EE981 - Image and Video Processing

Image Segmentation Using Machine Learning and Deep Learning

Laura Parra Garcia - 201989845
Richard Gilchrist - 201983648
Chavinpat Naimee - 201976778

Date: 19/04/2020

University of Strathclyde
Department of Electronic & Electrical Engineering
Lecturers: Dr. Jinchang Ren & Dr. Vladimir Stankovic

Contents

List of Figures

List of Tables

1 Introduction & Objectives

Computer vision (CV) is defined as an interdisciplinary research field involving the development of methods that allow computers “see” and understand the content of digital images. From the engineering perspective, it attempts to automate functions that can be done by the human visual system. Computer vision tasks aim to extract, process and analyze information from images, which may include object segmentation and detection; semantic text description and searching image content. Understanding the content of digital images is the important goal of computer vision.
Image segmentation plays an important role in many applications. Some of these applications are: surveillance, recognition (facial, iris, fingerprint), medical imaging, object detection (indispensable for autonomous cars to work) and target detection and tracking, among many others.

The aim of this project was to research and implement image segmentation using machine learning and deep learning techniques. The objectives of this projects are:

- In-depth literature review of the state-of-the-art image segmentation. A thorough study of image segmentation was conducted and state-of-the-art techniques were reviewed in order to familiarize ourselves with the topic and decide the path our work should follow.
- From a more general point of view, we investigated deep neural network architectures and their suitability for the task of image segmentation. We discuss two different approaches to segmentation; unsupervised and supervised methods. We provide an outline of some segmentation techniques, detail the methods implemented and present the results obtained.
- Hierarchical image segmentation was investigated using two examples where, by varying parameters such the number of clusters, different results were obtained and commented upon. The approach used for hierarchical image segmentation was agglomerative clustering and the examples were based on a grayscale image of a number of coins, as well as images from a dataset called Berkeley Segmentation Data Set (BSDS500) [1].
- Implementation of a deep learning model based on transfer learning using a pre-trained model. For the deep learning approach we used a model called U-Net and the pre-trained encoder applied was MobileNetV2. The dataset used was Oxford-IIIT Pets dataset, which is divided into two sub-sets: one set of images used for training and the rest used for testing.
- Implementation of the same deep learning model, U-Net, being fully trained. Once we had the results of the transfer learning approach, we decided to fully train the network to compare the results. This resulted being much more time and resource consuming but, at the end, results were more accurate.
- Data augmentation was applied through the U-Net model as it performs elastic modifications to the available images, leading to more image data being available to work with.
- An investigation of the effects of model parameters on the resulting segmentation is also presented, considering the results subjectively and using objective metrics. Four performance metrics are calculated varying the corresponding hyperparameters in the model. The performance was measured by calculating the precision, recall, f-score and accuracy of the model and results were presented and commented.

2 Background and State-of-the-art

Image segmentation can be divided into semantic segmentation and instance segmentation for a first approach. Semantic segmentation is a more general way of dividing the image as pixels will be classified in different classes. Instance segmentation digs a little deeper into the classification task. Pixels are divided into different classes based on affinity among them and classification will also differentiate objects that belong to the same class [2]. There are conventional solutions such as thresholding, edge detection or region growing, among others. Artificial intelligence and deep learning have advanced rapidly in the past few years in the field of image segmentation, leading to outstanding results that could not be obtained before.

From the perspective of machine and deep learning segmentation, two main approaches may be defined: unsupervised and supervised methods. Unsupervised methods segment an image without having any prior knowledge of the 'ground truth' segmentation. This result depends on statistical measures on the segmented image. Examples of unsupervised image segmentation methods include k-means clustering algorithm, thresholding and expectation maximization algorithm (EM) [3]. Supervised methods learn by example, with training requiring many instances of images and associated ground truth segmentations. This ground truth is known in the case of synthetic images, while for real images it requires manual annotation [4], [5], [6].

The key to image segmentation is to identify the problem that needs to be addressed. Once the problem is clear, the choice for the segmentation technique will be easier to make. Depending on the application, many approaches to building image segmentation are currently known to be state-of-the-art. Some examples are:

- Fast Fully Convolutional Networks: Fully convolutional networks are made of locally connected layers and can work with inputs of any size. They have two main components: the downsampling module, where the images are downsampled through convolutions or pooling layers retrieving contextual and semantic information and reducing the resolution of the final feature map. The upsampling part, which is used to recover the fined-grained spatial information lost due to the downsampling part. The upsampling module is formed by dilated convolutions that are very time and memory consuming. To cope with this, the fast fully convolutional networks have been implemented leading to state-of-the-art results in Pascal context dataset and reducing the running time three times respecting the FCN. This method implements the upsampling module as a joint pyramid upsampling (JPU) that extracts high-resolution feature maps (without losing performance) and a FCN as the backbone [7].
- The One Hundred Layers Tiramisu: Fully Convolutional DenseNets for Semantic Segmentation. DenseNets are Densely Connected Convolutional Networks that have proven to be specially successful for image classification. The idea behind this model is to extend DenseNets to deal with semantic segmentation while reducing the parameters and without extending postprocessing or pretraining. It reached state-of-the-art results in urban scenes [8].
- Gated Shape CNNs: this model presents a two-stream CNN architecture approach where shape information is retrieved as a separated processing stream that works in parallel with the classical stream. The key to this approach is the interconnection of the intermediate layers through a new type of gates. This solution outperforms DeepLab3 on CityScapes benchmarks (2% in terms of mask, mIoU, and 4% in terms of boundary, F-score) getting sharper prediction on edges and enhancing performance on small objects [9].
- U-Net: End-to-end fully convolutional network widely used for medical purposes. Training is performed using the input pictures, their ground truth and a stochastic gradient descent implementation of Caffe. The architecture relies on a multi-stage cascade CNN that takes the regions and make predictions. As it extracts low level features continuously, sometimes it uses excessive use of computational resources [10]. The U-Net architecture was used for our deep learning implementation and it is explained in more detail later on.

3 Methodology

3.1 Hierarchical Image Segmentation Using Agglomerative Clustering

Image segmentation divides an image into homogeneous regions or objects according to a perceptual element, such as colour tone homogeneity. Hierarchical segmentation algorithms evaluate the image at many different scales of measurement. Its result is not a single partition, but rather a hierarchy of regions or structures that capture various partitions for different levels of study.

Agglomerative clustering is an unsupervised technique of hierarchical clustering where each point is initially put in a group on its own and then the group will be joined by the most similar attributes based on a bottom-up approach [11]. The distance of data is calculated by the distance between observations of pairs of groups. One disadvantage of the agglomerative clustering method is that if points are incorrectly grouped during the early stages, they cannot be re-assigned later [12].

In this work, the agglomerative clustering was implemented in Python using the Scikit Learn library. In Scikit Learn, the linkage criterion is used to minimise the distance between data and connectivity defines the structure of the data that pixels are connected to their neighbours. In this study, 'Ward linkage' will be used as a linkage parameter. 'Ward' is a variance-minimising approach to minimise the sum of squared distance within all clusters. It generally finds segments of roughly equal size [13]. For the connectivity, the grid_to_graph() function will be used as the feature extraction function for the image.

To implement the agglomerative clustering technique, in the first case, an image from a Scikit Learn example [14] was used. Additionally, hierarchical segmentation was investigated using images from the BSDS500 dataset, in order to show the results obtained by using different numbers of clusters. The BSDS500 dataset contains original images along with human-generated annotations that can be used to assess the performance [1].

3.2 Deep Learning

The dataset used to investigate the deep learning approach was the Oxford-IIIT Pets dataset [15]. This dataset consists of 37 different classes, corresponding to 37 different breeds of pet, with around 200 images for each class. Within the dataset, the images exhibit variation in scale, the pose of the animal and the lighting conditions. All images have a ground truth annotation of breed, a head bounding box for the animal, and a tri-map pixel level segmentation. This means that pixels belonging to the pet itself are assigned a value of 0, the border or outline of the pet is assigned a value of 1 and a value of 2 is assigned to any pixel that does not meet the criteria for the above classification i.e background and surroundings. The instances in the dataset were already split into training and testing sub-sets, with 3,669 images for testing and 3,680 for training. Figure 1 shows an example of an instance in the dataset, with the original image, the bounding box and the ground truth segmentation.

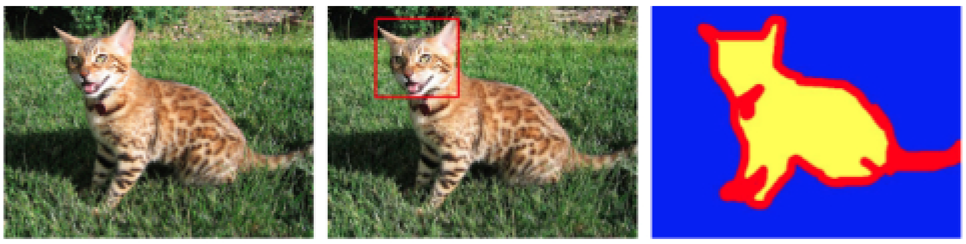


Figure 1: Example image with annotation and segmentation

The model that was used is called a U-Net. Originally developed for image segmentation in the context of biomedical imaging, it has a symmetrical architecture composed of an encoder and a decoder [16]. An approach known as transfer learning was adopted, which involves the use of a model that has previously been trained for a particular task, typically on a large dataset. The pre-trained model is then modified for application to another set of data. The advantages of transfer learning include a reduction in the requirements of dataset size and computational resources, while a disadvantage is that the target problem must be similar to the initial problem in order to achieve good results.

The encoder used in this case was a pre-trained model, MobileNetV2, as described in [17]. The model was trained for the image segmentation task using the PASCAL VOC dataset. During training, the aim is to learn sets of convolutional filter weights that extract features from the input which prove useful for correctly classifying the input data. In a transfer learning application, some of these filter weights are 'frozen', so the number of trainable parameters is reduced, thus it is faster and less computationally expensive to train. MobileNetV2 is a neural network from the MobileNets family from Google that can be used in machines with limited computing power, providing accuracy with as little memory and computing power as possible. This pre-trained model is already implemented and ready to use in TensorFlow [18], with some modifications and additions being made in this work. There are three main components to MobileNetV2:

- Depthwise separable convolutions: they split convolutions into two; a depthwise convolution that convolves filtering with the input channels and a pointwise convolution that sums (in a weighted way) the convolution outputs. These weighted sums are the key for the depthwise separable convolutions to get more characteristics from fewer parameters.
- Width multiplier: this is a hyperparameter that reduces the total number of parameters in the model to lower the computational cost as needed. Only the important features will be retained and the less important will be discarded.
- Linear bottlenecks and shortcut connections: linear bottlenecks are used between the layers so no information is lost and shortcut connections assure that the training will be faster and accuracy better.

The part of the decoder for the U-Net model is made of transposed convolutions that give the information about the localization of the segmented objects in the picture. U-Net is an end-to-end fully convolutional network (FCN) and so it can work with any image size, unlike a CNN with a fully-connected layer which requires input of fixed size. For the training phase, the model optionally performs a data augmentation step, introducing some probability with which an image will be flipped horizontally. Other transformations such as scaling, rotation or translation may be applied in order to artificially increase the variation seen within the training examples, in order to try to improve the ability of the model to generalise [16].

The hyperparameters associated with the network were investigated and the effect on the segmentation performance when varying these hyperparameters was examined. The hyperparameters of interest were the following:

- Epochs: this is a hyperparameter of stochastic gradient descendent (SGD), it controls the number of iterations for the whole training set. In each epoch, the model updates the internal parameters until convergence.
- Batch size: the batch size is a hyperparameter of SGD. When the data is too big, it is separated into batches so multiple batches will be processed in each epoch. In this work, a range of batch size values were implemented.
- Steps per epoch: is the number of batches needed to complete an epoch. As an example, if the batch size is 64 and there are 3680 examples in the training set, the number of steps per epoch will be 57.
- Learning rate: controls the speed at which the model will learn during training. The weights update is scaled by the learning rate parameter during the training phase. Stochastic gradient descent (SGD) is a commonly-used optimisation algorithm for training neural networks. The error gradient is computed for each state of the network (at the end of each batch) using the images in the training dataset. Then the weights of the model are updated using backpropagation [19]. This work used a variation of SGD, known as Adam, which adopts an adaptive learning rate approach [20].

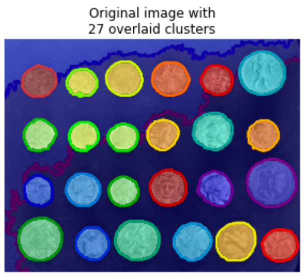
Keras callbacks were utilised during model training such that if the validation accuracy did not improve for some number of consecutive epochs, the training would be stopped earlier than the prescribed number of epochs. Additionally, by evaluating the validation set accuracy, a callback was used to save the best model to file so this model could easily be loaded in future. In this work, the suitability of the transfer learning approach for this particular dataset and task was investigated. The model was trained with and without the pre-trained encoder weights, in order to assess the results in each case and compare the results.

4 Results

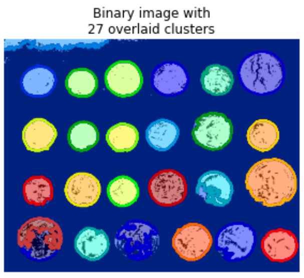
4.1 Hierarchical Segmentation Results

A grayscale image of a number of coins was used to assess the performance of an agglomerative clustering method for image segmentation. The original image is shown in figure 2a, with figures 2b and 2c showing the resulting segmentation in the form of segment contours and a segmentation mask.

(a) Original grayscale image of coins



(b) Original image with cluster contours and mask



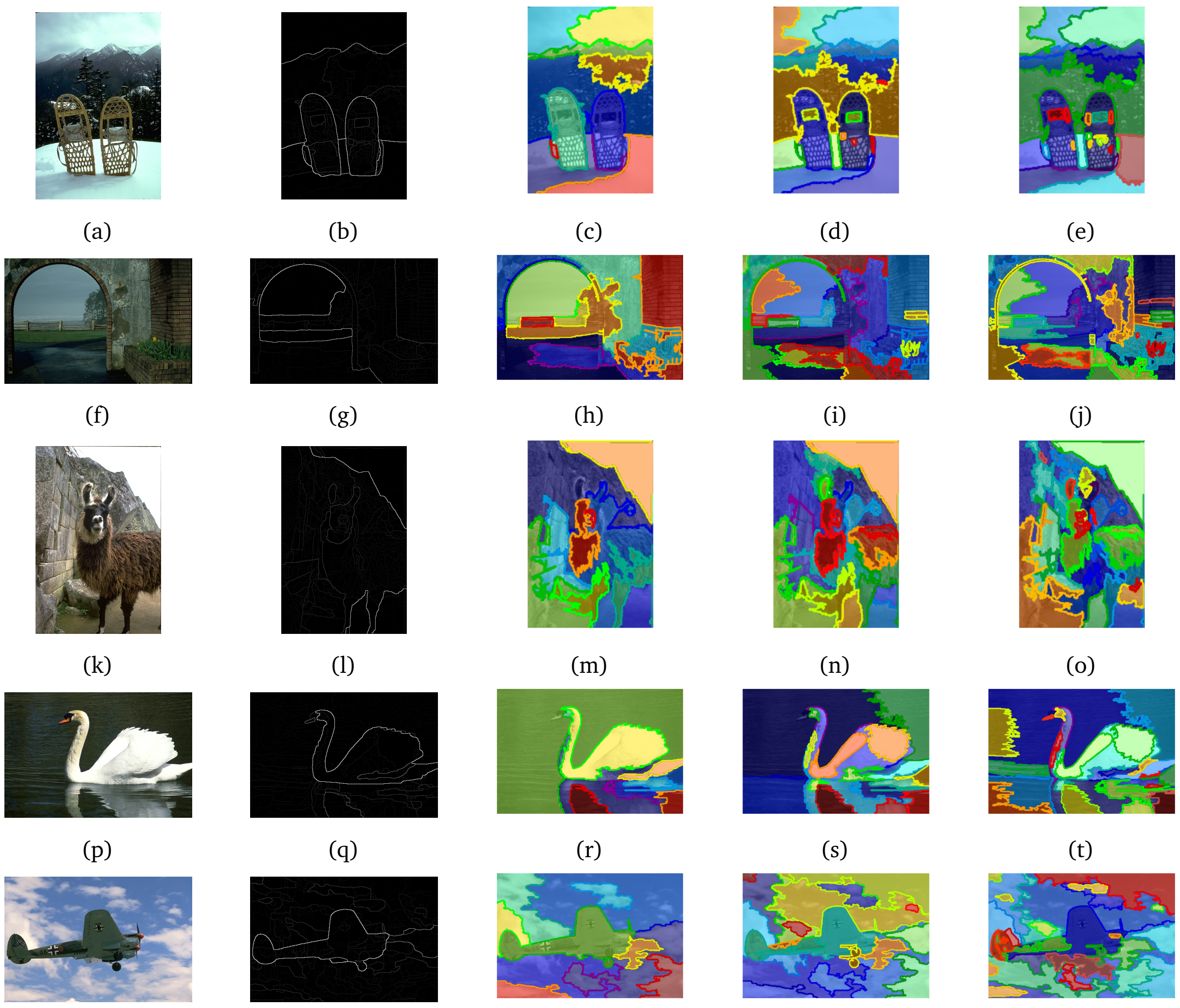
(c) Binary image with cluster contours and mask

Figure 2: Original image of coins and resulting segmentation

In order for each of the 24 coins to be properly segmented, it was determined that 27 clusters were required. It can be seen in figure 2b that despite the coins being segmented very well, the background is segmented into three distinct sections as a result of the variation in background pixel intensity values. In order to try to more effectively isolate the coins from the background, binary thresholding was applied with a threshold value of 127. In the resulting segmentation of the binary image, shown in figure 2c, it can be seen that almost all of the background pixels are correctly assigned to one cluster. While most of the coins are still segmented reasonably well, two notable exceptions are seen in the bottom-left of the image. In the binary image, the coins for which segmentation was poor contained a large number of black pixels. Lowering the threshold value helps, but in this specific case, has a detrimental effect on the segmentation of the background.

The agglomerative clustering method was also implemented to segment some images from the BSDS500 dataset. In this dataset, ground truth segmentations were provided but these were only used for visualisation and were not involved in the clustering process. Different numbers of clusters were used to generate hierarchical image segmentations using agglomerative clustering. Figure 3 shows several examples of images, their ground truth segmentations and segmentations resulting from using 10, 20 and 30 clusters.

It can be seen that each segmentation image usually becomes more refined when the number of clusters increases. For example, figure 3w shows a picture of an aeroplane, the segmentation with 10 clusters of this image provides the body of the plane segmented in green colour, which is only one cluster. The green cluster from figure 3w is considered to be the foreground of the and the rest is the background of the picture. In the segmentation with 30 clusters, the foreground image of figure 3y, which is the whole plane, was divided into around 5 parts. Some parts of the plane could be segmented, such as pattern and landing gear of the plane. These results can be considered as a hierarchical chart of segmentations. However, when the number of clusters is too high, the result would show some clusters that are not important or too many clusters that should be all combined in only one cluster.



(a) (b) (c) (d) (e)

(f) (g) (h) (i) (j)

(k) (l) (m) (n) (o)

(p) (q) (r) (s) (t)

(u) Original image (v) Ground truth segmentation (w) Segmentation with 10 clusters (x) Segmentation with 20 clusters (y) Segmentation with 30 clusters

Figure 3: Hierarchical segmentation using agglomerative clustering with different clusters

4.2 Deep Learning Results

The metrics used to evaluate model performance were accuracy, precision recall and F-score, as defined in appendix B. Since predictions are made on the pixel level and this is a multi-class problem, the average values of the performance metrics are quoted from here on. The model performance was first evaluated by adopting the transfer learning approach of using the pre-trained encoder weights. Then, for comparison, the whole model was trained from scratch using the pets dataset, and the results evaluated. Figure 4 shows an example of how the predictions typically change with training.

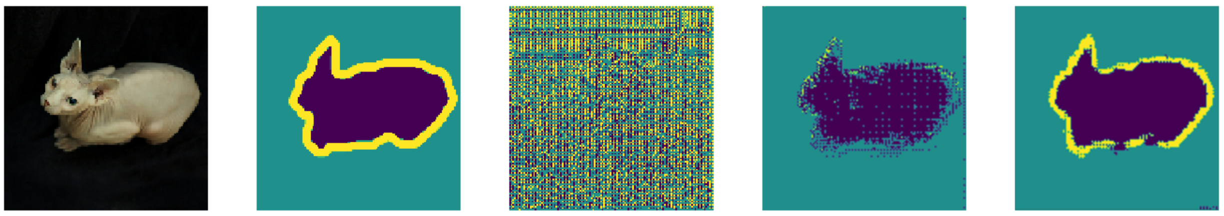

(a) Original image (b) Ground truth segmentation mask (c) Prediction from untrained model (d) Prediction after 3 epochs (e) Prediction after 50 epochs

Figure 4: Model predictions with progression of training (transfer learning)

Figure 5 shows examples of good and bad predictions from the test set, output from the same model, using the pre-trained encoder.

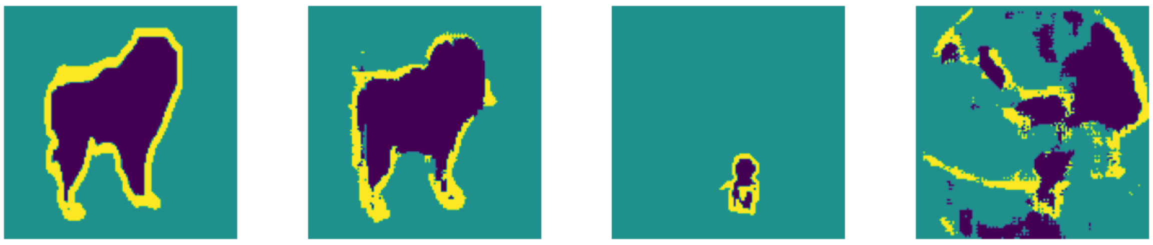

(a) Ground truth segmentation (b) Predicted segmentation (c) Ground truth segmentation (d) Predicted segmentation

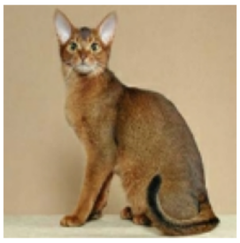
Figure 5: An example of good and bad segmentation predictions (transfer learning)

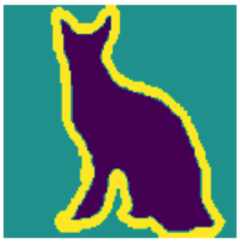
The full tables of results are displayed in appendix C, having been omitted here for brevity. The results show the performance achieved by the model using different hyperparameter values. Firstly, results were obtained by training only the decoder component of the model, using the pre-trained encoder weights. Secondly, results were obtained by training the whole model using the pets dataset. As we can see from table 1 the values for accuracy, precision, recall and f-score are lower for the bad prediction results and higher when prediction gets better.

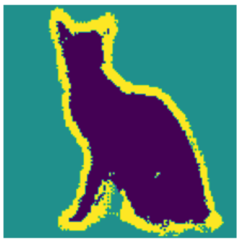
	Learning rate	Batch size	Epochs	Accuracy	Precision	Recall	F-score
Good prediction in figure 5b	1.00E-5	26	20	0.924	0.853	0.815	0.831
Bad prediction in figure 5d	1.00E-6	64	20	0.697	0.381	0.721	0.359

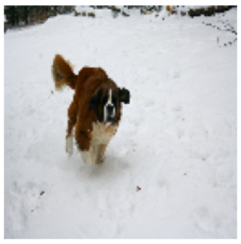
Table 1: Performance metrics for the predictions shown in figures 5b and 5d

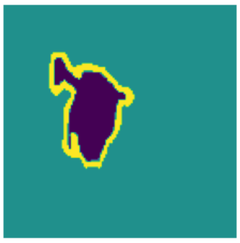
Figure 6 shows some examples of images from the testing dataset, along with the associated ground truth and predicted segmentations. These predictions were generated using the model that did not include the pre-trained encoder weights, thus it was trained fully using the pets dataset. The values for accuracy, precision, recall and f-score associated with these examples can be seen in appendix B in table 3

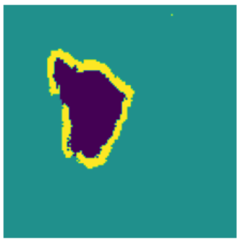

(a) Original image

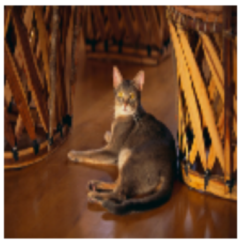

(b) Ground truth segmentation

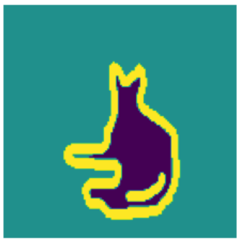

(c) Predicted segmentation

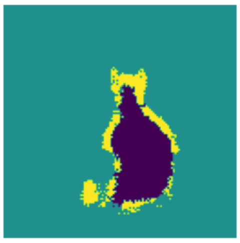

(d) Original image

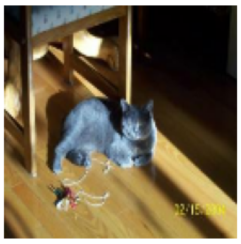

(e) Ground truth segmentation

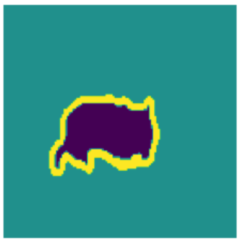

(f) Predicted segmentation

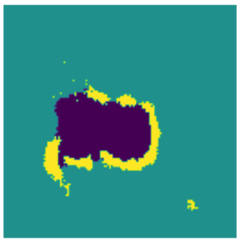

(g) Original image

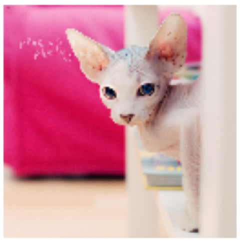

(h) Ground truth segmentation

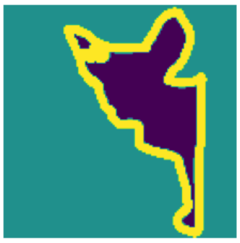

(i) Predicted segmentation

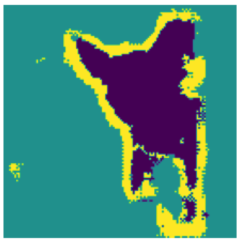

(j) Original image

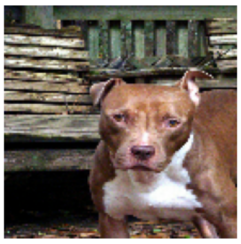

(k) Ground truth segmentation

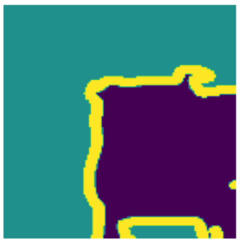

(l) Predicted segmentation


(m) Original image


(n) Ground truth segmentation


(o) Predicted segmentation


(p) Original image


(q) Ground truth segmentation

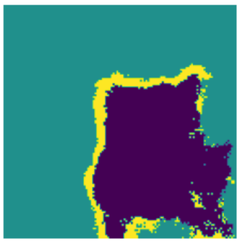
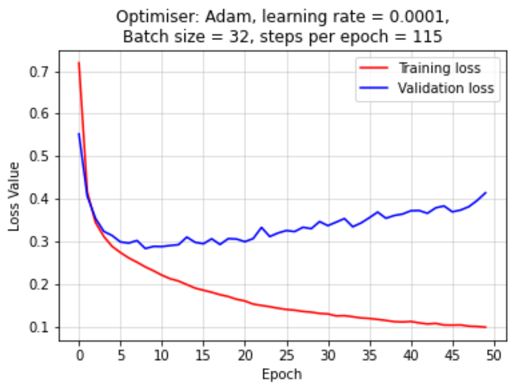

(r) Predicted segmentation

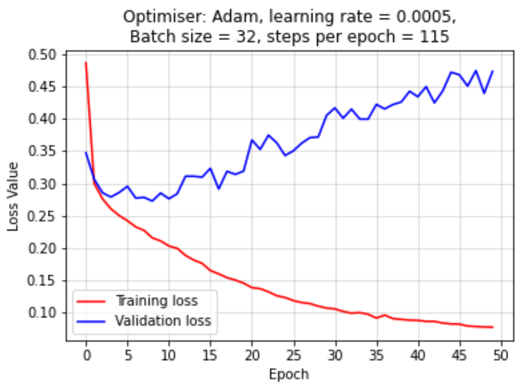
Figure 6: Examples of segmentation predictions from whole model trained

In figure 6, it can be seen that the predicted segmentations are visually acceptable, in general. Some interesting cases are illustrated. The first two images show the pet with a simple background and the predicted segmentation is good, with the pet being effectively identified and separated from the background. The images in 6g and 6j show the pet in a scene with a complicated background or with shadows present. The prediction is generally acceptable, with the border pixels being noticeably poorly predicted. Finally, when the pet is incomplete or partially occluded, the pet is well separated from the background, but the segmentation is often incomplete or lacking detail. Interestingly, in figure 6o, the cat's ear is present, despite being erroneously marked as border pixels in the ground truth segmentation.

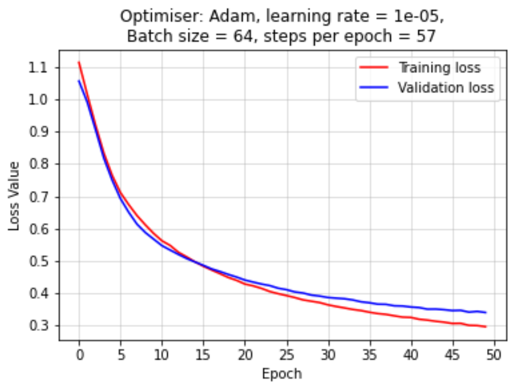
In figure 7, we can see the validation loss and the training loss obtained from the transfer learning approach for different sets of hyperparameters. Figures 7a and 7b show examples of overfitting occurring during the training of the model, identified by the validation loss increasing while the training loss continues to decrease. By comparing this result with the plots obtained for the rest of the hyperparameter combinations, it can be seen that the parameter that leads to overfitting is a high learning rate. Figure 7c shows the training and validation loss varying as training progressed for 50 epochs, using the model with pre-trained encoder weights. It can be seen that by 50 epochs, the training and validation loss curves both seem to still have a downward trajectory, suggesting that increasing the number of epochs could reduce the validation error further. Figure 7d shows the training and validation loss for 100 epochs where the whole model was being trained on the pets dataset. It can be seen that towards the end of the training period, the validation loss curve has flattened out, suggesting further training would not improve the performance on the validation set.



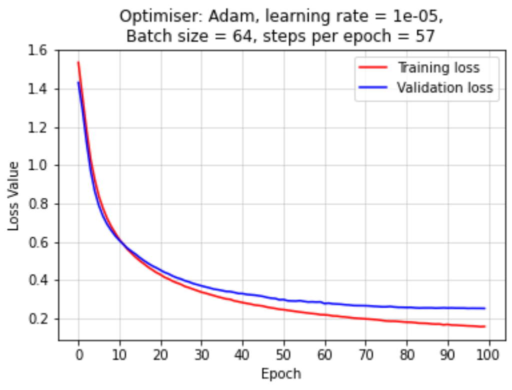
(a) Using pre-trained encoder



(b) Using pre-trained encoder



(c) Using pre-trained encoder



(d) Training the full model

Figure 7: Training and validation loss for different parameter combinations

5 Discussion

The aim of any image segmentation is to separate an image into reasonable units. However, it is difficult to indicate what is the proper logical segments. Thus, image segmentation techniques in both supervised and unsupervised are discussed below. In hierarchical segmentation, agglomerative clustering method was chosen to implement with coins image and sample images of BSDS500 dataset. This method was selected because it is one of the most powerful techniques for an unsupervised problem, which means it is able to segment images without corresponding output values (ground truth). A grayscale picture of coins was used to compare the resulting segmentation of the original grayscale image and binary image. Both images were segmented reasonably well, although the original one seems to segment coins better, but almost all of the background pixels are assigned to one segment for the binary image that contained a large number of black pixels.

The main results of the agglomerative clustering method were obtained from samples from the BSDS500 dataset. There are several hyperparameters can be adjusted, i.e. linkage criterion, affinity, number of clusters, and so on. For linkage criterion, it has 4 types of linkage, which are 'ward', 'complete', 'average' and 'single'. For this task we used 'Ward linkage' to implement the clustering because Scikit Learn recommends it as an excellent linkage for image segmentation. However, other linkage criteria were used for experimentation, although they resulted in poor performance for this specific case. Ward minimises the variance by minimising the sum of squared distances between items within clusters. Then, if the linkage is 'Ward', only 'Euclidean' distance is to be used for affinity parameter.

In Scikit Learn, there are two ways to specify the number of clusters. First, the number of clusters can be decided directly by designating it or second, it can be set by a distance threshold. In this task, the number of clusters was varied to 10, 20 and 30 clusters to observe the different results. These resulting segmentation can be considered as hierarchical image segmentation. As a result, each segmentation image usually becomes more refined when increasing the numbers of clusters. Nevertheless, when the numbers of clusters are too high, a result would demonstrate some clusters that are not important or too many clusters in an image that can be shown in a fewer number of clusters.

In order to evaluate the performance of the deep learning models, a number of different values for the learning rate, batch size and number of epochs were used. For each combination of these three parameters, the average precision, recall, accuracy and F-score were computed for the training and testing datasets. While the hyperparameter values were chosen manually in this case, there are a number of more methodical approaches which could have been adopted including grid search, random search and Bayesian optimisation [21]. Such methods, especially grid search, are computationally expensive and it was not practically feasible to methodically explore the hyperparameter space in this work, due to restraints related to time and computational resources.

Beyond the selection of hyperparameters such as learning rate and batch size, optimisation techniques may be used to determine a network structure and configuration that improves the performance for a given task. In this work, the network used had a fixed structure and it was trained with and without the pre-trained encoder weights. Further work could involve the development of a new network structure. Since it is not clear how to identify a particular network structure that will offer significant performance improvement [22], the network structure and layer configuration could be determined by utilising hyperparameter tuning techniques.

Tables 2 and 3 from appendix B show that training the model from scratch, rather than using the pre-trained encoder, resulted in improved performance for both training and testing data for every combination of hyperparameters investigated. This suggests that for this segmentation task using this particular dataset, the transfer learning approach produces slightly inferior results. This was also evident upon visual inspection of the results. In general, the transfer learning approach did not cope well with separating the pet from complicated backgrounds.

The results show that the model was consistently overfitting slightly, with the performance on the test data consistently poorer than the performance on the training data. Introducing dropout regularisation or adding additional data augmentation steps during training may have helped to reduce this overfitting.

6 Conclusion

For this project we investigated image segmentation and the different techniques that are used to implement it as well as applications of it. We found this topic to be very interesting and realised how many things image segmentation can be useful for. Our research included the state-of-the-art techniques and their architectures, for a better understanding of how this task is being addressed at the moment. We decided to implement two different methods, one using machine learning and another one using deep learning.

For implementing a hierarchical image segmentation we opted for a machine learning approach, using agglomerative clustering. Different numbers of clusters were used to segment the images and thresholding was applied to better separate the background from the objects of interest. We noticed that the thresholding decreased the quality of the segmentation for the objects in the foreground but worked very well for assigning the background an only cluster. The images were fairly well segmented taking into account that this is an unsupervised method with no ground truth reference.

For the deep learning approach, we worked with two versions of the same architecture. In the first case, we worked with a pre-trained encoder for the U-Net model. For the second approach, we fully trained the model from scratch using the pets dataset. This gave us an interesting insight into the model and allowed us to compare the performance between both versions. We found that the results were better when the model was fully trained, but the drawback was that it took quite a long time to be trained. Another aspect to take into account was the decision we made about the hyperparameters; while a range of hyperparameter combinations were used for training and testing the network, a more methodical approach utilising techniques such as grid search or random search would have been preferred. Due to the time and computational resources required for such investigation, it was not practically feasible to identify the optimal combination of hyperparameter values.

Future research could be related to implementing the U-Net model with another dataset that contains images with more than one object in them, or conducting a comparison with different CNN architectures. Also, as results have proven that training the whole network gives a better outcome than using the pre-trained encoder, an interesting research would be a way to fully train the network in an efficient way that will not take as much time and computational resources. Regarding hierarchical segmentation, the implementation of the proposed method using deep learning techniques would also be an interesting approach for future work. Once the result of segmentation is obtained, parents in each layer will be the background while the children will be the foreground. Foreground regions will be used as a new image and passed through the deep learning model again. Another topic of study may focus on the application of image segmentation and recognition for autonomous vehicles, which is a very relevant topic nowadays.

A Project Deliverables

The deliverables from the project are summarised below. Commented code for each implementation is included in appendix D and the .ipynb files have been included with the submission.

- Python code of agglomerative clustering: the code document for implementing hierarchical image segmentation using agglomerative clustering.
- Python code of deep image segmentation: the code document for implementing segmentation using deep learning method.
- Deep image segmentation report: this explains about image segmentation in supervised and unsupervised techniques and contains the summarise and results of our work.

B Performance Metric Definitions

- Precision: depends on the total number of examples that have been correctly classified and the total number of examples that have been predicted positive. The higher the precision the higher the chance for a sample classified as positive to actually be positive. The equation followed can be seen in equation 1, where TP is the number of true positives and FP is the number of false positives [23].

$$\text{Precision} = \frac{TP}{TP + FP} \tag{1}$$

- Recall: this value is the result of counting the number of positive class predictions out of all the positive samples in the dataset. High recall implies low precision and vice-versa. A high recall means that the number of false positive is high even though the majority of the positive samples are well classified. This can be seen in equation 2, where FN is the number of false negatives [23].

$$\text{Recall} = \frac{TP}{TP + FN} \tag{2}$$

- F-score: it is used to balance both precision and recall in one value an it follows equation 3,

$$\text{F-score} = (1 + \beta^2) \left[\frac{Precision \cdot Recall}{(\beta^2 \cdot Precision) + Recall} \right], \tag{3}$$

  where *scikit-learn* uses by default a β value of 1, assigning equal weight to precision and recall, resulting in equation 4, [23].

$$\text{F-score} = 2 \left[\frac{Precision \cdot Recall}{Precision + Recall} \right] \tag{4}$$

- Accuracy: it is the overall performance of the prediction an it is given by equation 5, where TN is the number of true negatives [23].

$$\text{Accuracy} = \frac{TN + TP}{TN + TP + FN + FP} \tag{5}$$

C Deep Learning Results Tables

Table 2 shows the results obtained by training the decoder component of the model, with the pre-trained encoder weights.

	Learning rate	1.00E-6	1.00E-5	1.00E-5	1.00E-4	5.00E-4	1.00E-5	1.00E-5	1.00E-5
	Batch size	64	16	32	32	32	64	64	26
	Epochs	20	50	50	50	50	50	100	20
Train	Precision	0.385	0.797	0.801	0.831	0.869	0.787	0.795	0.857
Test	Precision	0.381	0.790	0.791	0.770	0.794	0.765	0.769	0.853
Train	Recall	0.734	0.785	0.778	0.826	0.840	0.744	0.776	0.821
Test	Recall	0.721	0.734	0.785	0.766	0.771	0.735	0.759	0.815
Train	Accuracy	0.703	0.778	0.775	0.821	0.847	0.745	0.772	0.933
Test	Accuracy	0.697	0.764	0.754	0.752	0.768	0.723	0.746	0.924
Train	F-score	0.365	0.885	0.889	0.910	0.924	0.875	0.893	0.844
Test	F-score	0.359	0.870	0.873	0.862	0.881	0.853	0.858	0.831

Table 2: Results obtained using the pre-trained encoder

Table 3 shows the results obtained by training the whole model using the pets dataset. Less results were obtained as training the whole model was very time and resource consuming.

	Learning rate	1.00E-05	1.00E-05	1.00E-04	5.00E-04	1.00E-05	1.00E-05
	Batch size	16	32	32	32	64	64
	Epochs	50	50	50	50	50	100
Train	Precision	0.885	0.882	0.943	0.895	0.849	0.892
Test	Precision	0.830	0.828	0.823	0.818	0.813	0.815
Train	Recall	0.883	0.881	0.965	0.85	0.822	0.894
Test	Recall	0.825	0.822	0.838	0.779	0.778	0.817
Train	Accuracy	0.882	0.88	0.953	0.863	0.832	0.892
Test	Accuracy	0.821	0.818	0.825	0.779	0.786	0.808
Train	F-score	0.941	0.943	0.977	0.939	0.926	0.948
Test	F-score	0.904	0.905	0.906	0.889	0.886	0.892

Table 3: Results obtained by training the full network

D Code

Agglomerative clustering Python Code

```python
# Import necessary libraries and packages
import numpy as np
import cv2
import scipy as sp
from distutils.version import LooseVersion
import matplotlib.pyplot as plt
import skimage
from skimage.data import coins
from sklearn.feature_extraction.image import grid_to_graph
from sklearn.cluster import AgglomerativeClustering

# Function definitions

def segment_image(img, n_clusters=10):
  '''
  Image segmentation using Ward hierarchical clustering.
  Default number of clusters = 10

  Inputs: img: grayscale image
          n_clusters: number of clusters to use for segmentation
  Return: labels: array containing cluster label for each pixel in img
  '''

  # Flatten the coins image array into a 1D array
  X = np.reshape(img, (-1, 1))

  # Define the structure A of the data. Pixels connected to their neighbors.
  connectivity = grid_to_graph(*img.shape)

  # Create an AgglomerativeClustering instance
  ward = AgglomerativeClustering(n_clusters=n_clusters, linkage='ward',
                                 connectivity=connectivity)
  # Fit hierarchical clustering
  ward.fit(X)
  # Create a 2D array of labels with the same dimension as coins
  label = np.reshape(ward.labels_, img.shape)

  return label


def save_segmentation_result(img, label, imtype='original', plot_type='contour', save_path=
                                                        None):
  '''
  Save a plot showing the result of the segmentation.
  Inputs: img: grayscale image
          label: 2D array of values same size as img - pixel cluster labels
          imtype: 'original' or 'binary' - dynamically populate plot title
          plot_type: 'contour' or 'mask' - dynamically populate file name
  '''

  # Get the number of clusters from the input img to avoid global variable conflict
  n_clusters = len(np.unique(label))
  assert(n_clusters > 0)

  if plot_type.lower() == 'contour':
    # Plot the image with the segmentation contours
    plt.figure()
    plt.imshow(img, cmap=plt.cm.gray)
    for l in range(n_clusters):
        plt.contour(label == l,
                    colors=[plt.cm.nipy_spectral(l / float(n_clusters)), ])
    plt.axis('off')
    plt.title('{} image with\n{} clusters'.
              format(imtype.capitalize(), n_clusters))

  elif plot_type.lower() == 'mask':
    # Plot the image with the segmentation mask
    plt.figure()
    plt.imshow(img, cmap=plt.cm.gray)
    # Add the contours as well. It looks better
    for l in range(n_clusters):
        plt.contour(label == l,
                    colors=[plt.cm.nipy_spectral(l / float(n_clusters)), ])
    plt.imshow(label, cmap='jet', alpha=0.5)
    plt.axis('off')
    plt.title('{} image with\n{} overlaid clusters'.
              format(imtype.capitalize(), n_clusters))

  # Define the file name using imtype argument
  plt.savefig(save_path + '{}_image_{}_{}_clusters.png'.
              format(imtype.capitalize(), plot_type, n_clusters), bbox_inches='tight')

  # Show the plot for debugging purposes
  plt.show()


# Define a path for saving files
save_path = "/content/drive/My Drive/EE981 Project/Clustering results/"

# Generate coins data
coins_data = coins()

# Apply binary threshold to give binary image
thresh = 127
val, coins_binary = cv2.threshold(coins_data, thresh, 255, cv2.THRESH_BINARY)

# Save the original and binary images to file
plt.imshow(coins_data, cmap=plt.cm.gray)
plt.axis('off')
plt.savefig(save_path + 'Original_coins_image.png', bbox_inches='tight')
plt.show()

plt.imshow(coins_binary, cmap=plt.cm.gray)
plt.axis('off')
plt.savefig(save_path + 'Binary_coins_image.png', bbox_inches='tight')
plt.show()

# Define the number of clusters to use
n_clusters = 27

# Segment the original coins image
label_original = segment_image(coins_data, n_clusters)

# Segment the binary coins image
label_binary = segment_image(coins_binary, n_clusters)

# Save 'original' image plots to file
save_segmentation_result(coins_data, label_original, imtype='original', plot_type='mask',
                                                save_path=save_path)
save_segmentation_result(coins_data, label_original, imtype='original', plot_type='contour',
                                                save_path=save_path)
```

```

# Save 'binary' image plots to file
save_segmentation_result(coins_binary, label_binary, imtype='binary', plot_type='mask',
                                                 save_path=save_path)
save_segmentation_result(coins_binary, label_binary, imtype='binary', plot_type='contour',
                                                 save_path=save_path)
```

Hierarchical Segmentation Python Code

```
# -*- coding: utf-8 -*-
"""Hierarchical Segmentation.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/11EuvzITOP4oIvTBhVaKghCwADVo8r79v
"""

# Author : Vincent Michel, 2010
#          Alexandre Gramfort, 2011
# License: BSD 3 clause
# Modified by: Group Project 10 Deep Image Segmentation, Image abd Video Processing Module.
                                                University of Strathclyde, 2020

# Commented out IPython magic to ensure Python compatibility.
# import libraries that we have to use in this project

print(__doc__)

import time as time
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
from sklearn.feature_extraction.image import grid_to_graph
from sklearn.cluster import AgglomerativeClustering
from skimage.color import rgb2gray
# %matplotlib inline
from scipy import ndimage

# upload and insert the original image file

image = plt.imread('3063.jpg') # choose your file name of original image here
image.shape
plt.title('Original image')
plt.imshow(image)

image.shape

gray = rgb2gray(image)
plt.title('Gray scale image')
plt.imshow(gray, cmap='gray')

gray.shape

# upload and insert the ground truth image file

image_ground_truth = plt.imread('3063.png') # choose your file name of ground truth image here
plt.title('ground truth image')
plt.imshow(image_ground_truth, cmap='gray')

# Function definitions
from sklearn import metrics
def segment_image(img, n_clusters=10):
  '''
  Image segmentation using Ward hierarchical clustering.
  Default number of clusters = 10

  Inputs: img: grayscale image
          n_clusters: number of clusters to use for segmentation
  Return: labels: array containing cluster label for each pixel in img
  '''

  # Flatten the coins image array into a 1D array
  X = np.reshape(img, (-1, 1))

  # Define the structure A of the data. Pixels connected to their neighbors.
  connectivity = grid_to_graph(*img.shape)

  # Create an AgglomerativeClustering instance
  ward = AgglomerativeClustering(n_clusters=n_clusters, linkage='ward',distance_threshold =
                                                     None,
                               connectivity=connectivity)
  # Fit hierarchical clustering
  ward.fit(X)
  # Create a 2D array of labels with the same dimension as coins
  label = np.reshape(ward.labels_, img.shape)

  #score = metrics.silhouette_score(X, ward.labels_)
  return label

def save_segmentation_result(img, label, imtype='original', plot_type='contour', save_path=
                                                   None):
  '''
  Save a plot showing the result of the segmentation.
  Inputs: img: grayscale image
          label: 2D array of values same size as img - pixel cluster labels
          imtype: 'original' or 'binary' - dynamically populate plot title
          plot_type: 'contour' or 'mask' - dynamically populate file name
  '''

  # Get the number of clusters from the input img to avoid global variable conflict
  n_clusters = len(np.unique(label))
  assert(n_clusters > 0)

  if plot_type.lower() == 'contour':
    # Plot the image with the segmentation contours
    plt.figure()
    plt.imshow(img, cmap=plt.cm.gray)
    for l in range(n_clusters):
        plt.contour(label == l,
                    colors=[plt.cm.nipy_spectral(l / float(n_clusters)), ])
    plt.axis('off')
    plt.title('{} image with\n{} clusters'.
              format(imtype.capitalize(), n_clusters))

  elif plot_type.lower() == 'mask':
    # Plot the image with the segmentation mask
    plt.figure()
    plt.imshow(img, cmap=plt.cm.gray)
    # Add the contours as well. It looks better
    for l in range(n_clusters):
        plt.contour(label == l,
                    colors=[plt.cm.nipy_spectral(l / float(n_clusters)), ])
    plt.imshow(label, cmap='jet', alpha=0.5)
    plt.axis('off')

  # Define the file name using imtype argument
  plt.savefig('{}_image_{}_{}_clusters.png'.format(imtype.capitalize(), plot_type, n_clusters)
                                                     ,bbox_inches='tight')

  # Show the plot for debugging purposes
  plt.show()
```

```python

# Generate data
image_gray = gray

# Define the number of clusters to use
n_clusters_10 = 10
n_clusters_20 = 20
n_clusters_30 = 30

# Segment the original coins image
label_10 = segment_image(image_gray, n_clusters_10)
label_20 = segment_image(image_gray, n_clusters_20)
label_30 = segment_image(image_gray, n_clusters_30)

# Save image plots to file
save_path = "/content/"
save_segmentation_result(image_gray, label_10, imtype='original', plot_type='mask', save_path=
                                                save_path)
save_segmentation_result(image_gray, label_20, imtype='original', plot_type='mask', save_path=
                                                save_path)
save_segmentation_result(image_gray, label_30, imtype='original', plot_type='mask', save_path=
                                                save_path)

def display(display_list):
  plt.figure(figsize=(15, 15))

  title = ['Input Image', 'contour', 'cluster = 10', 'cluster = 20', 'cluster = 30']

  for i in range(len(display_list)):
    plt.subplot(1, len(display_list), i+1)
    plt.title(title[i])
    plt.imshow(display_list[i],cmap=plt.cm.gray)
    plt.axis('off')
  plt.show()

cluster_10 = plt.imread('Original_image_mask_10_clusters.png')
cluster_20 = plt.imread('Original_image_mask_20_clusters.png')
cluster_30 = plt.imread('Original_image_mask_30_clusters.png')
contour = image_ground_truth

display([image, contour, cluster_10, cluster_20, cluster_30])
```

U-Net TensorFlow Segmentation Code [18]

```
# -*- coding: utf-8 -*-
"""EE981 Segmentation.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1Eqq-9jV4h53GV5uyYXoNYxnlyo_DBm_2

##### Copyright 2019 The TensorFlow Authors.
# Code modified by Group 10 - EE981 Image and Video Processing - University of Strathclyde,
                                              2020.

Licensed under the Apache License, Version 2.0 (the "License");
"""

from google.colab import drive
drive.mount('/content/drive')


!pip install git+https://github.com/tensorflow/examples.git

# Commented out IPython magic to ensure Python compatibility.
try:
  # %tensorflow_version only exists in Colab.
#   %tensorflow_version 2.x
except Exception:
  pass
import tensorflow as tf
print(tf.__version__)

# import necessary libraries and packages
from __future__ import absolute_import, division, print_function, unicode_literals
from tensorflow_examples.models.pix2pix import pix2pix
import tensorflow_datasets as tfds
tfds.disable_progress_bar()
from IPython.display import clear_output
import matplotlib.pyplot as plt
import numpy as np
from google.colab import files
from sklearn.metrics import precision_recall_fscore_support
import time
from sklearn.metrics import accuracy_score

# Define the path used for saving figures etc
save_path = "/content/drive/My Drive/EE981 Project/Segmentation_template_code/"

# Download the Oxford-IIIT Pets dataset
dataset, info = tfds.load('oxford_iiit_pet:3.*.*', with_info=True)

# The following code performs a simple augmentation of flipping an image. In addition,  image
                                              is normalized to [0,1].

def normalize(input_image, input_mask):
  input_image = tf.cast(input_image, tf.float32) / 255.0
  input_mask -= 1
  return input_image, input_mask

@tf.function
def load_image_train(datapoint):
  input_image = tf.image.resize(datapoint['image'], (128, 128))
  input_mask = tf.image.resize(datapoint['segmentation_mask'], (128, 128))

  if tf.random.uniform(()) > 0.5:
    input_image = tf.image.flip_left_right(input_image)
    input_mask = tf.image.flip_left_right(input_mask)

  input_image, input_mask = normalize(input_image, input_mask)

  return input_image, input_mask

def load_image_test(datapoint):
  input_image = tf.image.resize(datapoint['image'], (128, 128))
  input_mask = tf.image.resize(datapoint['segmentation_mask'], (128, 128))

  input_image, input_mask = normalize(input_image, input_mask)

  return input_image, input_mask

# The dataset already contains the required splits of test and train. Continue to use the same
                                                  split.
# Show the number of training and testing examples
print('There are {} training examples'.format(info.splits['train'].num_examples))
print('There are {} testing examples'.format(info.splits['test'].num_examples))

# The "info" object was loaded at the same time as the dataset.
# It contains information about the dataset
TRAIN_LENGTH = info.splits['train'].num_examples

# These are parameters we can change to see the effect
BATCH_SIZE = 32
BUFFER_SIZE = 1000
STEPS_PER_EPOCH = TRAIN_LENGTH // BATCH_SIZE

print("Number of training examples: {}".format(TRAIN_LENGTH))
print("Steps per epoch: {}".format(STEPS_PER_EPOCH))

# Looks like this is calling the functions 'load_image_train' and 'load_image_test'
# with the data from dataset['train'] and dataset['test'] as function arguments.
# The num_parallel_calls thing is some optimisation for parallel processing
# Within load_image_train, there is some data augmentation occurring with 50% probability.

# The train and test variables being generated here will each contain
# a collection of images and ground-truth masks. Shapes (128, 128, 3) and
# (128, 128, 1), respectively. The labels or classes are {0,1,2} for the mask.

train = dataset['train'].map(load_image_train, num_parallel_calls=tf.data.experimental.
                                                 AUTOTUNE)
test = dataset['test'].map(load_image_test)

# Generate train and test datasets to be used later
train_dataset = train.cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE).repeat()
train_dataset = train_dataset.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)
test_dataset = test.batch(BATCH_SIZE)

# Let's take a look at an image example and its correponding mask from the dataset.

def display(display_list):
  plt.figure(figsize=(15, 15))

  title = ['Input Image', 'True Mask', 'Predicted Mask']

  for i in range(len(display_list)):
    plt.subplot(1, len(display_list), i+1)
    plt.title(title[i])
    plt.imshow(tf.keras.preprocessing.image.array_to_img(display_list[i]))
    plt.axis('off')
  plt.show()

# Display a sample image and the associated ground truth mask
for image, mask in train.take(1):
  sample_image, sample_mask = image, mask
display([sample_image, sample_mask])

# Define the model
OUTPUT_CHANNELS = 3

base_model = tf.keras.applications.MobileNetV2(input_shape=[128, 128, 3], include_top=False)

# Use the activations of these layers
layer_names = [
    'block_1_expand_relu',   # 64x64
    'block_3_expand_relu',   # 32x32
    'block_6_expand_relu',   # 16x16
    'block_13_expand_relu',  # 8x8
    'block_16_project',      # 4x4
]
layers = [base_model.get_layer(name).output for name in layer_names]

# Create the feature extraction model
down_stack = tf.keras.Model(inputs=base_model.input, outputs=layers)

# Set to True if you want to train the whole model and False if you want to
# use the pre-trained encoder weights
down_stack.trainable = True

# The decoder/upsampler is simply a series of upsample blocks implemented in TensorFlow
                                                examples.

up_stack = [
    pix2pix.upsample(512, 3),  # 4x4 -> 8x8
    pix2pix.upsample(256, 3),  # 8x8 -> 16x16
    pix2pix.upsample(128, 3),  # 16x16 -> 32x32
    pix2pix.upsample(64, 3),   # 32x32 -> 64x64
]

def unet_model(output_channels):
  inputs = tf.keras.layers.Input(shape=[128, 128, 3])
  x = inputs

  # Downsampling through the model
  skips = down_stack(x)
  x = skips[-1]
  skips = reversed(skips[:-1])

  # Upsampling and establishing the skip connections
  for up, skip in zip(up_stack, skips):
    x = up(x)
    concat = tf.keras.layers.Concatenate()
    x = concat([x, skip])

  # This is the last layer of the model
  last = tf.keras.layers.Conv2DTranspose(
      output_channels, 3, strides=2,
      padding='same')  #64x64 -> 128x128

  x = last(x)

  return tf.keras.Model(inputs=inputs, outputs=x)

# Train the model

# Change learning rate and optimizer here
l_rate = 1e-5
optimizer = tf.keras.optimizers.Adam(learning_rate=l_rate)

model = unet_model(OUTPUT_CHANNELS)
model.compile(optimizer=optimizer,
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

# Have a quick look at the resulting model architecture
# Commented out to supress output
# tf.keras.utils.plot_model(model, show_shapes=True)


def create_mask(pred_mask):
  pred_mask = tf.argmax(pred_mask, axis=-1)
  pred_mask = pred_mask[..., tf.newaxis]
  return pred_mask[0]

# Get the prediction before any traning
for image, mask in test_dataset.take(1):
  predicted_mask = model.predict(image)
  predicted_mask = create_mask(predicted_mask)

# Convert the sample mask to a numpy array, squeeze and cast to integer.
# Result is a 2D array of integer values.
predicted_mask = predicted_mask.numpy().squeeze().astype(int)

# Save the result to file
plt.imshow(predicted_mask)
plt.axis('off')
plt.savefig(save_path + "untrained_prediction.png", bbox_inches='tight')
plt.show()

def show_predictions(dataset=None, num=1):
  if dataset:
    i = 0
    for image, mask in dataset.take(num):
      pred_mask = model.predict(image)
      print("Index: {}".format(i))
      i += 1
      display([image[0], mask[0], create_mask(pred_mask)])
  else:
    # These sample_x variables are global variables
    display([sample_image, sample_mask,
             create_mask(model.predict(sample_image[tf.newaxis, ...]))])

# Observe how the model improves while it's training. A callback function is defined.
class DisplayCallback(tf.keras.callbacks.Callback):
  def on_epoch_end(self, epoch, logs=None):
    clear_output(wait=True)
    show_predictions()
    print ('\nSample Prediction after epoch {}\n'.format(epoch+1))

# Additional callbacks
callback_checkpoint = tf.keras.callbacks.ModelCheckpoint(filepath='/content/drive/My Drive/
                                                EE981 Project/best_model',
                                      monitor='val_accuracy', verbose=0, save_best_only=True,
                                      save_weights_only=False, mode='max')

callback_early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', patience=6)

# Train the model
EPOCHS = 50
VAL_SUBSPLITS = 10
VALIDATION_STEPS = info.splits['test'].num_examples//BATCH_SIZE//VAL_SUBSPLITS

# Comment or un-comment callbacks line as required
model_history = model.fit(train_dataset, epochs=EPOCHS,
                          steps_per_epoch=STEPS_PER_EPOCH,
                          validation_steps=VALIDATION_STEPS,
                          validation_data=test_dataset,
                          callbacks=[DisplayCallback()])
                          #callbacks=[DisplayCallback(), callback_checkpoint,
                                                               callback_early_stopping
                                                               ])

# The loss and accuracy values from training can be obtained from the history
loss = model_history.history['loss']
val_loss = model_history.history['val_loss']

train_accuracy = model_history.history['accuracy']
val_accuracy = model_history.history['val_accuracy']

steps = range(STEPS_PER_EPOCH)
epochs = range(EPOCHS)

# Here's some more flexible code for creating and editing the plot appearance
# Notice it has a problem if you use early stopping

def plot_performance(opt, lr=l_rate, metric_used='loss', save_to_file=False, save_path=
                                                save_path):
  '''
  Function for plotting the training and validation performance with each epoch.

  The following variables are assumed to be global: epochs, loss, val_loss,
  train_accuracy, val_accuracy.
  '''

  fig = plt.figure()
  ax = fig.add_subplot(1, 1, 1)

  if metric_used == 'loss':
    plt.plot(epochs, loss, 'r', label='Training loss')
    plt.plot(epochs, val_loss, 'b', label='Validation loss')

    ax.set_title('Optimiser: {}, learning rate = {},\nBatch size = {}, steps per epoch = {}'.
                                                    format(opt, lr, BATCH_SIZE, STEPS_PER_EPOCH
                                                    ))
    ax.set_ylabel('Loss Value')
    #y_major_ticks = np.arange(0, 1.1, 0.1)
    #ax.set_yticks(y_major_ticks)

  elif metric_used == 'accuracy':
    plt.plot(epochs, train_accuracy, 'r', label='Training accuracy')
    plt.plot(epochs, val_accuracy, 'b', label='Validation accuracy')
    ax.set_title('Optimiser: {}, learning rate = {},\nBatch size = {}, steps per epoch = {}'.
                                                    format(opt, lr, BATCH_SIZE, STEPS_PER_EPOCH
                                                    ))
    ax.set_ylabel('Accuracy Value')

  # Generic commands irrespective of metric
  ax.set_xlabel('Epoch')
  ax.set_xbound(-1,EPOCHS)
  x_major_ticks = np.arange(0, EPOCHS+1, 1)
  #ax.set_xticks(x_major_ticks)
  plt.locator_params(axis='x', nbins=20)
  ax.legend()

  # Add a grid. Use alpha parameter to control gridline opacity
  ax.grid(which='major', alpha=0.5)

  # Save the figure to file, with filename determined by metric_used value
  if save_to_file:
    #fig.savefig(save_path + "train_val_" + metric_used + "_plot_" + opt + "_{}.png".format(lr), bbox='tight')
    fig.savefig(save_path + 'full_train_L_{}_BS_{}_E_{}_opt_{}.png'.format(l_rate, BATCH_SIZE, EPOCHS, opt))

  # Show the plot in the notebook
  plt.show()

# Call the function to generate and optionally save the plot to file
plot_performance(opt="Adam", lr=l_rate, metric_used='loss', save_to_file=True)

# Make predictions

# The second argument is the number of predictions you want to display
# For each row of images, the index will be printed above. This is to
# make it easier to select individual results to save using show_prediction_metrics()
show_predictions(test_dataset, 100)

# Evaluate model performance

# Given a model and a dataset, compute the mean precision,
# recall, f-score and accuracy

def get_performance_metrics(model_in, in_dataset, print_progress=True):

  # Initialise an empty matrix with the correct dimensions.
  # Inefficient approach, but it works
  pm = [[None for i in range(STEPS_PER_EPOCH)] for j in range(4)]

  # Initialise a list to store execution times
  times = [None for i in range(STEPS_PER_EPOCH)]

  # Take some samples from the dataset passed to the function
  example = in_dataset.take(STEPS_PER_EPOCH)

  # Initialise a counter variable for populating the metrics matrix
  i = 0
  for sample in example:
    tic = time.clock()
    image, mask = sample[0], sample[1]
    predicted_mask = model_in.predict(image)
    predicted_mask = create_mask(predicted_mask)

    # You can do all the operations at once (numpy, squeeze, to int, flatten)
    predicted_mask = predicted_mask.numpy().squeeze().astype(int).flatten()

    # Similarly here, we can get the GT mask (128, 128) image in one line
    gt_mask = mask[0].numpy().squeeze().astype(int).flatten()

    # Populate the 'performance metrics' (pm) matrix with the values each iteration
    pm[0][i], pm[1][i], pm[2][i], support = precision_recall_fscore_support(gt_mask,
                                                          predicted_mask, average='macro')
    pm[3][i] = accuracy_score(gt_mask, predicted_mask)

    toc = time.clock()
    times[i] = (toc - tic)

    # Print progress and iteration duration for debugging purposes
    if print_progress:
      if i%5 == 0:
        print("Currently executing iteration {}. Last iteration took {:.2f} seconds".format(i,
                                                              times[i]))
    i += 1

  # Compute the mean value for each metric
  pm_mean = np.mean(pm, axis=1)

  # Print the results in a nice format
  print("MODEL PARAMETERS:")
  print("Learning rate: {}".format(l_rate))
  print("Batch size: {}".format(BATCH_SIZE))
  print("Steps per epoch: {}".format(STEPS_PER_EPOCH))
  print("----------------------------------------")
  print("RESULTS:")
  print("Mean precision: {:.3f}".format(pm_mean[0]))
  print("Mean recall: \t{:.3f}".format(pm_mean[1]))
  print("Mean F-score: \t{:.3f}".format(pm_mean[2]))
  print("Mean accuracy: \t{:.3f}\n\n".format(pm_mean[3]))

  return pm_mean


def generate_results_file(train_metrics, test_metrics, opt_used):
  # Summarise the model parameters and results in a text file
  # Uses global variable save_path and global variables that store model parameters
  # Takes pm array containing the performance metric values
  # Takes opt_used (optimiser used) as a string

  with open(save_path + 'full_train' + 'L_{}_BS_{}_E_{}_opt_{}.txt'.format(l_rate, BATCH_SIZE,
                                                         EPOCHS, opt_used), 'w') as f:
    f.write("SEGMENTATION MODEL RESULTS\n" +
            "-------------------\n" +
            "Parameters:\n" +
            "\tLearning rate: {}\n".format(l_rate) +
            "\tBatch size: {}\n".format(BATCH_SIZE) +
            "\tEpochs: {}\n".format(EPOCHS) +
            "\tOptimizer: {}\n".format(opt_used) +
            "-------------------\n" +
            "Train metrics\n" +
            "\tMean precision: {:.3f}\n".format(train_metrics[0]) +
            "\tMean recall: {:.3f}\n".format(train_metrics[1]) +
            "\tMean accuracy: {:.3f}\n".format(train_metrics[2]) +
            "\tMean F-score: {:.3f}\n".format(train_metrics[3]) +
            "-------------------\n" +
            "Test metrics\n" +
            "\tMean precision: {:.3f}\n".format(test_metrics[0]) +
            "\tMean recall: {:.3f}\n".format(test_metrics[1]) +
            "\tMean accuracy: {:.3f}\n".format(test_metrics[2]) +
```

```python
            "\tMean F-score: {:.3f}".format(test_metrics[3]))


train_metrics = get_performance_metrics(model, train_dataset, print_progress=False)
test_metrics = get_performance_metrics(model, test_dataset, print_progress=False)
generate_results_file(train_metrics, test_metrics, 'Adam')

# TESTING CELL - LOAD THE BEST MODEL SAVED FROM THE CALLBACK AND USE IT FOR PREDICTIONS ON THE TEST DATA

new_model = tf.keras.models.load_model('/content/drive/My Drive/EE981 Project/best_model')
# Print a summary of the model, optionally
# new_model.summary()

best_model_test_results = get_performance_metrics(new_model, test_dataset, print_progress=True)

# Find an example of a good or bad prediction, show it and compute the scores
# Terrible solution, but I can't work out how to extract a single
# instance from the TensorFlow object otherwise

def show_prediction_metrics(dataset, num, index, save_to_file=False):
  assert(num >= index)
  example = dataset.take(num)
  i = 0
  for sample in example:
    if i != index:
      i += 1
      continue
    else:
      image, mask = sample[0], sample[1]
      predicted_mask = model.predict(image)
      predicted_mask = create_mask(predicted_mask)
      predicted_mask = predicted_mask.numpy().squeeze().astype(int)
      gt_mask = mask.numpy().squeeze().astype(int)
      break

  # Configure the plots and save them to file for inclusion in report
  fig = plt.figure()
  ax = fig.add_subplot(1, 1, 1)
  plt.imshow(image[0])
  plt.axis('off')
  if save_to_file:
    fig.savefig(save_path + "test_{}_prediction_original.png".format(index), bbox_inches='tight')

  fig = plt.figure()
  ax = fig.add_subplot(1, 1, 1)
  plt.imshow(gt_mask[0])
  plt.axis('off')
  if save_to_file:
    fig.savefig(save_path + "test_{}_prediction_ground_truth.png".format(index), bbox_inches='tight')

  fig = plt.figure()
  ax = fig.add_subplot(1, 1, 1)
  plt.imshow(predicted_mask)
  plt.axis('off')
  if save_to_file:
    fig.savefig(save_path + "test_{}_prediction_predicted_mask.png".format(index), bbox_inches='tight')

  # Get the precision, recall and F-score for this prediction
  y_true = gt_mask[0].flatten()
  y_pred = predicted_mask.flatten()

  precision, recall, f_score, support = precision_recall_fscore_support(y_true, y_pred,
                                                 average='macro')
  accuracy = accuracy_score(y_true, y_pred)

  print("Accuracy: {:.3f}".format(accuracy))
  print("Precision: {:.3f}".format(precision))
  print("Recall: {:.3f}".format(recall))
  print("F-score: {:.3f}".format(f_score))


  with open(save_path + "test_{}_prediction_metrics.txt".format(index), 'w') as f:
    f.write("Prediction metrics:\n" +
            "\tAccuracy: {:.3f}\n".format(accuracy) +
            "\tPrecision: {:.3f}\n".format(precision) +
            "\tRecall: {:.3f}\n".format(recall) +
            "\tF-score: {:.3f}\n".format(f_score))


  return accuracy, precision, recall, f_score

# Call the function to get some scores and save the plots as images
show_prediction_metrics(test_dataset, 100, 76, save_to_file=True)
```

References

[1] Pablo Arbelaez, Michael Maire, Charless Fowlkes, and Jitendra Malik. Contour detection and hierarchical image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(5):898–916, May 2011.

[2] Jifeng Dai, Kaiming He, and Jian Sun. Instance-aware semantic segmentation via multi-task network cascades. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[3] Binmei Liang and Jianzhou Zhang. Kmsgc: An unsupervised color image segmentation algorithm based on k-means clustering and graph cut, 2014.

[4] Christophe Rosenberger, Sébastien Chabrier, Hélène Laurent, and Bruno Emile. Unsupervised and supervised image segmentation evaluation. *Advances in Image and Video Segmentation, IGI Global*, pages 365–393, 01 2006.

[5] Hui Zhang, Jason E. Fritts, and Sally A. Goldman. Image segmentation evaluation: A survey of unsupervised methods. *Computer Vision and Image Understanding*, 110(2):260 – 280, 2008.

[6] Mariana Belgiu and Lucian Drăguţ. Comparing supervised and unsupervised multiresolution segmentation approaches for extracting buildings from very high resolution imagery. *ISPRS Journal of Photogrammetry and Remote Sensing*, 96:67 – 75, 2014.

[7] Huikai Wu, Junge Zhang, Kaiqi Huang, Kongming Liang, and Yizhou Yu. Fastfcn: Rethinking dilated convolution in the backbone for semantic segmentation. *arXiv preprint arXiv:1903.11816*, 2019.

[8] Simon Jégou, Michal Drozdzal, David Vázquez, Adriana Romero, and Yoshua Bengio. The one hundred layers tiramisu: Fully convolutional densenets for semantic segmentation. *CoRR*, abs/1611.09326, 2016.

[9] Towaki Takikawa, David Acuna, Varun Jampani, and Sanja Fidler. Gated-scnn: Gated shape cnns for semantic segmentation. *CoRR*, abs/1907.05740, 2019.

[10] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015.

[11] *Essentials of Business Analytics: An Introduction to the Methodology and its Applications*, volume 264 of *International Series in Operations Research & Management Science*. Springer International Publishing, Cham, 2019.

[12] M Namratha and TR Prajwala. A comprehensive overview of clustering algorithms in pattern recognition. *IOR Journal of Computer Engineering*, 4(6):23–30, 2012.

[13] Vincent Michel and Alexandre Gramfort. sklearn.cluster.agglomerativeclustering. `https://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html`. Accessed 28th March 2020.

[14] Vincent Michel and Alexandre Gramfort. A demo of structured ward hierarchical clustering on an image of coins. `https://scikit-learn.org/dev/auto_examples/cluster/plot_coin_ward_segmentation.html#sphx-glr-download-auto-examples-cluster-plot-coin-ward-segmentation-py`, 2011. Accessed 21st March 2020.

[15] A Zisserman O. M. Parkhi, A Vedaldi and C. V. Jawahar. The oxford-iiit pet dataset.

[16] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015.

[17] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks, 2018.

[18] TensorFlow Tutorial Website. Image segmentation. `https://www.tensorflow.org/tutorials/images/segmentation`, 2015. Accessed 21st March 2020.

[19] Jason Brownlee. How to configure the learning rate when training deep learning neural networks. `https://machinelearningmastery.com/learning-rate-for-deep-learning-neural-networks/`. Accessed: 12th March 2020.

[20] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[21] Matthias Feurer and Frank Hutter. *Hyperparameter Optimization*, pages 3–33. Springer International Publishing, Cham, 2019.

[22] E. Bochinski, T. Senst, and T. Sikora. Hyper-parameter optimization for convolutional neural network committees based on evolutionary algorithms. In *2017 IEEE International Conference on Image Processing (ICIP)*, pages 3924–3928, 2017.

[23] Jason Brownlee. How to calculate precision, recall, and f-measure for imbalanced classification. `https://machinelearningmastery.com/precision-recall-and-f-measure-for-imbalanced-classification/"`, February 2020. Accessed: 4th April 2020.