, 1, stride=1)
    self.bn15   =  nn.BatchNorm2d(512)
    self.conv16 =  nn.Conv2d(512, 512, 3, stride=1, groups=512, padding=1)
    self.bn16   =  nn.BatchNorm2d(512)
    self.conv17 =  nn.Conv2d(512, 512, 1, stride=1)
    self.bn17   =  nn.BatchNorm2d(512)
    self.conv18 =  nn.Conv2d(512, 512, 3, stride=1, groups=512, padding=1)
    self.bn18   =  nn.BatchNorm2d(512)
    self.conv19 =  nn.Conv2d(512, 512, 1, stride=1)
    self.bn19   =  nn.BatchNorm2d(512)
    self.conv20 =  nn.Conv2d(512, 512, 3, stride=1, groups=512, padding=1)
    self.bn20   =  nn.BatchNorm2d(512)
    self.conv21 =  nn.Conv2d(512, 512, 1, stride=1)
    self.bn21   =  nn.BatchNorm2d(512)
    self.conv22 =  nn.Conv2d(512, 512, 3, stride=1, groups=512, padding=1)
    self.bn22   =  nn.BatchNorm2d(512)
    self.conv23 =  nn.Conv2d(512, 512, 1, stride=1)
    self.bn23   =  nn.BatchNorm2d(512)
    self.conv24 =  nn.Conv2d(512, 512, 3, stride=2, groups=512, padding=1)
    self.bn24   =  nn.BatchNorm2d(512)
    self.conv25 =  nn.Conv2d(512, 1024, 1, stride=1)
    self.bn25   =  nn.BatchNorm2d(1024)
    self.conv26 =  nn.Conv2d(1024, 1024, 3, stride=1, groups=1024, padding=1)
    self.bn26   =  nn.BatchNorm2d(1024)
    self.conv27 =  nn.Conv2d(1024, 1024, 1, stride=1)
    self.bn27   =  nn.BatchNorm2d(1024)

    self.fc1    =  nn.Linear(in_features=1024,  out_features=1000);

    self.avg    =  nn.AvgPool2d(7, stride=1)

    self.relu   =  nn.ReLU()

  # The forward function will be run three times
  # Once for anchor, positive and negative
  def forward(self, x):

    x = self.relu(self.bn1(self.conv1(x)))
    x = self.relu(self.bn2(self.conv2(x)))
    x = self.relu(self.bn3(self.conv3(x)))
    x = self.relu(self.bn4(self.conv4(x)))
    x = self.relu(self.bn5(self.conv5(x)))
    x = self.relu(self.bn6(self.conv6(x)))
    x = self.relu(self.bn7(self.conv7(x)))
    x = self.relu(self.bn8(self.conv8(x)))
    x = self.relu(self.bn9(self.conv9(x)))
    x = self.relu(self.bn10(self.conv10(x)))
    x = self.relu(self.bn11(self.conv11(x)))
    x = self.relu(self.bn12(self.conv12(x)))
    x = self.relu(self.bn13(self.conv13(x)))
    x = self.relu(self.bn14(self.conv14(x)))
    x = self.relu(self.bn15(self.conv15(x)))
    x = self.relu(self.bn16(self.conv16(x)))
    x = self.relu(self.bn17(self.conv17(x)))
    x = self.relu(self.bn18(self.conv18(x)))
    x = self.relu(self.bn19(self.conv19(x)))
    x = self.relu(self.bn20(self.conv20(x)))
    x = self.relu(self.bn21(self.conv21(x)))
    x = self.relu(self.bn22(self.conv22(x)))
    x = self.relu(self.bn23(self.conv23(x)))
    x = self.relu(self.bn24(self.conv24(x)))
    x = self.relu(self.bn25(self.conv25(x)))
    x = self.relu(self.bn26(self.conv26(x)))
    x = self.relu(self.bn27(self.conv27(x)))

    x = self.avg(x)

    x = x.view(-1, 1024)

    x = self.fc1(x)

    return x
```

Training Helper Functions:

The functions below load an array with all possible triplet combinations in the dataset as indices to help save space. The triple loader fetches the images in batches from the training dataset, yeilding a batch at a time.

In [6]:
```
#Code to ensure no duplicates in training and to allow for epochs

def load_indices(dataset):
  '''This function loads training indices to cover all possible training samples'''
  training_indices = []
  samples = [-1, -1, -1]
  for i in range(dataset.__len__()):
    samples[0] = i;
    pos_class = list(range((int(i/51))*51, (int(i/51))*51+51))
    for j in pos_class:
      if (j != i):
        samples[1] = j
        for k in range(0, dataset.__len__()):
          if (not(k in pos_class)):
            samples[2] = k
            training_indices.append([samples[0], samples[1], samples[2]])
  return training_indices



def siamese_triple_loader(dataset, batch_size=1, shuffle=True):
  '''This function generates a minibatch of pos, neg and anchor images'''
  training_indices = load_indices(dataset)

  train_order = list(range(0, (NUM_CLASSES-1)*(PER_CLASS-1)*(PER_CLASS)*(NUM_CLASSES*PER_CLASS)))
  test_counter = 0;
  for epoch in range(EPOCHS):
    if (shuffle):
      random.shuffle(train_order)
    batch = torch.empty(batch_size, 3, 3, input_size, input_size)
    for index in range(0,  len(train_order)):
      for i in range(batch_size):
        batch[i][0], anchor = dataset.__getitem__(training_indices[train_order[index]][0])

        batch[i][1], pos = dataset.__getitem__(training_indices[train_order[index]][1])
        batch[i][2], neg = dataset.__getitem__(training_indices[train_order[index]][2])
        index+=1
      yield batch
```

Evaluation Function

The avgDistance function was intended for use during training and would have evaluated the model every 100 batches. However, due to Google Colab's memory constraints, a different solution, shown in the Evaluation Process code block was used.

In [ ]:
```
def avgDistances(dataset, model):
  '''This function evals the model by calculating the avg distances between negs
  (bigger is better) and the avg distances between positives (smaller better)'''
  check         =  nn.PairwiseDistance()
  count         =  0
  count2        =  0
  sum_same      =  0
  sum_different =  0

  for outer in range(204): #gives 10404 tests
    out1 = (model((eval_dataset.__getitem__(outer)[0].view(1,3,input_size,input_size)).to(device))).
to('cpu')
    for inner in range(0,204,4):
      out2 = (model((eval_dataset.__getitem__(inner)[0].view(1,3,input_size,input_size)).to(device
))).to('cpu')
      if ((int)(inner/51) == (int)(outer/51)):
        sum_same += check(out1, out2)
        count += 1
      elif ((int)(inner/51) != (int)(outer/51)):
        sum_different += check(out1, out2)
        count2 += 1
      else: print('Error: ', outer, inner)

  return sum_same/count, sum_different/count2
```

Necessary Training Variables:

In [7]:
```
# Transforms for the training input images
input_transforms = transforms.Compose([
                            transforms.CenterCrop((500,500)),
                            transforms.Resize((input_size, input_size)),
                            transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue
=(-0.1, 0.1)),
                            transforms.ToTensor()
                                    ])

train_dataset =  dset.ImageFolder(root="./train", transform=input_transforms)
dataloader    =  siamese_triple_loader(train_dataset, batch_size)
LR            =  0.001
model         =  Net().to(device)
criterion     =  nn.TripletMarginLoss(reduction='mean',margin=1, p=2)

model.train()

optimizer     =  optim.SGD(model.parameters(), lr=LR)

# For graphing the losses afterwards
all_losses = torch.empty(1)
```

Training Process:

The batches of triplets are fetched from the customised training dataloader generator. They are passed to the model one batch at a time. First the batch of anchors, the batch of positives and then the batch of negatives. The loss is then calculated using TripletMarginLoss for the entire batch at the same time.

In [ ]:
```
# Training loop, uses the dataloader generator
for i_batch, batch in enumerate(dataloader):

  # Saving the model, even the untrained version
  if (i_batch%100 == 0):
    model_name = str(i_batch)+"_Siamese_Model_2D(1:52PM)"+str(LR)+"_.pt"
    PATH = "./drive/My Drive/"+str(model_name)
    torch.save(model.state_dict(), PATH)

  #Training model
  model.train()
  loss = 0
  optimizer.zero_grad()
  batch = batch.view(3,batch_size,3,input_size, input_size)

  anchors    = batch[0].to(device)
  positives  = batch[1].to(device)
  negatives  = batch[2].to(device)

  anchor_outs   = model(anchors)
  pos_outs      = model(positives)
  neg_outs      = model(negatives)

  # Getting all the loss over the whole batch and performing mean reduction
  loss += criterion(anchor_outs, pos_outs, neg_outs)

  # Printing batch number and loss for that batch
  print('Batch number: ', i_batch+1)
  print('Accumulated loss: ', loss.item())
  print('---------------------------------------')
  print('\n')

  # For graphing post training
  all_losses = torch.cat((all_losses, torch.as_tensor(loss.item()).view(1)), 0)

  # Backprop and adjusting weights
  loss.backward()
  optimizer.step()
```

Plotting the losses:

The loss graph will be quite noisy since we are using batches of size 60 as opposed to the entire training set.

In [ ]:
```
fig=plt.figure(figsize=(19, 12), dpi= 80, facecolor='w', edgecolor='k')

# Skip 1st index when plotting since this is the value from torch.empty(1)
plt.plot(all_losses[1:1000])
plt.show
```

Necessary Evaluation Variables:

For each quicktest, using a subset of the test dataset, the model_name was changed to fetch the correct models from Google Drive.

In [ ]:
```
# Transforms and dataset for evaluation and testing
eval_transforms  =   transforms.Compose([
                                    transforms.CenterCrop((500,500)),
                                    transforms.Resize((input_size, input_size)),
                                    transforms.ToTensor()
                                    ])
eval_dataset     =   dset.ImageFolder(root="./test", transform=eval_transforms)

# Outputs of testing and evaluation
threshold        =  2
check            =  nn.PairwiseDistance()
correct          =  0
false_negative   =  0
false_positive   =  0
avg_distance_sum =  0

# Loading in the trained model from Google Drive
model            =   Net()
model_name       =   "600_Siamese_Model_UsedOldLR(9:20AM)0.001_.pt"
PATH             =   "./drive/My Drive/ConvergedModel/"+str(model_name)

model.load_state_dict(torch.load(PATH))
model.to(device)
model.eval()
```

Evaluation Process:

Due to Google Colab's memory constraints, the models could not be evaluated during training. Therefore, they were saved every 100 batches (6000 Triplets, 18 000 images) and were evaluated for accuracy using a subset of the testing data.

In [ ]:
```
# Full test loop gives 93 636 tests
# (change to range(1, eval_dataset.__len__(), 12) on the outer loop and range(0, eval_dataset.__len_
_(), 6)
# on the inner loop to perform 1 250 quicktests)
for outer in range(0, eval_dataset.__len__()):
  out1 = (model((eval_dataset.__getitem__(outer)[0].view(1,3,input_size,input_size)).to(device))).to
('cpu')

  for inner in range(0, eval_dataset.__len__()):
    out2 = (model((eval_dataset.__getitem__(inner)[0].view(1,3,input_size,input_size)).to(device))).
to('cpu')

    if (check(out1, out2) < threshold and (int)(inner/51) == (int)(outer/51)):
      correct+=1
    elif (check(out1, out2) > threshold and (int)(inner/51) == (int)(outer/51)):
      false_negative+=1
    elif (check(out1, out2) < threshold and (int)(inner/51) != (int)(outer/51)):
      false_positive +=1
    elif (check(out1, out2) > threshold and (int)(inner/51) != (int)(outer/51)):
      correct +=1
    else: print('Error: ', outer, inner)

  print('Eval Accuracy: ', (float)(correct/(correct+false_positive+false_negative)))
  print('Correct: ', correct)
  print('False Positives: ', false_positive)
  print('False Negatives: ', false_negative)
  print('---------------------------------------------')
```

Plotting the accuracies:

The code below just plots the accuracies which were collected from the previous code block. These are for the quick tests which compare 1250 pairs from a possible 93 636 pairs in the test dataset. The batch numbers are the number of batches each trained model had seen.

In [67]:
```
fig        =  plt.figure(figsize=(19, 12), dpi= 80, facecolor='w', edgecolor='k')
accuracies =  [0.167430814479638, 0.865007541478129, 0.875565610859728, 0.89894419306184,
               0.90497737556561, 0.902714932126696, 0.924585218702865, 0.919306184012066,
               0.920060331825037, 0.917043740573152, 0.895927601809954]
batches    =  [0, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]

plt.plot(batches, accuracies)
plt.show()
```

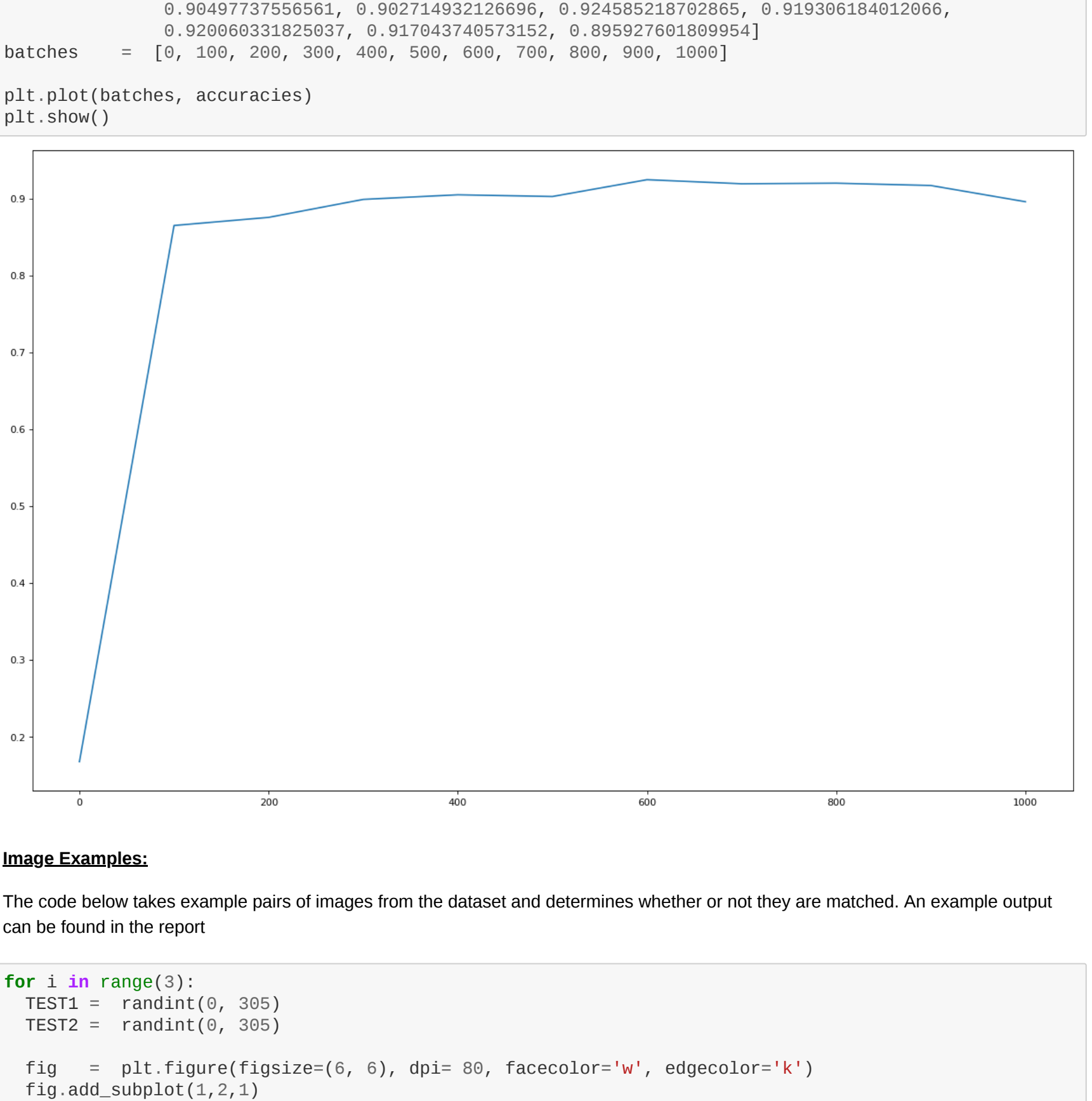


Image Examples:

The code below takes example pairs of images from the dataset and determines whether or not they are matched. An example output can be found in the report

In [ ]:
```
for i in range(3):
  TEST1 =  randint(0, 305)
  TEST2 =  randint(0, 305)

  fig   =  plt.figure(figsize=(6, 6), dpi= 80, facecolor='w', edgecolor='k')
  fig.add_subplot(1,2,1)
  plt.imshow(eval_dataset.__getitem__(TEST1)[0].permute(1,2,0))

  fig.add_subplot(1,2,2)
  plt.imshow(eval_dataset.__getitem__(TEST2)[0].permute(1,2,0))

  out1 = model((eval_dataset.__getitem__(TEST1)[0].view(1,3,input_size,input_size)).to(device))
  out2 = model((eval_dataset.__getitem__(TEST2)[0].view(1,3,input_size,input_size)).to(device))
  eval_dist = nn.PairwiseDistance()

  if (eval_dist(out1, out2).item() > 2):
      print("\t\t      NO MATCH")
  else:
      print("\t\t        MATCH")

  plt.show()
```

Checking Average Distances:

The code to measure average distances is shown below. This tests the average distance between the 1000 point vectors output by the net for images in the same class. The code can be changed as explained by the comments below in order to measure average different-class distances. The different class measurement simply measures the distances between outputs of the net from one class to another. It's performed in this way in order to match the number of measurements made for checking the same class distances. Both averages are therefore calculated for 1275×6 measurements

In [ ]:
```
check            =  nn.PairwiseDistance()
count            =  0
avg_distance_sum =  0

# Loop range can be changed to measure distances between images in different classes:
# outer in range(51), inner in range(51, 102, 2)
for outer in range(51): #gives 1275 tests
  out1 = (model((eval_dataset.__getitem__(outer)[0].view(1,3,input_size,input_size)).to(device))).to
('cpu')
  for inner in range(0, 51, 2):
    out2 = (model((eval_dataset.__getitem__(inner)[0].view(1,3,input_size,input_size)).to(device))).
to('cpu')
    avg_distance_sum += check(out1, out2).item()
    count+=1

# outer in range(51, 102), inner in range(102, 153, 2)
for outer in range(51, 102): #gives 1275 tests
  out1 = (model((eval_dataset.__getitem__(outer)[0].view(1,3,input_size,input_size)).to(device))).to
('cpu')
  for inner in range(51, 102, 2):
    out2 = (model((eval_dataset.__getitem__(inner)[0].view(1,3,input_size,input_size)).to(device))).
to('cpu')
    avg_distance_sum += check(out1, out2).item()
    count+=1

# outer in range(102, 153), inner in range(153, 204, 2)
for outer in range(102, 153): #gives 1275 tests
  out1 = (model((eval_dataset.__getitem__(outer)[0].view(1,3,input_size,input_size)).to(device))).to
('cpu')
  for inner in range(102, 153, 2):
    out2 = (model((eval_dataset.__getitem__(inner)[0].view(1,3,input_size,input_size)).to(device))).
to('cpu')
    avg_distance_sum += check(out1, out2).item()
    count+=1

# outer in range(153, 204), inner in range(204, 255, 2)
for outer in range(153, 204): #gives 1275 tests
  out1 = (model((eval_dataset.__getitem__(outer)[0].view(1,3,input_size,input_size)).to(device))).to
('cpu')
  for inner in range(153, 204, 2):
    out2 = (model((eval_dataset.__getitem__(inner)[0].view(1,3,input_size,input_size)).to(device))).
to('cpu')
    avg_distance_sum += check(out1, out2).item()
    count+=1

# outer in range(204, 255), inner in range(255, 306, 2)
for outer in range(204, 255): #gives 1275 tests
  out1 = (model((eval_dataset.__getitem__(outer)[0].view(1,3,input_size,input_size)).to(device))).to
('cpu')
  for inner in range(204, 255, 2):
    out2 = (model((eval_dataset.__getitem__(inner)[0].view(1,3,input_size,input_size)).to(device))).
to('cpu')
    avg_distance_sum += check(out1, out2).item()
    count+=1

# outer in range(255, 306), inner in range(0, 51, 2)
for outer in range(255, 306): #gives 1275 tests
  out1 = (model((eval_dataset.__getitem__(outer)[0].view(1,3,input_size,input_size)).to(device))).to
('cpu')
  for inner in range(255, 306, 2):
    out2 = (model((eval_dataset.__getitem__(inner)[0].view(1,3,input_size,input_size)).to(device))).
to('cpu')
    avg_distance_sum += check(out1, out2).item()
    count+=1

print("AVG SAME DISTANCE: ", avg_distance_sum/count)
```