LONDON'S GLOBAL UNIVERSITY

Anomaly detection in parcel X-rays using Siamese and triplet neural networks

Martynas Janonis[1]

BSc Computer Science

Supervisor: Dr. Lewis Griffin

Submission date: 19th May 2020

[1]**Disclaimer:** This report is submitted as part requirement for the BSc Computer Science at UCL. It is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged

Abstract

Detecting anomalies in X-ray security imaging is quite a challenging task. Previous solutions used features extracted from a CNN trained only on photographic images. This work aims to improve the performance of previous approaches by using a Siamese/triplet CNN that is pre-trained on ImageNet and fine tuned on an X-ray image dataset. In particular, this paper evaluates whether a representation extracted from a Siamese/triplet network, trained to distinguish if two X-ray images were different views of the same parcel or not, is effective for anomaly detection. Comparison of various network architectures and training methods is provided in detail. The best results, achieved by an isolation forest trained on embeddings generated by a triplet Densenet-201 model (pre-trained on ImageNet and fine tuned on the X-ray images dataset), were: $F_1 = 0.4048$, AUROC $= 0.6922$, $\kappa = 0.0908$. The results indicate that these embeddings show minor potential for use in anomaly detection.

Contents

Chapter 1

Introduction

1.1 Problem outline

X-ray imaging is prevalent in modern society. It is used to detect various forms of cancers in humans [1] and inspect parcels, luggage and vehicles for illicit items [2].

While multiple advances in X-ray screening equipment have been made, the resulting images are still manually inspected by humans. As a result, the performance of the screening system ultimately depends on human factors and training [3]. Additionally, due to the massive volume of parcels, the repetitive nature of the job, as well as the visual challenge of discerning individual items in translucent and overlapping X-ray images, human operators are prone to making costly mistakes [4, 5, 6].

Dense and large objects, such as laptops and game consoles, are especially difficult to analyse. Their construction makes it hard for the X-rays to penetrate and image properly, thus making them an ideal place to store illegal goods. Threats placed inside or behind the laptop are especially difficult for human operators to spot [7]. Furthermore, in an airport setting, X-ray operators have about 3 seconds to make a decision on whether the bag contains illegal items [8]. This time pressure further diminishes the effectiveness of current day screening systems. An automated, ML based appearance anomaly detection solution, could potentially solve these issues by flagging abnormal looking packages, indicating that their contents might be malicious.

1.2 Aims and goals

Some previous works indicate that transfer learnt representations extracted from a CNN can be used for anomaly detection [9]. The question is whether these representations can be improved by training the CNN on another task, such as determining whether two X-ray images show the same parcel. Potentially, the representations generated after training a network on that task will have certain features which are more useful for detecting anomalies.

The aim of this project is to find out whether representations extracted from a Siamese or a triplet network, trained on discerning whether a pair of X-ray images are different views of the same parcel or not, can be used for effective appearance anomaly detection.

This project has the following goals:

- Train a Siamese and a triplet network to decide if two parcel X-rays are different views of the same parcel or not.
- Assess whether the resulting representations extracted from those networks are effective or not for anomaly detection.

1.3 Project development overview

This project was carried out in multiple stages. Firstly, a literature review was performed to find out the current approaches to appearance anomaly detection. Secondly, multiple network architectures were compared and the best performing one was picked for further training. Thirdly, a Siamese and a triplet network was trained and evaluated. Finally, the representations generated by the best performing neural network were passed through an isolation forest and a linear SVM to determine their usefulness in anomaly detection.

1.4 Chapter overview

The rest of the paper is structured as follows:

Chapter 2 gives a summary of the background research, network architectures and datasets used for training.

Chapter 3 provides a detailed explanation of the design and implementation of the anomaly detection system.

Chapter 4 evaluates the results achieved by the final models.

Chapter 5 concludes the paper by giving a summary of what was achieved and provides an evaluation of the entire project. It also briefly discuses potential future work that could be done if the project was developed for a longer period of time.

Chapter 2

Context

2.1 Background research

The background research is based on the works of Mouton *et al.* [10], Rogers *et al.* [11] and Akcay *et al.* [12].

2.1.1 Anomaly detection

Currently, there aren't many works published on the topic of unsupervised anomaly detection within X-ray imaging. Most authors instead focus on supervised learning for the purposes of object classification, detection and segmentation.

One of the earliest anomaly detection approaches [13] trains sparse feed-forward auto-encoder networks on normal data and uses the hidden layer activations of the network as representations. An SVM is then used to classify the images as normal or anomalous. Results on the MNIST dataset and a cargo container dataset showed some success as the normal class of images had a low variability, which was easily captured by the network. The work shows that representations extracted from the network are significant for detecting anomalies in images. Unfortunately, the data contained in X-ray images of parcels has a high variability, which means that the auto-encoder networks might not be suitable for the task of anomaly detection in X-ray imaging [14, 15].

The work by Andrews *et al.* [9] shows a promising representation learning framework for anomaly detection. The extracted representations from a CNN's pooling layer, that was trained to distinguish empty cargo containers from non-empty ones proved to be useful for anomaly detection. This method has the advantage of making no assumptions about the generating distributions of the normal or anomaly classes. The authors show that the transfer learnt representations, with some tweaking, show promise for use in anomaly detection.

Griffin *et al.* [14] describes another approach to anomaly detection in X-ray imaging that uses features extracted from a network similar to Inception V3, which is trained on the ImageNet dataset, to train a multivariate Gaussian model. The Gaussian model is then used to capture the distribution of the dataset. The resulting anomaly status of an image is based on the computed likelihoods relative to the Gaussian model. This method achieved an AUC (Area Under the Curve) of 92.5%. This paper also suggested multiple ways to increase the performance of the system. One method is to use a more advanced CNN architecture to generate embeddings and another is to use a CNN specifically trained on X-ray images. The authors also mention that while there is major potential to achieve better performance by using a CNN trained on X-ray images, sourcing the data might be a significant challenge.

This project seeks to implement and test both performance increasing suggestions of Griffin *et al.* [14].

2.1.2 Self-supervised representation learning

Self-supervised learning enables the ability to train a neural network using the labels that can be inferred from the data. Labelled datasets are hard and expensive to produce, but, in contrast, unlabelled data is abundant. One way to make use of that unlabelled data is to set the training task so that the supervision can be achieved from the data itself [16].

The network is trained to perform a self-supervised task (a pretext task). The task can be quite arbitrary (such as predicting image rotations [17]), but the idea is that by learning to do that task well, the network learns certain representations of the input that can be later used to complete a more complex task. In the context of this project, the pretext task is to classify whether two X-ray images show the same parcel from different views or not, and the more complex, downstream task, is to use those generated embeddings to classify images as anomalous or benign.

The work by Doersch *et al.* [18] explores a similar concept. The authors extract random pairs of patches from images of a large, unlabelled dataset, and train a CNN to predict the position of the second patch relative to the first. This, in theory, makes the network learn to recognize shapes, objects and spatial context of the images. This work shows that the resulting CNN trained on the aforementioned task outperformed a randomly initialized CNN, resulting in a state-of-the-art performance on the Pascal VOC 2011 dataset. This provides the motivation, that by training a CNN to determine if two image patches belong to the same parcel, it could be used to generate useful intermediate representations of X-ray images.

2.2 Siamese networks

A Siamese neural network consists of two identical networks, that share the same weights, joined at their output [19]. This architecture allows a model to tell how similar its inputs are, instead of making it classify one input. Siamese networks are commonly used in one-shot learning [20] (where one has very few training examples of each class) and facial recognition [21].

Siamese networks are used with a distance-based loss, in this case — contrastive loss. Each sister network takes one input each. The outputs of the last layers are used to compute the distance-based loss, which shows how similar the inputs are (how close are their embeddings in n-dimensional space). Training the network makes it generate embeddings which have low distances between positive pairs and distances greater than some margin between negative pairs.

Positive pairs consist of an anchor sample and a positive sample, which are similar in certain metrics. In contrast, negative pairs consist of an anchor sample and a negative sample, which are dissimilar in those metrics. In this project, a positive pair contains two images of the same parcel, but from different perspectives, and a negative pair contains two images of unrelated parcels.

In the context of this project, let Img_1 and Img_2 be a pair of images. Let Y be a binary label, $Y = 0$ if the images show different parcels and $Y = 1$ if the images show different views of the same parcel. Let $\mathbf{W}$ be the shared weight vector that is learned. Then $\text{CNN}_{\mathbf{W}}(\text{Img})$ is an embedding of the image Img that is extracted from the network's last pooling layer after running the image through it.

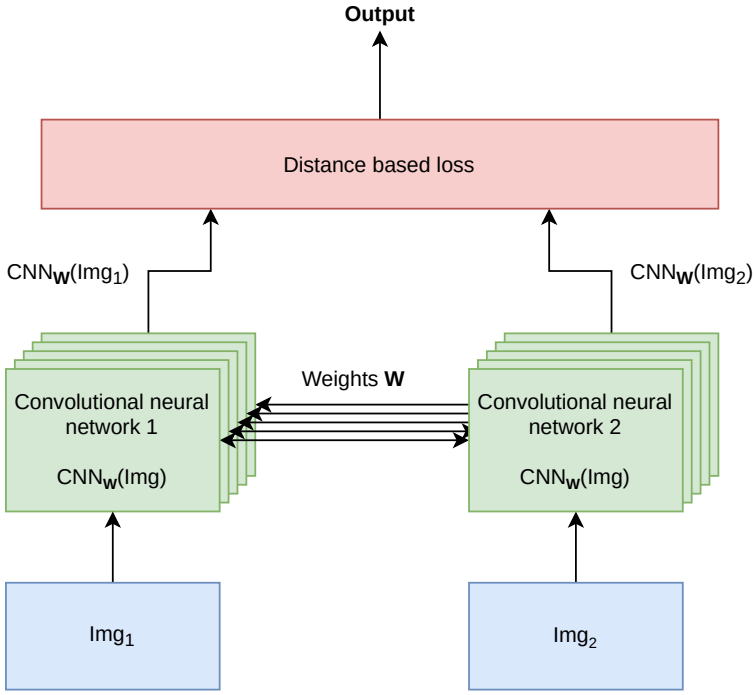


Figure 2.1: Siamese network architecture diagram [22]

Then the contrastive loss is defined as [23]:

$$\mathcal{L}_{\text{contrastive}}(\mathbf{W}, Y, \text{Img}_1, \text{Img}_2) = (1-Y)\frac{1}{2}(D_{\mathbf{W}})^2 + (Y)\frac{1}{2}(\text{ReLU}(m - D_{\mathbf{W}}))^2$$

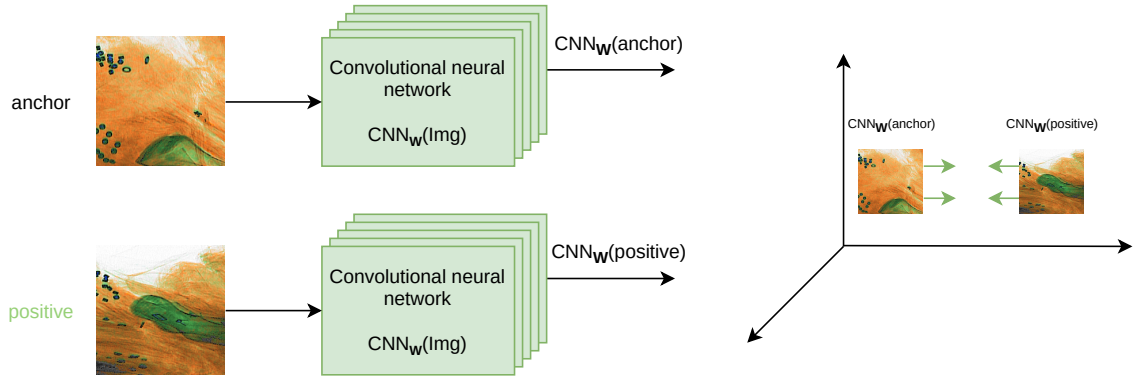
where $m > 0$ is a margin that defines a radius around $\text{CNN}_{\mathbf{W}}(\text{Img})$ and $D_{\mathbf{W}}$ is the euclidean distance between the generated embeddings [23]:

$$D_{\mathbf{W}} = \|\text{CNN}_{\mathbf{W}}(\text{Img}_1) - \text{CNN}_{\mathbf{W}}(\text{Img}_2)\|_2$$

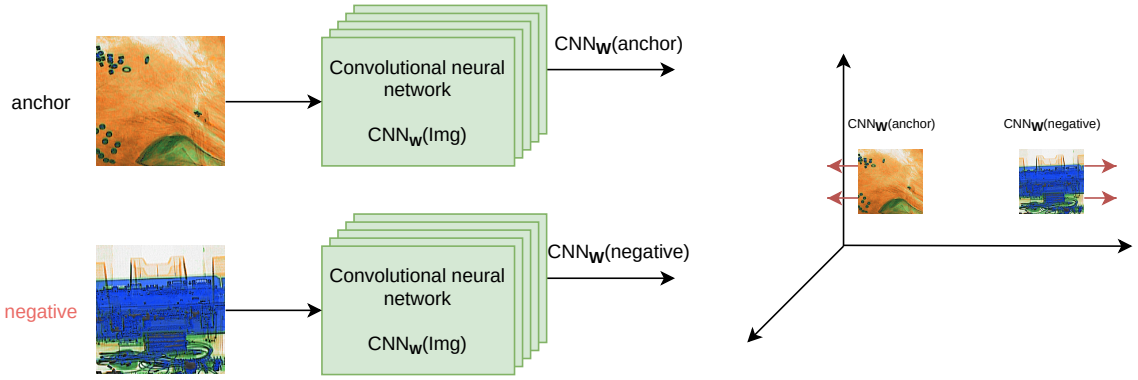
Positive pairs will have $\mathcal{L}_{\text{contrastive}} = 0$ only if the distance between the two generated representations is 0. The bigger the distance between the positive pair, the bigger the resulting loss.

Negative pairs will have $\mathcal{L}_{\text{contrastive}} = 0$ only if the distance between the two representations is greater than margin m. Otherwise, the smaller the distance, the bigger the loss (up to m).

The margin is used to avoid wasting efforts on maximizing the distance between already distant negative pairs. This allows the network to focus on more difficult training examples [24].



(a) Training example with a positive pair



(b) Training example with a negative pair

Figure 2.2: Training setup using a distance-based loss[1]

Figure 2.2 shows an example of two training steps, one containing a positive pair and another containing a negative pair. In the positive case, the network "pulls" the representations together while in the negative case, the network "pushes" them apart.

The architecture shown in figure 2.1 takes quite a lot of memory in the GPU, as two big convolutional neural networks have to be loaded at the same time. To reduce the memory footprint, instead of having two identical networks loaded at the same time and passing Img_1 to one sister network and Img_2 to another, only one CNN is loaded and the images are passed to it sequentially. As the weights of the two CNNs are the same, the resulting network will perform the same.

2.3 Triplet networks

A triplet neural network consists of three identical networks that share the same weights. When the network gets fed three inputs, it outputs 2 values — the L_2 distance between the anchor and the positive sample as well as the L_2 distance between the anchor and the negative sample [25]. Those values are then fed into the triplet loss function.

[1] Adapted from [24]

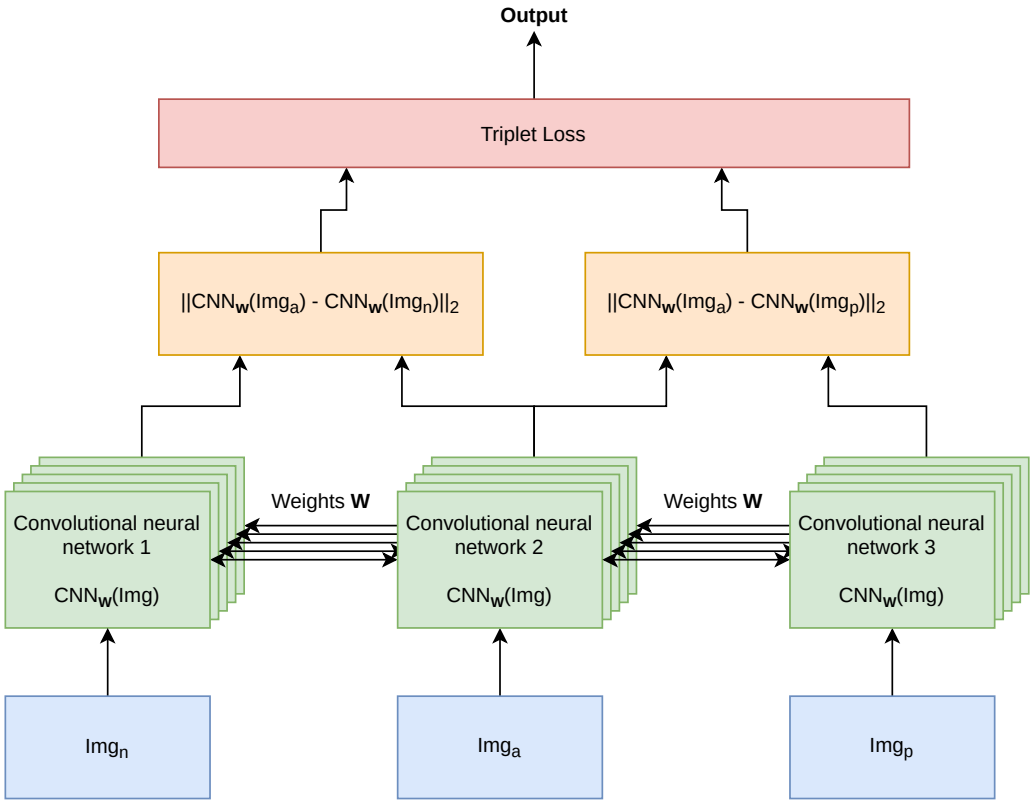


Figure 2.3: Triplet network architecture diagram

Let Img_a, Img_p and Img_n be a triplet of images. Let $\mathbf{W}$ be the shared weight vector that is learned. Let $d(x, y)$ be the distance between embeddings x and y in some embedding space. Then $\text{CNN}_\mathbf{W}(\text{Img})$ is an embedding of the image Img that is extracted from the network's last pooling layer after running the image through it. In this case, the anchor and the positive samples are X-ray images of the same parcel, but taken from different views while the negative sample is an X-ray image of a completely different parcel.

Then the triplet loss is defined as:

$$\mathcal{L}_{\text{triplet}}\left(\mathbf{W}, \text{Img}_a, \text{Img}_p, \text{Img}_n\right) = ReLU(m + \|\text{CNN}_\mathbf{W}(\text{Img}_a) - \text{CNN}_\mathbf{W}(\text{Img}_p)\|_2^2 - \|\text{CNN}_\mathbf{W}(\text{Img}_a) - \text{CNN}_\mathbf{W}(\text{Img}_n)\|_2^2)$$

There are three categories of triplets [26]:

- Easy triplets: $d(a, p) + m < d(a, n)$ have a loss of 0 as the negative sample is further away from the anchor than the positive one in the embedding space.
- Hard triplets: $d(a, n) < d(a, p)$ have a loss greater than m as the negative sample is closer to the anchor than the positive one.
- Semi-hard triplets: $d(a, p) < d(a, n) < d(a, p) + m$ have a loss $0 < \mathcal{L} < m$ as the negative sample is further away from the anchor than the positive, but the negative sample is still within the margin m.

The learning objective of a triplet network is to generate embeddings such that the anchor and the positive sample are closer together than the anchor and the negative sample in n-dimensional embedding space by some margin $m > 0$.

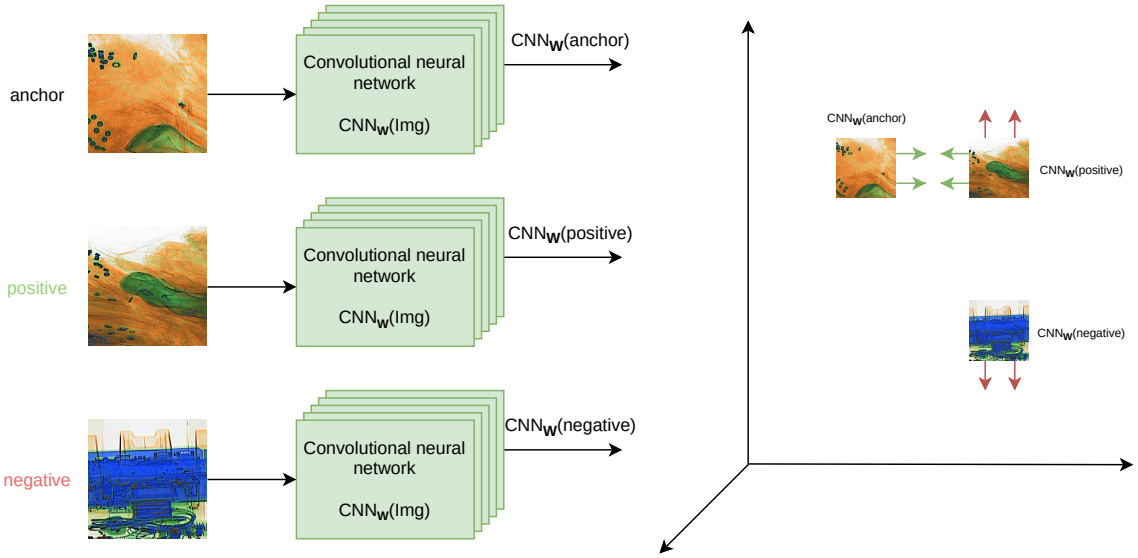


Figure 2.4: Training step of a triplet network. The network "pulls" the anchor and the positive representations together while, at the same time, "pushes" the anchor and the negative representations away from each other.[2]

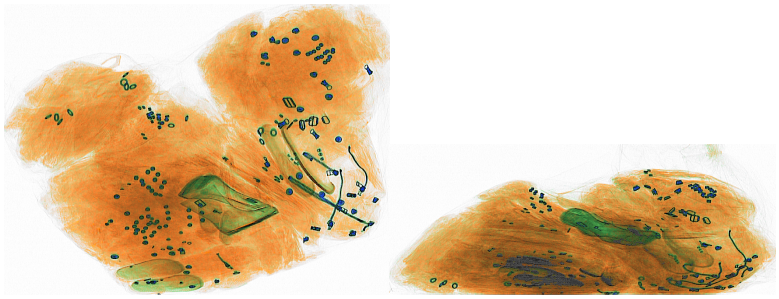
2.4 CAST dataset

This project uses the OSCT Borders X-ray Image dataset, made by the UK Home Office Centre for Applied Science and Technology (CAST). The dataset contains X-ray images of parcels, divided into two sets.

The first, stream-of-commerce set, contains X-ray images of 5000 parcels. The images were taken in a UK postal distribution centre. Most parcel contents are unrecognizable, but various clothing, machine and electronic items can be seen in some.

The second, staged threat set, contains images of 160 parcels. These images show firearms or their parts hidden inside parcels containing normal items.

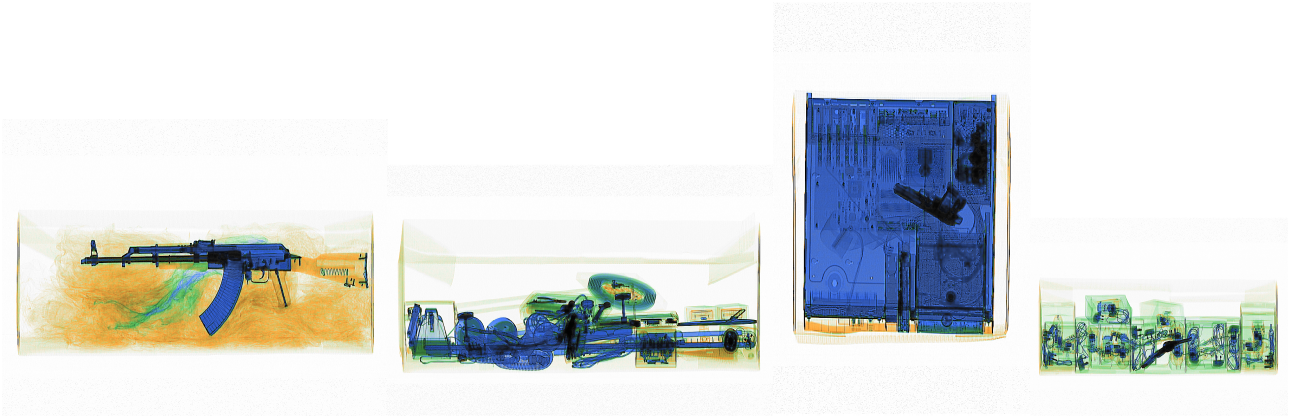
All parcel images are dual-view, meaning they have been taken from two different perspectives (i.e. one shows a top-down view and another — side-view). They are also pseudo-coloured using information from dual-energy imaging. An example is shown in figure 2.5



(a) Top-down view (b) Side view

Figure 2.5: An example of a dual-view parcel image from the dataset, containing clothing and footwear

[2]Adapted from [24]



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

Figure 2.6: Examples from the staged threat set

2.5 Software & other tools used for the project

This project makes heavy use of the PyTorch deep-learning library [27]. All training scripts, network architecture and loss implementations, etc. were written using PyTorch. The TensorFlow library was a viable alternative. Both libraries are more or less functionally equivalent, so the choice ultimately boils down to personal preference.

Scikit-learn [28] was used to train SVMs and isolation forests for anomaly detection. It was chosen due to previous experience with it, its popularity in the field and great documentation.

OpenCV was used in various stages of the data pre-processing pipeline. It was used to trim, resize and pad the images. It was selected as it is easy to use, is well documented and performs well.

The starting codebase was taken from Adam Bielski's `siamese-triplet` repository on GitHub [29], which provides sample implementations of Siamese and triplet networks using PyTorch. It is licensed under the BSD 3-Clause license. The code was modified and extended during the development of this project. All modifications are licensed under the EUPL v1.2-or-later license.

Chapter 3

Design and Implementation

3.1 High-level design overview

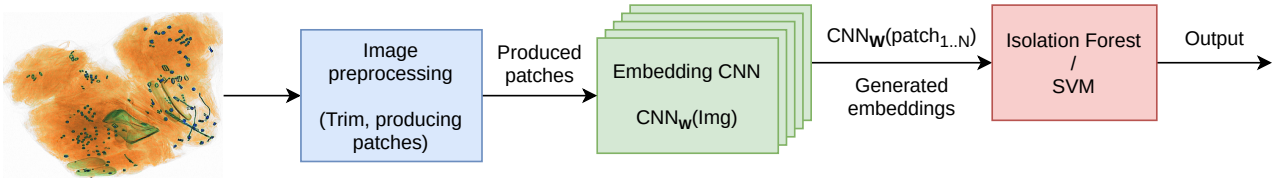


Figure 3.1: A high level architecture of the anomaly detection system

The complete anomaly detection system works in multiple stages. First, an image is loaded, trimmed to remove whitespace and reduced to a set of 224×224 patches. Those image patches are then fed into a Densenet-201 CNN that was pre-trained on ImageNet and then trained to determine whether two images show the same parcel or not. The network takes the patches and produces a set of embeddings, one embedding for each patch. The resulting set of embeddings is passed to an isolation forest that was trained using embeddings generated from the X-ray images dataset. The isolation forest then outputs a set of labels (one for each embedding), 1 if the embedding was normal and -1 if the embedding was anomalous. The resulting classification of the whole image is then taken as a minimum value of the whole set of labels. This means that if at least one embedding was classified as anomalous, the whole image is treated as such.

In total, 4 different configurations of this system, each with a different combination of components, were implemented and evaluated:

Embedding network	Anomaly detection model
Densenet-201 (pre-trained on ImageNet and trained on X-ray dataset)	Linear SVM
Densenet-201 (pre-trained on ImageNet and trained on X-ray dataset)	Isolation forest
Densenet-201 (trained only on ImageNet)	Linear SVM
Densenet-201 (trained only on ImageNet)	Isolation forest

Table 3.1: Different configurations of the anomaly detection system

The following sections describe each component of this system in detail.

3.2 Image pre-processing pipeline

First, all images were trimmed to remove the air around the parcel (which shows up as white in the image). To achieve that, a grayscale image together with thresholding is used to get a bounding box around the contents

of the image. That bounding box is then used to crop the RGB image. If two images are of the same parcel, then they are trimmed in a way that keeps their widths equal.

When training a Siamese or a triplet network, a 224×224 random crop is taken from the image. That specific resolution is used as the networks were pre-trained on images of the same resolution. If the image isn't large enough, it is padded with white pixels before taking the crop. If two images belong to the same parcel, the crops are taken from the same x coordinate. This makes it so the two patches correspond to roughly the same part of the parcel. When the networks are evaluated, a 224×224 center crop is taken instead to keep the validation between epochs consistent.

Each 3-channel RGB image (3×H×W) is normalized using mean = $[0.485, 0.456, 0.406]$ and std = $[0.229, 0.224, 0.225]$ before feeding it to the network. That is required as the CNN models were pre-trained on the ImageNet dataset and they expect their inputs to be normalized using those values.

For anomaly detection, each image is reduced to a set of 224×224 patches using a stride of 112. Those patches are then preprocessed as explained above.

3.3 Embedding networks

The embedding networks are used to generate 1024-D vector representations of the images, which are later fed into an isolation forest model or a linear SVM. The networks are first trained in a Siamese or triplet architecture and then "decoupled" for use in the anomaly detection pipeline (by using the `get_embedding` function as seen in listing 1). The idea is that the CNN generated embeddings potentially have certain features that makes them useful for anomaly detection.

All networks were pre-trained on the ImageNet dataset. Then their final classification layer (a 1000-D dense layer, each dimension corresponding to a unique image class in the ImageNet dataset) was stripped and replaced by a 3-D average pooling layer and a flattening layer after it. That made them output 1024-D vector representations of their inputs.

```python
class SiameseNet(nn.Module):
    def __init__(self, embedding_net):
        super(SiameseNet, self).__init__()
        self.embedding_net = embedding_net

    def forward(self, x1, x2):
        output1 = self.embedding_net(x1)
        output2 = self.embedding_net(x2)
        return output1, output2


    def get_embedding(self, x):
        return self.embedding_net(x)
```

```python
class TripletNet(nn.Module):
    def __init__(self, embedding_net):
        super(TripletNet, self).__init__()
        self.embedding_net = embedding_net

    def forward(self, x1, x2, x3):
        output1 = self.embedding_net(x1)
        output2 = self.embedding_net(x2)
        output3 = self.embedding_net(x3)
        return output1, output2, output3

    def get_embedding(self, x):
        return self.embedding_net(x)
```

Listing 1: Implementation of Siamese and triplet architectures

3.3.1 Siamese networks

The task for the Siamese networks was to determine whether the two images it received as input were of the same parcel or not. If the two pictures show the same parcel, but from a different perspective, the network outputs two embeddings which are very close in 1024-D space, otherwise it outputs embeddings which are far apart.

Multiple Siamese networks were trained and evaluated, each using a different embedding CNN. The embedding networks used were: ResNeXt-101-32×8d [30], Densenet-201 [31], ResNet-50 [32] and Wide ResNet-50-2 [33].

These exact CNNs were selected because they represent the state-of-the-art in image classification performance on the ImageNet dataset. As the task of determining whether two images belong together is relatively similar to image classification, they should also perform quite well here.

All networks were trained for 50 epochs. The contrastive loss function had a margin set to 2. The batch size was set to 16. The Adam optimizer was used (with parameters $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$). The learning rate was initially set to 0.001, but with each epoch it was updated to be 99% of the previous value. All hyperparameters were tuned by experimentation. During training, the networks were fed randomly generated pairs of images, which were preprocessed as explained in the previous section. During validation, the networks were fed pre-determined pairs to maintain consistency between epochs.

After 50 epochs, the networks in order of validation loss (lowest to highest) were: Densenet-201 (0.1081), ResNeXt-101-32×8d (0.11), ResNet-50 (0.1221) and Wide ResNet-50-2 (0.1226). The best model correctly identified whether the images belong to the same parcel 95% of the time.

A Densenet-201 model, which was not pre-trained on ImageNet, was also trained on this task as an experiment to check whether the pre-training makes any difference. Out of all tested models, this one performed the worst. At the end of 50 epochs, its validation loss was 0.1413, which is significantly higher than the loss of the same model pre-trained on ImageNet. This shows that feature discrimination, edge and shape detection learnt during training on ImageNet is useful for this task as well.

Another experiment was carried out by first freezing the pre-trained model's weights and then adding a couple of dense and convolutional layers at the end. This proved to be unsuccessful. The model failed to learn to correctly identify pairs of images and performed no better than random guessing. That is most likely because it didn't have enough trainable parameters and the features generated by the model trained only on ImageNet are not good enough to perform the aforementioned task.

3.3.2 Triplet networks

The task for the triplet networks was to generate embeddings which had a lower distance between the anchor and the positive sample and a longer distance between the anchor and the negative sample. The better it is at this task, the lower the triplet loss. The validation triplet loss was used to evaluate the performance of these networks.

Because the Densenet-201 and ResNeXt-101-32×8d networks performed the best in the Siamese network, they were selected for further testing in a triplet network scenario.

The triplet networks were trained very similarly to the Siamese ones, except the batch size was set to 8 and the triplet margin loss was used with the margin set to 2. During training, the network was fed randomly generated triplets, while during testing it was fed pre-determined ones to keep consistency between epochs.

After 50 epochs, the Densenet-201 model outperformed the ResNeXt-101 one in terms of validation loss (0.0834 and 0.101 respectively). As a result of the Densenet-201 model being the best in the Siamese and triplet case, it was selected to be the embedding network for the final anomaly detection solution.

Various triplet selection methods were tried — random, online semi-hard negative and online hardest negative:

- Random triplet selection: for each anchor and positive sample a random negative is selected. Because no additional embeddings need to be calculated, this method is the fastest.
- Online semi-hard negative selection: for each anchor and positive sample, a negative is selected so that the resulting triplet loss is larger than 0 and smaller than the margin. That negative is selected from a mini-batch of 16 random negatives.
- Online hardest negative selection: for each anchor and a positive sample, a negative is selected so that the resulting triplet loss is larger than 0. In the end, the negative producing the largest loss is selected

out of a mini-bach of 16 random negatives.

In theory, the online hardest negative selection method should result in the best performing network [34]. Curiously, all methods produced very similar results, indicating that this task is not very sensitive to triplet selection or that the network wasn't trained long enough for the selection to matter (i.e. most triplets still produced a loss larger than 0).

3.4 Anomaly detection using the generated embeddings

In the end, the Densenet-201 model trained in a triplet setup was selected to be the embedding network. All tests indicated that this model performed better than the one trained in a Siamese setup.

The 1024-D representations this model generates are then used to train a linear SVM in a supervised manner and an isolation forest in an unsupervised manner. After the training is complete, a novel image can be fed through the network and the generated embedding can be analysed by an SVM/isolation forest.

This is the final stage of the whole pipeline. A set of patches produced by the data preprocessor was previously fed into the embedding CNN, which produced a set of embeddings. That set of embeddings is now passed to the SVM/isolation forest, which produces a set of labels, one for each embedding. The label 1 indicates a normal embedding and -1 indicates an anomalous one. The image is deemed anomalous if at least one label is -1 and normal if all labels are 1.

3.4.1 Using a linear SVM

To determine whether the generated embeddings contain any potential for anomaly detection, a linear SVM was trained to determine whether an embedding is anomalous or not (i.e. whether the image contained a gun or not). A Densenet-201 model trained only on ImageNet was selected as a baseline embedding network for comparison.

This is not the optimal method to do anomaly detection as the vectors generated by the CNN might not be linearly separable, which would render the linear SVM useless. Additionally, supervised learning was used which would not be possible in an anomaly detection setting in the real world, as it is very difficult to get enough labelled data, especially of the anomalous class. This method was only used to verify if the vectors are linearly separable, which would make the final solution much simpler. It was never intended to be the final solution on its own.

The SVMs were trained using the ASGD (Averaged Stochastic Gradient Descent) optimizer [35] and class weights of 1 for the normal class and 50 for the anomalous class. Number of training epochs was set to 50 and the batch size was 256.

3.4.2 Using isolation forests

For the final stage in the anomaly detection pipeline, an isolation forest was selected. It has many benefits over other solutions: low memory usage, great performance, ability to deal with high-dimensional data and, most importantly, it can be trained without many examples of anomalies [36].

To train the isolation forest, a train and validation set was created. The training set had 80041 generated embeddings and the test set had 14125. The embeddings were generated by the triplet Densenet-201 model. The isolation forest was initially configured to have 1024 base estimators in the ensemble. It was then trained using the training set.

Another isolation forest was also trained, but using the embeddings from a Densenet-201 model only trained on ImageNet for comparison purposes.

Chapter 4

Results evaluation

4.1 Overall performance

Embedding network	Anomaly detection model	F1 Macro score	AUROC	Cohen's kappa
Densenet-201 (ImageNet + X-ray)	Linear SVM	0.1849	0.8489	0.1454
Densenet-201 (ImageNet only)	Linear SVM	0.5143	0.9697	0.5009
Densenet-201 (ImageNet + X-ray)	Isolation forest	0.4048	0.6922	0.0908
Densenet-201 (ImageNet only)	Isolation forest	0.5271	0.5383	0.0581

Table 4.1: Results of all anomaly detection system configurations on a validation set of the X-ray dataset

Table 4.1 shows the results achieved by different anomaly detection systems on a validation set containing 14125 1024-D embeddings.

The following metrics were used:

- F1 Macro — the unweighted mean of F1 score of each label. The best value is 1, which indicates perfect precision and recall for every label.
- AUROC — area under the receiving operating characteristic curve. Best value is 1. A random classifier can achieve a value of 0.5.
- Cohen's kappa — a coefficient measuring the agreement between the classifier and the correct labels. The best value is 1, which indicates perfect agreement between the two classifiers. The worst value is -1, which indicates total disagreement. Negative values imply that the classifier is performing worse than random guessing.

Interestingly, the embedding network, trained on ImageNet only, generated very sparse and low variance embeddings, while the model fine-tuned on the X-ray image dataset generated embeddings with high variance. This might mean that the ImageNet model prioritizes more generic features (e.g. edges and shapes) while the X-ray trained model can detect features more relevant to X-ray imaging.

4.1.1 Linear SVM version

The linear SVM version was used as a way to gauge whether the embeddings generated by the model trained on the X-ray dataset were even remotely useful for anomaly detection. If an SVM trained in a supervised manner showed abysmal performance (e.g. a negative Cohen's kappa or an F1 close to 0), that would be a strong indication that those embeddings are completely useless.

The results from table 4.1 show that the SVM trained using embeddings from a CNN that was trained on

ImageNet + X-ray datasets performed significantly worse than the one trained using embeddings from a CNN trained on ImageNet only. This could be an indication that the embeddings generated by an X-ray trained model are not linearly separable in 1024-D space. However, that model didn't perform too badly, which means the embeddings have some potential to be useful.

The performance differential between these models wasn't unexpected. The supervised anomaly detection task is very similar to simple image classification, in which the Densenet-201 (trained on ImageNet only) performs reasonably well.

4.1.2 Isolation forest version

The isolation forest models are used to provide results in a "real-world" scenario. These models were taught in an unsupervised manner, so their performance is a relatively good indication of how useful the embeddings actually are.

The embedding CNN trained on ImageNet + X-ray datasets outperformed the baseline model trained only on ImageNet in all metrics but F1 score. This means that in the process of training the triplet model, the embedding network managed to generate embeddings that have certain features which are useful for anomaly detection.

Overall, the performance of the isolation forest using embeddings from Densenet-201 (ImageNet + X-ray) is mediocre. Even though the model achieved a decent AUROC value of ∼0.7, the Cohen's kappa coefficient indicates that it misclassified many examples. This would not be acceptable in a "real-world" environment, where even one false negative could cause a loss of human life.

The isolation forest's results are by no means state-of-the-art. Previous works using multivariate Gaussian models trained on similar embeddings achieved significantly better performance [14].

4.2 Computational performance

The whole process of loading the image into memory, pre-processing and producing patches, feeding the patches to the CNN and then feeding the generated embeddings to an isolation forest model takes ∼1.5 seconds on a GeForce GTX Titan X GPU and a Intel Xeon E5-2620 v3 CPU. This does not include the time it takes to load the CNN or the isolation forest model into memory.

The biggest bottleneck is passing image patches to the CNN (which includes copying them from RAM to VRAM) and then copying the resulting tensors back to RAM from VRAM, as the isolation forest model isn't loaded to VRAM. The image processing pipeline is also not very well optimized. With careful optimizations, the whole anomaly detection system could analyse an image in ∼1 second (depending on the resolution of the input image).

4.3 Generated embedding suitability for anomaly detection

The results show that the embeddings generated by a network, trained in a Siamese or a triplet architecture to determine whether two X-ray images show the same parcel or not, show minor potential for use in anomaly detection.

The isolation forest model, trained using embeddings from a CNN trained on ImageNet + X-ray dataset outperformed the baseline model. This means that the triplet model's embeddings are more useful for anomaly detection than the baseline ones. Other anomaly detection models, such as the multivariate Gaussian model mentioned in Griffin *et al.* [14], might see an increase in performance using the embeddings from the triplet network, but further testing is required to see if that's the case.

Chapter 5

Conclusions and evaluation

5.1 Summary

This work assessed whether embeddings generated from a CNN, trained to determine whether two X-ray images show the same parcel, but from a different perspective, are useful for anomaly detection.

Multiple CNN architectures were trained in a Siamese and triplet setup to distinguish pairs/triplets of images. Their performance was evaluated and the best performing one was selected as the final embedding network. Online triplet mining impact on the models' performance was also discussed.

The usefulness of the generated embeddings was assessed by training a linear SVM and an isolation forest using those embeddings. 4 configurations of embedding CNN/anomaly detection model were evaluated. The best version of the anomaly detection system achieved $F_1 = 0.4048$, AUROC $= 0.6922$, $\kappa = 0.0908$. This version used embeddings generated by a triplet Densenet-201 model, trained on the X-ray dataset, which shows that embeddings extracted from a CNN, trained to distinguish pairs of X-ray images, can be used for anomaly detection.

Thus, all goals set at the beginning of the project were successfully achieved.

5.2 Future work

Potential future work includes:

- Evaluating the usefulness of these embeddings in a more sophisticated anomaly detection model. Griffin *et al.* [14], for instance, used a multivariate Gaussian model trained on similar embeddings. It would be interesting to see if the performance of that model would increase if it was trained using the embeddings generated by the triplet Densenet-201 model. Other anomaly detection models, such as the Local Outlier Factor or Elliptic Envelope could also be tested and evaluated.
- Evaluating the usefulness of embeddings generated by other CNN architectures. The Densenet-201 model was selected only because it achieved the lowest validation loss in both Siamese and triplet training setups. There is a possibility, that even though the other CNNs finished with a higher loss, their embeddings might contain richer and more useful features for anomaly detection. That can only be determined via extensive testing.
- Deeper analysis of the generated embeddings and a more detailed comparison of embeddings before and after training the network in a Siamese or triplet setup. Currently, only variance and sparsity of the generated embeddings was measured and discussed. There might be other important changes in the

embeddings after the network was trained. A detailed analysis might give more insight into why certain embeddings are good/bad for anomaly detection.

In conclusion, an anomaly detection system, that detects anomalies in X-ray security imaging, can achieve a moderate degree of performance using the embeddings generated by an image classification CNN, that was fine-tuned on a pair finding task in X-ray images. Better performance could most likely be achieved by using a more sophisticated anomaly detection model.

Bibliography

[1] S. Hisamichi and N. I. SUGAWARA, "Mass screening for gastric cancer by x-ray examination," *Japanese journal of clinical oncology*, vol. 14, no. 2, pp. 211–223, 1984.

[2] H. Vogel and D. Haller, "Luggage and shipped goods," *European journal of radiology*, vol. 63, no. 2, pp. 242–253, 2007.

[3] A. Bolfing, T. Halbherr, and A. Schwaninger, "How image based factors and human factors contribute to threat detection performance in x-ray aviation security screening," in *symposium of the Austrian HCI and Usability Engineering Group*, pp. 419–438, Springer, 2008.

[4] G. Chen, "Understanding x-ray cargo imaging," *Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms*, vol. 241, no. 1-4, pp. 810–815, 2005.

[5] G. Tomei, M. Cinti, D. Cerratti, and M. Fioravanti, "Attention, repetitive works, fatigue and stress," *Annali di igiene: medicina preventiva e di comunita*, vol. 18, no. 5, pp. 417–429, 2006.

[6] "Passenger jets carried dubai bomb." https://web.archive.org/web/20101101203811/http://english.aljazeera.net/news/middleeast/2010/10/20101031144429122829.html, 2010. Accessed: 2010-11-01.

[7] M. Mendes, A. Schwaninger, N. Strebel, and S. Michel, "Why laptops should be screened separately when conventional x-ray screening is used," in *2012 IEEE International Carnahan Conference on Security Technology (ICCST)*, pp. 267–273, IEEE, 2012.

[8] B. Burns, "Why do laptops have to be removed when tablets can stay in the bag?," 2012.

[9] J. Andrews, T. Tanay, E. J. Morton, and L. D. Griffin, "Transfer representation-learning for anomaly detection," in *Proceedings of the 33rd International Conference on Machine Learning. JMLR: New York, NY, USA. (2016)*, JMLR, 2016.

[10] A. Mouton and T. P. Breckon, "A review of automated image understanding within 3d baggage computed tomography security screening," *Journal of X-ray science and technology*, vol. 23, no. 5, pp. 531–555, 2015.

[11] T. W. Rogers, N. Jaccard, E. J. Morton, and L. D. Griffin, "Automated x-ray image analysis for cargo security: Critical review and future promise," *Journal of X-ray science and technology*, vol. 25, no. 1, pp. 33–56, 2017.

[12] S. Akcay and T. Breckon, "Towards automatic threat detection: A survey of advances of deep learning within x-ray security imaging," *arXiv preprint arXiv:2001.01293*, 2020.

[13] J. T. Andrews, E. J. Morton, and L. D. Griffin, "Detecting anomalous data using auto-encoders," *International Journal of Machine Learning and Computing*, vol. 6, no. 1, p. 21, 2016.

[14] L. D. Griffin, M. Caldwell, J. T. Andrews, and H. Bohler, ""unexpected item in the bagging area": Anomaly detection in x-ray security images," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 6, pp. 1539–1553, 2018.

[15] J. An and S. Cho, "Variational autoencoder based anomaly detection using reconstruction probability," *Special Lecture on IE*, vol. 2, no. 1, 2015.

[16] "Self-supervised representation learning." https://lilianweng.github.io/lil-log/2019/11/10/self-supervised-learning.html, 2019. Accessed: 2020-04-25.

[17] S. Gidaris, P. Singh, and N. Komodakis, "Unsupervised representation learning by predicting image rotations," *arXiv preprint arXiv:1803.07728*, 2018.

[18] C. Doersch, A. Gupta, and A. A. Efros, "Unsupervised visual representation learning by context prediction," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1422–1430, 2015.

[19] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a" siamese" time delay neural network," in *Advances in neural information processing systems*, pp. 737–744, 1994.

[20] G. Koch, R. Zemel, and R. Salakhutdinov, "Siamese neural networks for one-shot image recognition," in *ICML deep learning workshop*, vol. 2, Lille, 2015.

[21] L. Zheng, S. Duffner, K. Idrissi, C. Garcia, and A. Baskurt, "Siamese multi-layer perceptrons for dimensionality reduction and face identification," *Multimedia Tools and Applications*, vol. 75, no. 9, pp. 5055–5073, 2016.

[22] S. Chopra, R. Hadsell, and Y. LeCun, "Learning a similarity metric discriminatively, with application to face verification," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 1, pp. 539–546, IEEE, 2005.

[23] R. Hadsell, S. Chopra, and Y. LeCun, "Dimensionality reduction by learning an invariant mapping," in *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, vol. 2, pp. 1735–1742, IEEE, 2006.

[24] R. Gómez, "Understanding ranking loss, contrastive loss, margin loss, triplet loss, hinge loss and all those confusing names," 2019.

[25] E. Hoffer and N. Ailon, "Deep metric learning using triplet network," in *International Workshop on Similarity-Based Pattern Recognition*, pp. 84–92, Springer, 2015.

[26] O. Moindrot, "Triplet loss and online triplet mining in tensorflow," 2018.

[27] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[28] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[29] A. Bielski, "Siamese and triplet networks with online pair/triplet mining in pytorch." https://github.com/adambielski/siamese-triplet, 2019.

[30] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1492–1500, 2017.

[31] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.

[32] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[33] S. Zagoruyko and N. Komodakis, "Wide residual networks," *arXiv preprint arXiv:1605.07146*, 2016.

[34] A. Hermans, L. Beyer, and B. Leibe, "In defense of the triplet loss for person re-identification," *arXiv preprint arXiv:1703.07737*, 2017.

[35] W. Xu, "Towards optimal one pass large scale learning with averaged stochastic gradient descent," *arXiv preprint arXiv:1107.2490*, 2011.

[36] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM computing surveys (CSUR)*, vol. 41, no. 3, pp. 1–58, 2009.

Appendix A

Quick guide

The following code is a script to run a novel image through the anomaly detection system.

```
import torch
import tochvision
import cv2
import numpy as np

from networks import TripletNet, EmbeddingNet
from joblib import load
from generate_vectors import produce_patches

model = TripletNet(EmbeddingNet(torchvision.models.densenet201()))

# Map location should be changed depending on whether CUDA is available or not
model.load_state_dict(torch.load('triplet_densenet201_m2.pth', map_location=torch.device('cpu')))
model.eval()
embedding_net = model.embedding_net

isf = load('isolation_forest.joblib')

embeddings = np.empty((0, 1024))

patches = produce_patches(PATH_TO_IMAGE)

for patch in patches:
    with torch.no_grad():
        vec = embedding_net(patch).cpu().detach().numpy()
    if vec.shape == (1024,):
        vec = np.expand_dims(a, axis=0)
    embeddings = np.append(embeddings, vec, axis=0)

y_pred = min(isf.predict(embeddings))
print(y_pred)
```

`triplet_densenet_m2.pth` is a file containing the saved weights of the model, which was trained on the X-ray image dataset.

`isolation_forest.joblib` is a file containing the saved isolation forest, which was trained using the embeddings generated by the model saved in `triplet_densenet_m2.pth`.

The latest source code is provided at https://github.com/mjanonis/anomaly-detection. Any questions can be asked in the issue section.

Appendix B

Source code

B.1 datasets.py

```
# SPDX-License-Identifier: EUPL-1.2
# Unmodified code written by Adam Bielski is licensed under the BSD-3-Clause license
# All further additions and modifications: Copyright (c) 2020, Martynas Janonis
# Licensed under the EUPL-1.2-or-later

import numpy as np
import cv2
import os
import re
import csv

from PIL import Image
from random import shuffle, choice

import torch
from torch.utils.data import Dataset
from torch.utils.data.sampler import BatchSampler
from torchvision.transforms.functional import to_tensor
from torchvision.transforms import Normalize

from trim import trim_xray_images

CROP_MARGIN = 32

# Returns the index of the top view image in a pair
def top_view(pair):
    img1 = int(re.sub("\D", "", pair[0]))
    img2 = int(re.sub("\D", "", pair[1]))
    return int(img1 > img2)


# Returns the index of the side view image in a pair
def side_view(pair):
    img1 = int(re.sub("\D", "", pair[0]))
    img2 = int(re.sub("\D", "", pair[1]))
    return int(img1 < img2)

"""
Generates two .csv files from the root directory with the structure:

IMAGE, POSITIVE_PAIR
"""

def siamses_train_test_csv(root, train_size=0.8):
    # Get all the filepaths of the images
    filepaths = []
    pairs = []
    for subdir, dirs, files in os.walk(root):
        for file in files:
```

```
            filepath = subdir + os.sep + file
            filepaths.append(filepath)

    # Sort the filenames
    filepaths.sort(key=lambda f: int(re.sub("\D", "", f)))

    for i in range(len(filepaths)):
        if i % 2 == 0:
            pairs.append([filepaths[i], filepaths[i + 1]])
        else:
            pairs.append([filepaths[i], filepaths[i - 1]])

    train_samples = int(train_size * len(pairs))

    # Generate the training .csv
    with open("train.csv", "w") as csvfile:
        train = pairs[:train_samples].copy()
        shuffle(train)
        writer = csv.writer(csvfile)
        writer.writerows(train)

    # Generate the testing .csv

    test = pairs[train_samples:]
    for pair in test:
        target = np.random.randint(0, 2)
        if target:
            pair.append(target)
        else:
            pr = pair
            while pr == pair or pr[0] == pair[1]:
                pr = choice(test)
            v1_t = top_view(pair)

            if v1_t == 0:
                pair[1] = pr[side_view(pr)]
            else:
                pair[1] = pr[top_view(pr)]

            pair.append(target)

    with open("test.csv", "w") as csvfile:
        shuffle(test)
        writer = csv.writer(csvfile)
        writer.writerows(test)


# Outputs all pairs to a .csv file
def triplet_train_test_csv(root, train_size=0.8):
    # Get all the filepaths of the images
    filepaths = []
    pairs = []
    for subdir, dirs, files in os.walk(root):
        for file in files:
            filepath = subdir + os.sep + file
            filepaths.append(filepath)

    # Sort the filenames
    filepaths.sort(key=lambda f: int(re.sub("\D", "", f)))

    for i in range(len(filepaths)):
        if i % 2 == 0:
            pairs.append([filepaths[i], filepaths[i + 1]])

    train_samples = int(train_size * len(pairs))
    shuffle(pairs)

    # Generate the training .csv
    with open("triplet_train.csv", "w") as csvfile:
        writer = csv.writer(csvfile)
        writer.writerows(pairs[:train_samples])

    # Generate anchor, positive, negative triplets
```

```
    test = pairs[train_samples:]
    test_set = []
    for idx in range(0, (len(pairs) - train_samples) * 2):
        anchor = test[idx // 2][idx % 2]
        positive = test[idx // 2][(idx + 1) % 2]
        n_idx = idx
        while n_idx == idx:
            n_idx = np.random.randint(0, (len(pairs) - train_samples) * 2)
        negative = test[n_idx // 2][(idx + 1) % 2]
        test_set.append((anchor, positive, negative))

    with open("triplet_test.csv", "w") as csvfile:
        writer = csv.writer(csvfile)
        writer.writerows(test_set)


def svm_train_test_csv(root_neg, root_pos, train_size=0.8):
    # Get all the filepaths of the images
    filepaths = []
    for subdir, dirs, files in os.walk(root_neg):
        for file in files:
            filepath = subdir + os.sep + file
            filepaths.append((filepath, 0))

    for subdir, dirs, files in os.walk(root_pos):
        for file in files:
            filepath = subdir + os.sep + file
            filepaths.append((filepath, 1))

    shuffle(filepaths)
    train_samples = int(train_size * len(filepaths))

    with open("svm_train.csv", "w") as csvfile:
        writer = csv.writer(csvfile)
        writer.writerows(filepaths[:train_samples])

    with open("svm_test.csv", "w") as csvfile:
        writer = csv.writer(csvfile)
        writer.writerows(filepaths[train_samples:])


class SiameseXRayParcels(Dataset):
    def __init__(self, xray_csv, image_size=224, train=True, transform=False):
        self.train = train
        self.transform = transform
        self.pairs = []
        self.image_size = image_size

        # Read the .csv
        with open(xray_csv, newline="") as csvfile:
            reader = csv.reader(csvfile)
            for row in reader:
                self.pairs.append(row)

    def __getitem__(self, index):

        # Final images have to be at least 224x224

        if self.train:
            target = np.random.randint(0, 2)
            img1 = cv2.imread(self.pairs[index][0])
            if target:
                img2 = cv2.imread(self.pairs[index][1])
            else:
                pr = self.pairs[index]
                while pr == self.pairs[index] or pr[0] == self.pairs[index][1]:
                    pr = choice(self.pairs)
                v1_t = top_view(self.pairs[index])
                if v1_t == 0:
                    img2 = cv2.imread(pr[side_view(pr)])
                else:
                    img2 = cv2.imread(pr[top_view(pr)])

```

```
        else:
            img1 = cv2.imread(self.pairs[index][0])
            img2 = cv2.imread(self.pairs[index][1])
            target = int(self.pairs[index][2])

        img1, img2 = trim_xray_images(img1, img2, self.image_size + CROP_MARGIN, target)

        CROP_TOP_MAX_1 = img1.shape[0] - self.image_size
        CROP_TOP_MAX_2 = img2.shape[0] - self.image_size
        CROP_LEFT_MAX_1 = img1.shape[1] - self.image_size
        CROP_LEFT_MAX_2 = img2.shape[1] - self.image_size

        if self.train:
            if self.transform:
                # Apply random cropping
                top_1 = np.random.randint(0, CROP_TOP_MAX_1)
                top_2 = np.random.randint(0, CROP_TOP_MAX_2)
                left_1 = np.random.randint(0, CROP_LEFT_MAX_1)
                left_2 = np.random.randint(0, CROP_LEFT_MAX_2)

                img1 = img1[
                    top_1 : self.image_size + top_1, left_1 : self.image_size + left_1
                ]

                if target:
                    img2 = img2[
                        top_2 : self.image_size + top_2,
                        left_1 : self.image_size + left_1,
                    ]
                else:
                    img2 = img2[
                        top_2 : self.image_size + top_2,
                        left_2 : self.image_size + left_2,
                    ]

            else:
                # Apply a center crop
                top_1 = CROP_TOP_MAX_1 // 2
                top_2 = CROP_TOP_MAX_2 // 2
                left_1 = CROP_LEFT_MAX_1 // 2
                left_2 = CROP_LEFT_MAX_2 // 2

                img1 = img1[
                    top_1 : self.image_size + top_1, left_1 : self.image_size + left_1
                ]
                img2 = img2[
                    top_2 : self.image_size + top_2, left_2 : self.image_size + left_2
                ]

        else:
            # Apply a center crop
            top_1 = CROP_TOP_MAX_1 // 2
            top_2 = CROP_TOP_MAX_2 // 2
            left_1 = CROP_LEFT_MAX_1 // 2
            left_2 = CROP_LEFT_MAX_2 // 2

            img1 = img1[
                top_1 : self.image_size + top_1, left_1 : self.image_size + left_1
            ]
            img2 = img2[
                top_2 : self.image_size + top_2, left_2 : self.image_size + left_2
            ]

        # Normalize using the mean and std of ImageNet
        norm = Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

        # Convert the images to tensors and return
        return (norm(to_tensor(img1)), norm(to_tensor(img2))), target

    def __len__(self):
        return len(self.pairs)
```

```python
class TripletXRayParcels(Dataset):
    def __init__(self, pair_csv, image_size=224, train=True, transform=False):
        self.train = train
        self.transform = transform
        self.pairs = []
        self.image_size = image_size

        # Read the .csv
        with open(pair_csv, newline="") as csvfile:
            reader = csv.reader(csvfile)
            for row in reader:
                self.pairs.append(row)

    def __getitem__(self, index):
        # Prepare the triplet
        if self.train:
            anchor = cv2.imread(self.pairs[index // 2][index % 2])
            positive = cv2.imread(self.pairs[index // 2][(index + 1) % 2])
            n_ind = index
            while n_ind == index:
                n_ind = np.random.randint(0, len(self.pairs) * 2)
            negative = cv2.imread(self.pairs[n_ind // 2][(index + 1) % 2])
        else:
            anchor = cv2.imread(self.pairs[index][0])
            positive = cv2.imread(self.pairs[index][1])
            negative = cv2.imread(self.pairs[index][2])

        # Trim the images
        anchor, positive = trim_xray_images(
            anchor, positive, self.image_size + CROP_MARGIN, 1
        )
        _, negative = trim_xray_images(
            anchor, negative, self.image_size + CROP_MARGIN, 0
        )

        # Get maximum crop size
        CROP_TOP_MAX_A = anchor.shape[0] - self.image_size
        CROP_TOP_MAX_P = positive.shape[0] - self.image_size
        CROP_TOP_MAX_N = negative.shape[0] - self.image_size

        CROP_LEFT_MAX_A = anchor.shape[1] - self.image_size
        CROP_LEFT_MAX_P = positive.shape[1] - self.image_size
        CROP_LEFT_MAX_N = negative.shape[1] - self.image_size

        if self.train:
            if self.transform:
                # Apply random cropping
                top_a = np.random.randint(0, CROP_TOP_MAX_A)
                top_p = np.random.randint(0, CROP_TOP_MAX_P)
                top_n = np.random.randint(0, CROP_TOP_MAX_N)

                left_a = np.random.randint(0, CROP_LEFT_MAX_A)
                # Anchor and positive must be cropped from the same X coordinate
                left_p = left_a
                left_n = np.random.randint(0, CROP_LEFT_MAX_N)

                anchor = anchor[
                    top_a : self.image_size + top_a, left_a : self.image_size + left_a
                ]
                positive = positive[
                    top_p : self.image_size + top_p, left_p : self.image_size + left_p
                ]
                negative = negative[
                    top_n : self.image_size + top_n, left_n : self.image_size + left_n
                ]

            else:
                # Apply a center crop
                top_a = CROP_TOP_MAX_A // 2
                top_p = CROP_TOP_MAX_P // 2
                top_n = CROP_TOP_MAX_N // 2

                left_a = CROP_LEFT_MAX_A // 2
```

```
                left_p = CROP_LEFT_MAX_P // 2
                left_n = CROP_LEFT_MAX_N // 2

                anchor = anchor[
                    top_a : self.image_size + top_a, left_a : self.image_size + left_a
                ]
                positive = positive[
                    top_p : self.image_size + top_p, left_p : self.image_size + left_p
                ]
                negative = negative[
                    top_n : self.image_size + top_n, left_n : self.image_size + left_n
                ]

        else:
            # Apply a center crop
            top_a = CROP_TOP_MAX_A // 2
            top_p = CROP_TOP_MAX_P // 2
            top_n = CROP_TOP_MAX_N // 2

            left_a = CROP_LEFT_MAX_A // 2
            left_p = CROP_LEFT_MAX_P // 2
            left_n = CROP_LEFT_MAX_N // 2

            anchor = anchor[
                top_a : self.image_size + top_a, left_a : self.image_size + left_a
            ]
            positive = positive[
                top_p : self.image_size + top_p, left_p : self.image_size + left_p
            ]
            negative = negative[
                top_n : self.image_size + top_n, left_n : self.image_size + left_n
            ]

        # Normalize using the mean and std of ImageNet
        norm = Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

        # Convert the images to tensors and return
        return (
            (
                norm(to_tensor(anchor)),
                norm(to_tensor(positive)),
                norm(to_tensor(negative)),
            ),
            [],
        )

    def __len__(self):
        if self.train:
            return len(self.pairs * 2)
        else:
            return len(self.pairs)


class XRayParcels(Dataset):
    def __init__(self, csv_file, image_size=224, train=True, transform=False):
        self.train = train
        self.transform = transform
        self.data = []
        self.image_size = image_size

        # Read the .csv
        with open(csv_file, newline="") as csvfile:
            reader = csv.reader(csvfile)
            for row in reader:
                self.data.append(row)

    def __getitem__(self, index):
        img = cv2.imread(self.data[index][0])
        label = np.int(self.data[index][1])

        img, _ = trim_xray_images(img, img, self.image_size + CROP_MARGIN, 0)

        CROP_TOP_MAX = img.shape[0] - self.image_size
```

```
        CROP_LEFT_MAX = img.shape[1] - self.image_size

        if self.train:
            if self.transform:
                # Apply random cropping
                top = np.random.randint(0, CROP_TOP_MAX)
                left = np.random.randint(0, CROP_LEFT_MAX)

                img = img[top : self.image_size + top, left : self.image_size + left]

            else:
                # Apply a center crop
                top = CROP_TOP_MAX // 2
                left = CROP_LEFT_MAX // 2

                img = img[top : self.image_size + top, left : self.image_size + left]
        else:
            # Apply a center crop
            top = CROP_TOP_MAX // 2
            left = CROP_LEFT_MAX // 2

            img = img[top : self.image_size + top, left : self.image_size + left]

        # Normalize using the mean and std of ImageNet
        norm = Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

        return norm(to_tensor(img)), label

    def __len__(self):
        return len(self.data)
```

B.2 generate_vectors.py

```
# SPDX-License-Identifier: EUPL-1.2
# Copyright (c) 2020, Martynas Janonis

# Licensed under the EUPL-1.2-or-later

import os
import re
import numpy as np
import cv2
import math
import torch

import trim

from skimage.util import view_as_windows
from torchvision.transforms import Normalize
from torchvision.transforms.functional import to_tensor


def generate_vectors(root, dest, model):
    model.eval()

    filepaths = []
    for subdir, dirs, files in os.walk(root):
        for file in files:
            filepath = subdir + os.sep + file
            filepaths.append(filepath)

    # Sort the filenames
    filepaths.sort(key=lambda f: int(re.sub("\D", "", f)))

    for file in filepaths:
        print(file)
        patches = produce_patches(file)
        print(len(patches))
        f = open(dest + os.path.splitext(os.path.basename(file))[0] + ".vec", "ab")
        for patch in patches:
            with torch.no_grad():
```

```
                vec = model(patch).cpu().detach().numpy()
            np.savetxt(f, vec, delimiter=",")
        f.close()


def produce_patches(path):
    img = cv2.imread(path)
    patches = []
    # Trim the image
    img = trim.trim_xray_images(img, img, 224, 0)[0]

    # Add a white border around the image to make sure that stride of 112 covers it entirely
    img = cv2.copyMakeBorder(
        img,
        (round_to(img.shape[0], 112) - img.shape[0]) // 2,
        (round_to(img.shape[0], 112) - img.shape[0]) // 2,
        (round_to(img.shape[1], 112) - img.shape[1]) // 2,
        (round_to(img.shape[1], 112) - img.shape[1]) // 2,
        borderType=cv2.BORDER_CONSTANT,
        value=[255, 255, 255],
    )

    window = view_as_windows(img, (224, 224, 3))

    # Normalize using the mean and std of ImageNet
    norm = Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

    for h in range(0, window.shape[0], 112):
        for w in range(0, window.shape[1], 112):
            patches.append(norm(to_tensor(window[h][w][0])).unsqueeze(0))

    return patches


def round_to(num, rnd):
    return int(math.ceil(num / rnd)) * rnd
```

B.3 losses.py

```
# SPDX-License-Identifier: BSD-3-Clause
# Copyright (c) 2019, Adam Bielski

import torch
import torch.nn as nn
import torch.nn.functional as F


class ContrastiveLoss(nn.Module):
    """
    Contrastive loss
    Takes embeddings of two samples and a target label == 1 if samples are from the same class and label == 0 otherwise
    """

    def __init__(self, margin):
        super(ContrastiveLoss, self).__init__()
        self.margin = margin
        self.eps = 1e-9

    def forward(self, output1, output2, target, size_average=True):
        distances = (output2 - output1).pow(2).sum(1)  # squared distances
        losses = 0.5 * (
            target.float() * distances
            + (1 + -1 * target).float()
            * F.relu(self.margin - (distances + self.eps).sqrt()).pow(2)
        )
        return losses.mean() if size_average else losses.sum()


class OnlineContrastiveLoss(nn.Module):
    """
    Online Contrastive loss
```

```
    Takes a batch of embeddings and corresponding labels.
    Pairs are generated using pair_selector object that take embeddings and targets and return indices of positive
    and negative pairs
    """

    def __init__(self, margin, pair_selector):
        super(OnlineContrastiveLoss, self).__init__()
        self.margin = margin
        self.pair_selector = pair_selector

    def forward(self, embeddings, target):
        positive_pairs, negative_pairs = self.pair_selector.get_pairs(
            embeddings, target
        )
        if embeddings.is_cuda:
            positive_pairs = positive_pairs.cuda()
            negative_pairs = negative_pairs.cuda()
        positive_loss = (
            (embeddings[positive_pairs[:, 0]] - embeddings[positive_pairs[:, 1]])
            .pow(2)
            .sum(1)
        )
        negative_loss = F.relu(
            self.margin
            - (embeddings[negative_pairs[:, 0]] - embeddings[negative_pairs[:, 1]])
            .pow(2)
            .sum(1)
            .sqrt()
        ).pow(2)
        loss = torch.cat([positive_loss, negative_loss], dim=0)
        return loss.mean()


class OnlineTripletLoss(nn.Module):
    """
    Online Triplets loss
    Takes a batch of embeddings and corresponding labels.
    Triplets are generated using triplet_selector object that take embeddings and targets and return indices of
    triplets
    """

    def __init__(self, margin, triplet_selector):
        super(OnlineTripletLoss, self).__init__()
        self.margin = margin
        self.triplet_selector = triplet_selector

    def forward(self, embeddings, target):

        triplets = self.triplet_selector.get_triplets(embeddings, target)

        if embeddings.is_cuda:
            triplets = triplets.cuda()

        ap_distances = (
            (embeddings[triplets[:, 0]] - embeddings[triplets[:, 1]]).pow(2).sum(1)
        )  # .pow(.5)
        an_distances = (
            (embeddings[triplets[:, 0]] - embeddings[triplets[:, 2]]).pow(2).sum(1)
        )  # .pow(.5)
        losses = F.relu(ap_distances - an_distances + self.margin)

        return losses.mean(), len(triplets)
```

B.4 metrics.py

```
# SPDX-License-Identifier: EUPL-1.2

# Unmodified code written by Adam Bielski is licensed under the BSD-3-Clause license

# All further additions and modifications: Copyright (c) 2020, Martynas Janonis

```

```python
# Licensed under the EUPL-1.2-or-later

from torch.nn.modules import distance


class Metric:
    def __init__(self):
        pass

    def __call__(self, outputs, target, loss):
        raise NotImplementedError

    def reset(self):
        raise NotImplementedError

    def value(self):
        raise NotImplementedError

    def name(self):
        raise NotImplementedError


class AccumulatedAccuracyMetric(Metric):
    """
    Works with classification model
    """

    def __init__(self):
        self.correct = 0
        self.total = 0

    def __call__(self, outputs, target, loss):
        pred = outputs[0].data.max(1, keepdim=True)[1]
        self.correct += pred.eq(target[0].data.view_as(pred)).cpu().sum()
        self.total += target[0].size(0)
        return self.value()

    def reset(self):
        self.correct = 0
        self.total = 0

    def value(self):
        return 100 * float(self.correct) / self.total

    def name(self):
        return "Accuracy"


class AccumulatedDistanceAccuracyMetric(Metric):
    """
    If the distance between the two outputs is less than the margin,
    classify as positive; else negative
    """

    def __init__(self, margin):
        self.correct = 0
        self.total = 0
        self.margin = margin

    def __call__(self, outputs, target, loss):
        pred = distance.PairwiseDistance().forward(outputs[0], outputs[1])
        pred = pred.flatten() < self.margin
        self.correct += sum(pred == target[0])
        self.total += target[0].size(0)
        return self.value()

    def reset(self):
        self.correct = 0
        self.total = 0

    def value(self):
        return 100 * float(self.correct) / self.total

```

```
80      def name(self):
81          return "Accuracy"
82
83
84  class TripletAccumulatedDistanceAccuracyMetric(Metric):
85      """
86      If the distance between the anchor and the positive is less than the distance between the anchor and the negative,
87      classify as positive; else negative
88      """
89
90      def __init__(self):
91          self.correct = 0
92          self.total = 0
93
94      def __call__(self, outputs, target, loss):
95          dist_pos = distance.PairwiseDistance().forward(outputs[0], outputs[1])
96          dist_neg = distance.PairwiseDistance().forward(outputs[0], outputs[2])
97          pred = dist_pos.flatten() < dist_neg.flatten()
98          self.correct += sum(pred == True)
99          self.total += pred.size(0)
100         return self.value()
101
102     def reset(self):
103         self.correct = 0
104         self.total = 0
105
106     def value(self):
107         return 100 * float(self.correct) / self.total
108
109     def name(self):
110         return "Accuracy"
```

B.5 networks.py

```
1   # SPDX-License-Identifier: EUPL-1.2
2
3   # Unmodified code written by Adam Bielski is licensed under the BSD-3-Clause license
4
5   # All further additions and modifications: Copyright (c) 2020, Martynas Janonis
6
7   # Licensed under the EUPL-1.2-or-later
8
9   import torch.nn as nn
10
11
12  class EmbeddingNet(nn.Module):
13      def __init__(self, network):
14          super().__init__()
15          self.convnet = network
16
17          # Strip the classification layer
18          self.convnet = nn.Sequential(*(list(self.convnet.children())[:-1]))
19          # Add an avg pooling layer (need output shape of [?, 1024, 1, 1])
20          self.aavgp3d = nn.AdaptiveAvgPool3d(output_size=(1024, 1, 1))
21          self.flat = nn.Flatten()
22
23      def forward(self, x):
24          output = self.convnet(x)
25          output = self.aavgp3d(output)
26          output = self.flat(output)
27          return output
28
29      def get_embedding(self, x):
30          return self.forward(x)
31
32
33  class SiameseNet(nn.Module):
34      def __init__(self, embedding_net):
35          super(SiameseNet, self).__init__()
36          self.embedding_net = embedding_net
37
```

```
    def forward(self, x1, x2):
        output1 = self.embedding_net(x1)
        output2 = self.embedding_net(x2)
        return output1, output2

    def get_embedding(self, x):
        return self.embedding_net(x)


class TripletNet(nn.Module):
    def __init__(self, embedding_net):
        super(TripletNet, self).__init__()
        self.embedding_net = embedding_net

    def forward(self, x1, x2, x3):
        output1 = self.embedding_net(x1)
        output2 = self.embedding_net(x2)
        output3 = self.embedding_net(x3)
        return output1, output2, output3

    def get_embedding(self, x):
        return self.embedding_net(x)
```

B.6 train_isolation.py

```
from sklearn.ensemble import IsolationForest
from random import shuffle
from joblib import dump, load

import numpy as np
import os
import re


def load_vectors(root):
    X = np.empty((0, 1024))

    filepaths = []
    for subdir, dirs, files in os.walk(root):
        for file in files:
            filepath = subdir + os.sep + file
            filepaths.append(filepath)

    # Sort the filenames
    filepaths.sort(key=lambda f: int(re.sub("\D", "", f)))

    for file in filepaths:
        a = np.loadtxt(file, delimiter=",")
        if a.shape == (1024,):
            a = np.expand_dims(a, axis=0)

        X = np.append(X, a, axis=0)

    print(X.shape)
    return X


X_train = load_vectors("./vectors")
shuffle(X_train)

isf = IsolationForest(n_estimators=1024, verbose=1)

isf.fit(X_train)

dump(isf, "isolation_forest.joblib")
```

B.7 train_siamese.py

```
# SPDX-License-Identifier: EUPL-1.2

# Unmodified code written by Adam Bielski is licensed under the BSD-3-Clause license

# All further additions and modifications: Copyright (c) 2020, Martynas Janonis

# Licensed under the EUPL-1.2-or-later

import torch
from torch.optim import lr_scheduler
import torch.optim as optim
from torchvision.models import resnext101_32x8d, densenet201

from trainer import fit

cuda = torch.cuda.is_available()
from networks import EmbeddingNet, SiameseNet
from datasets import SiameseXRayParcels
from losses import ContrastiveLoss
from metrics import AccumulatedDistanceAccuracyMetric

siamese_train_dataset = SiameseXRayParcels("train.csv", train=True, transform=True)
siamese_test_dataset = SiameseXRayParcels("test.csv", train=False, transform=False)
batch_size = 1
kwargs = {"num_workers": 1, "pin_memory": True} if cuda else {}
siamese_train_loader = torch.utils.data.DataLoader(
    siamese_train_dataset, batch_size=batch_size, shuffle=True, **kwargs
)
siamese_test_loader = torch.utils.data.DataLoader(
    siamese_test_dataset, batch_size=batch_size, shuffle=False, **kwargs
)

margin = 2.0
embedding_net = EmbeddingNet(resnext101_32x8d(pretrained=True))
model = SiameseNet(embedding_net)
if cuda:
    model.cuda()

loss_fn = ContrastiveLoss(margin)
lr = 1e-3
optimizer = optim.Adam(model.parameters(), lr=lr)
scheduler = lr_scheduler.StepLR(optimizer, 1, gamma=0.99, last_epoch=-1)
n_epochs = 50
log_interval = 10

fit(
    siamese_train_loader,
    siamese_test_loader,
    model,
    loss_fn,
    optimizer,
    scheduler,
    n_epochs,
    cuda,
    log_interval,
    [AccumulatedDistanceAccuracyMetric(margin // 2)],
)
```

B.8 train_svm.py

```
# SPDX-License-Identifier: EUPL-1.2
# Copyright (c) 2020, Martynas Janonis

# Licensed under the EUPL-1.2-or-later

import torch
import sys
import numpy as np

```

```
from torchvision.models import resnext101_32x8d, densenet201
from sklearn.linear_model import SGDClassifier
from sklearn.metrics import f1_score, roc_auc_score, accuracy_score, cohen_kappa_score
from networks import EmbeddingNet, TripletNet
from datasets import XRayParcels
from joblib import dump, load

cuda = torch.cuda.is_available()
device = torch.device("cuda") if cuda else torch.device("cpu")

xray_train_dataset = XRayParcels("svm_train.csv", train=True, transform=True)
xray_test_dataset = XRayParcels("svm_test.csv", train=False, transform=False)
batch_size = 64
kwargs = {"num_workers": 1, "pin_memory": True} if cuda else {}
xray_train_loader = torch.utils.data.DataLoader(
    xray_train_dataset, batch_size=batch_size, shuffle=True, **kwargs
)
xray_test_loader = torch.utils.data.DataLoader(
    xray_test_dataset, batch_size=batch_size, shuffle=False, **kwargs
)

# Initialize and load the embedding network
embedding_net = EmbeddingNet(densenet201())
model = TripletNet(embedding_net)
model.load_state_dict(torch.load("triplet_densenet201_m2.pth", map_location=device))
model.to(device)
model.eval()

# Initialize the SVM
svm = SGDClassifier(
    loss="hinge", verbose=0, class_weight={0: 1, 1: 50}, warm_start=True, average=True
)

n_epochs = 50
highest_f1 = 0

for epoch in range(n_epochs):

    print("Starting epoch {}".format(epoch))
    # Train stage
    # Generate #batch_size vectors to pass as a dataset to the SVM
    for batch_idx, (data, target) in enumerate(xray_train_loader):
        target = target if len(target) > 0 else None
        if not type(data) in (tuple, list):
            data = (data,)

        data = tuple(d.to(device) for d in data)
        if target is not None:
            target = target.to(device)

        with torch.no_grad():
            vectors = model.embedding_net(*data)

        # Convert from PyTorch tensors to NumPy arrays
        vectors = vectors.detach().cpu().numpy()
        target = target.detach().cpu().numpy()

        # Do one epoch of SGD for the SVM
        svm.partial_fit(vectors, target, classes=[0, 1])

        message = "Train: [{}/{} ({:.0f}%)]".format(
            batch_idx * len(data[0]),
            len(xray_train_loader.dataset),
            100.0 * batch_idx / len(xray_train_loader),
        )

        sys.stdout.write("\x1b[2K")  # Clear to the end of line
        sys.stdout.write("\r" + message)
        sys.stdout.flush()

    print()
    print("Starting validation")
    # Test stage
```

```
    # Generate #batch_size vectors to pass as a dataset to the SVM
    y_pred = []
    y_true = []
    for batch_idx, (data, target) in enumerate(xray_test_loader):
        target = target if len(target) > 0 else None
        if not type(data) in (tuple, list):
            data = (data,)

        data = tuple(d.to(device) for d in data)
        if target is not None:
            target = target.to(device)

        with torch.no_grad():
            vectors = model.embedding_net(*data)

        # Convert from PyTorch tensors to NumPy arrays
        vectors = vectors.detach().cpu().numpy()

        y_true = np.append(y_true, target.detach().cpu().numpy())
        y_pred = np.append(y_pred, svm.predict(vectors))

    print(
        "Epoch {}/{}. Validation set: Avg. accuracy: {:.4f}, avg. F1 score: {:.4f}, avg. AUC: {:.4f}, avg. Kappa {:.4f}".format(
            epoch,
            n_epochs,
            accuracy_score(y_true, y_pred),
            f1_score(y_true, y_pred),
            roc_auc_score(y_true, y_pred),
            cohen_kappa_score(y_true, y_pred),
        )
    )

    # Save the model if F1 is larger
    if f1_score(y_true, y_pred) > highest_f1:
        print("F1 score increased, saving model")
        dump(svm, "svm.joblib")
        highest_f1 = f1_score(y_true, y_pred)
```

B.9 train_triplet.py

```
# SPDX-License-Identifier: EUPL-1.2

# Unmodified code written by Adam Bielski is licensed under the BSD-3-Clause license

# All further additions and modifications: Copyright (c) 2020, Martynas Jononis

# Licensed under the EUPL-1.2-or-later

import torch
from torch.optim import lr_scheduler
import torch.optim as optim
from torchvision.models import resnext101_32x8d, densenet201

from trainer import fit

cuda = torch.cuda.is_available()
from networks import EmbeddingNet, TripletNet
from datasets import TripletXRayParcels
from torch.nn import TripletMarginLoss
from metrics import TripletAccumulatedDistanceAccuracyMetric

triplet_train_dataset = TripletXRayParcels(
    "triplet_train.csv", train=True, transform=True
)
triplet_test_dataset = TripletXRayParcels(
    "triplet_test.csv", train=False, transform=False
)
batch_size = 1
kwargs = {"num_workers": 1, "pin_memory": True} if cuda else {}
triplet_train_loader = torch.utils.data.DataLoader(
    triplet_train_dataset, batch_size=batch_size, shuffle=True, **kwargs
```

```
)
triplet_test_loader = torch.utils.data.DataLoader(
    triplet_test_dataset, batch_size=batch_size, shuffle=False, **kwargs
)

margin = 2.0
embedding_net = EmbeddingNet(resnext101_32x8d(pretrained=True))
model = TripletNet(embedding_net)
if cuda:
    model.cuda()

loss_fn = TripletMarginLoss(margin=margin)
lr = 1e-3
optimizer = optim.Adam(model.parameters(), lr=lr)
scheduler = lr_scheduler.StepLR(optimizer, 1, gamma=0.99, last_epoch=-1)
n_epochs = 50
log_interval = 10

fit(
    triplet_train_loader,
    triplet_test_loader,
    model,
    loss_fn,
    optimizer,
    scheduler,
    n_epochs,
    cuda,
    log_interval,
    [TripletAccumulatedDistanceAccuracyMetric()],
)
```

B.10 trainer.py

```
# SPDX-License-Identifier: EUPL-1.2

# Unmodified code written by Adam Bielski is licensed under the BSD-3-Clause license

# All further additions and modifications: Copyright (c) 2020, Martynas Janonis

# Licensed under the EUPL-1.2-or-later

import torch
import numpy as np
from torch.utils.tensorboard import SummaryWriter
import sys


def fit(
    train_loader,
    val_loader,
    model,
    loss_fn,
    optimizer,
    scheduler,
    n_epochs,
    cuda,
    log_interval,
    metrics=[],
    start_epoch=0,
):
    """
    Loaders, model, loss function and metrics should work together for a given task,
    i.e. The model should be able to process data output of loaders,
    loss function should process target output of loaders and outputs from the model

    Examples: Classification: batch loader, classification model, NLL loss, accuracy metric
    Siamese network: Siamese loader, siamese model, contrastive loss
    Online triplet learning: batch loader, embedding model, online triplet loss
    """

    writer = SummaryWriter()
```

```
    lowest_val_loss = float("inf")
    for epoch in range(start_epoch, n_epochs):

        # Train stage
        train_loss, metrics = train_epoch(
            train_loader, model, loss_fn, optimizer, cuda, log_interval, metrics
        )

        message = "Epoch: {}/{}. Train set: Average loss: {:.4f}".format(
            epoch + 1, n_epochs, train_loss
        )
        for metric in metrics:
            message += "\t{}: {}".format(metric.name(), metric.value())
            writer.add_scalar(metric.name() + "/train", metric.value(), epoch)

        writer.add_scalar("Loss/train", train_loss, epoch)

        val_loss, metrics = test_epoch(val_loader, model, loss_fn, cuda, metrics)
        val_loss /= len(val_loader)

        scheduler.step()

        message += "\nEpoch: {}/{}. Validation set: Average loss: {:.4f}".format(
            epoch + 1, n_epochs, val_loss
        )

        writer.add_scalar("Loss/test", val_loss, epoch)

        for metric in metrics:
            message += "\t{}: {}".format(metric.name(), metric.value())
            writer.add_scalar(metric.name() + "/test", metric.value(), epoch)

        print(message)

        if val_loss < lowest_val_loss:
            print(
                "{:.6f} < {:.6f}: saving model_state_dict to ./model.pth".format(
                    val_loss, lowest_val_loss
                )
            )
            lowest_val_loss = val_loss
            torch.save(model.state_dict(), "./model.pth")

        writer.flush()


def train_epoch(train_loader, model, loss_fn, optimizer, cuda, log_interval, metrics):
    for metric in metrics:
        metric.reset()

    model.train()
    losses = []
    total_loss = 0

    for batch_idx, (data, target) in enumerate(train_loader):
        target = target if len(target) > 0 else None
        if not type(data) in (tuple, list):
            data = (data,)
        if cuda:
            data = tuple(d.cuda() for d in data)
            if target is not None:
                target = target.cuda()

        optimizer.zero_grad()
        outputs = model(*data)

        if type(outputs) not in (tuple, list):
            outputs = (outputs,)

        loss_inputs = outputs
        if target is not None:
            target = (target,)
            loss_inputs += target
```

```
112
113         loss_outputs = loss_fn(*loss_inputs)
114         loss = loss_outputs[0] if type(loss_outputs) in (tuple, list) else loss_outputs
115         losses.append(loss.item())
116         total_loss += loss.item()
117         loss.backward()
118         optimizer.step()
119         for metric in metrics:
120             metric(outputs, target, loss_outputs)
121
122         if batch_idx % log_interval == 0:
123             message = "Train: [{}/{} ({:.0f}%)]\tLoss: {:.6f}".format(
124                 batch_idx * len(data[0]),
125                 len(train_loader.dataset),
126                 100.0 * batch_idx / len(train_loader),
127                 np.mean(losses),
128             )
129             for metric in metrics:
130                 message += "\t{}: {}".format(metric.name(), metric.value())
131
132             sys.stdout.write("\x1b[2K")  # Clear to the end of line
133             sys.stdout.write("\r" + message)
134             sys.stdout.flush()
135             losses = []
136
137     # Add a new line
138     print()
139     total_loss /= batch_idx + 1
140     return total_loss, metrics
141
142
143 def test_epoch(val_loader, model, loss_fn, cuda, metrics):
144     with torch.no_grad():
145         for metric in metrics:
146             metric.reset()
147         model.eval()
148         val_loss = 0
149         for batch_idx, (data, target) in enumerate(val_loader):
150             target = target if len(target) > 0 else None
151             if not type(data) in (tuple, list):
152                 data = (data,)
153             if cuda:
154                 data = tuple(d.cuda() for d in data)
155                 if target is not None:
156                     target = target.cuda()
157
158             outputs = model(*data)
159
160             if type(outputs) not in (tuple, list):
161                 outputs = (outputs,)
162             loss_inputs = outputs
163             if target is not None:
164                 target = (target,)
165                 loss_inputs += target
166
167             loss_outputs = loss_fn(*loss_inputs)
168             loss = (
169                 loss_outputs[0] if type(loss_outputs) in (tuple, list) else loss_outputs
170             )
171             val_loss += loss.item()
172
173             for metric in metrics:
174                 metric(outputs, target, loss_outputs)
175
176     return val_loss, metrics
```

B.11 trim.py

```
1 # SPDX-License-Identifier: EUPL-1.2
2 # Copyright (c) 2020, Martynas Janonis
3
```

```
4   # Licensed under the EUPL-1.2-or-later
5
6   import cv2
7   import numpy as np
8
9
10  def get_bounding_box(img):
11      # convert to grayscale
12      gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
13
14      # threshold to get just the contents
15      retval, thresh_gray = cv2.threshold(
16          gray, thresh=205, maxval=255, type=cv2.THRESH_BINARY
17      )
18
19      # find where the contents are and make a cropped region
20
21      # find where the black pixels are
22      points = np.argwhere(thresh_gray == 0)
23
24      # store them in x,y coordinates instead of row,col indices
25      points = np.fliplr(
26          points
27      )
28
29      # create a rectangle around those points
30      x, y, w, h = cv2.boundingRect(points)
31
32      return x, y, w, h
33
34
35  def trim_xray_images(img1, img2, image_size, label):
36      # add a white border around the images to make sure that there's enough space for cropping
37      img1 = cv2.copyMakeBorder(
38          img1,
39          img2.shape[0] // 2 + image_size,
40          img2.shape[0] // 2 + image_size,
41          img2.shape[1] // 2 + image_size,
42          img2.shape[1] // 2 + image_size,
43          borderType=cv2.BORDER_CONSTANT,
44          value=[255, 255, 255],
45      )
46      img2 = cv2.copyMakeBorder(
47          img2,
48          img1.shape[0] // 2 + image_size,
49          img1.shape[0] // 2 + image_size,
50          img1.shape[1] // 2 + image_size,
51          img1.shape[1] // 2 + image_size,
52          borderType=cv2.BORDER_CONSTANT,
53          value=[255, 255, 255],
54      )
55
56      # get bounding box for both images
57      x1, y1, w1, h1 = get_bounding_box(img1)
58      x2, y2, w2, h2 = get_bounding_box(img2)
59
60      # the bounding box has to be at least the size of the image_size
61      if w1 < image_size:
62          x1 -= (image_size - w1) // 2
63          w1 += image_size - w1
64      if h1 < image_size:
65          y1 -= (image_size - h1) // 2
66          h1 += image_size - h1
67
68      if w2 < image_size:
69          x2 -= (image_size - w2) // 2
70          w2 += image_size - w2
71      if h2 < image_size:
72          y2 -= (image_size - h2) // 2
73          h2 += image_size - h2
74
75      # if two images are of the same parcel
76      # make sure that the widths match
```

```
    if label:
        if w1 > w2:
            x2 -= (w1 - w2) // 2
            w2 = w1
        elif w2 > w1:
            x1 -= (w2 - w1) // 2
            w1 = w2

    # adjust the border if x or y is negative
    img1 = cv2.copyMakeBorder(
        img1,
        abs(min(0, y1)),
        0,
        abs(min(0, x1)),
        0,
        borderType=cv2.BORDER_CONSTANT,
        value=[255, 255, 255],
    )
    x1 = max(x1, 0)
    y1 = max(y1, 0)
    img2 = cv2.copyMakeBorder(
        img2,
        abs(min(0, y2)),
        0,
        abs(min(0, x2)),
        0,
        borderType=cv2.BORDER_CONSTANT,
        value=[255, 255, 255],
    )
    x2 = max(x2, 0)
    y2 = max(y2, 0)

    return img1[y1 : y1 + h1, x1 : x1 + w1], img2[y2 : y2 + h2, x2 : x2 + w2]
```

B.12 utils.py

```
# SPDX-License-Identifier: BSD-3-Clause
# Copyright (c) 2019, Adam Bielski

from itertools import combinations

import numpy as np
import torch


def pdist(vectors):
    distance_matrix = (
        -2 * vectors.mm(torch.t(vectors))
        + vectors.pow(2).sum(dim=1).view(1, -1)
        + vectors.pow(2).sum(dim=1).view(-1, 1)
    )
    return distance_matrix


class PairSelector:
    """
    Implementation should return indices of positive pairs and negative pairs that will be passed to compute
    Contrastive Loss
    return positive_pairs, negative_pairs
    """

    def __init__(self):
        pass

    def get_pairs(self, embeddings, labels):
        raise NotImplementedError


class AllPositivePairSelector(PairSelector):
    """
    Discards embeddings and generates all possible pairs given labels.
```

```
    If balance is True, negative pairs are a random sample to match the number of positive samples
    """

    def __init__(self, balance=True):
        super(AllPositivePairSelector, self).__init__()
        self.balance = balance

    def get_pairs(self, embeddings, labels):
        labels = labels.cpu().data.numpy()
        all_pairs = np.array(list(combinations(range(len(labels)), 2)))
        all_pairs = torch.LongTensor(all_pairs)
        positive_pairs = all_pairs[
            (labels[all_pairs[:, 0]] == labels[all_pairs[:, 1]]).nonzero()
        ]
        negative_pairs = all_pairs[
            (labels[all_pairs[:, 0]] != labels[all_pairs[:, 1]]).nonzero()
        ]
        if self.balance:
            negative_pairs = negative_pairs[
                torch.randperm(len(negative_pairs))[: len(positive_pairs)]
            ]

        return positive_pairs, negative_pairs


class HardNegativePairSelector(PairSelector):
    """
    Creates all possible positive pairs. For negative pairs, pairs with smallest distance are taken into consideration,
    matching the number of positive pairs.
    """

    def __init__(self, cpu=True):
        super(HardNegativePairSelector, self).__init__()
        self.cpu = cpu

    def get_pairs(self, embeddings, labels):
        if self.cpu:
            embeddings = embeddings.cpu()
        distance_matrix = pdist(embeddings)

        labels = labels.cpu().data.numpy()
        all_pairs = np.array(list(combinations(range(len(labels)), 2)))
        all_pairs = torch.LongTensor(all_pairs)
        positive_pairs = all_pairs[
            (labels[all_pairs[:, 0]] == labels[all_pairs[:, 1]]).nonzero()
        ]
        negative_pairs = all_pairs[
            (labels[all_pairs[:, 0]] != labels[all_pairs[:, 1]]).nonzero()
        ]

        negative_distances = distance_matrix[negative_pairs[:, 0], negative_pairs[:, 1]]
        negative_distances = negative_distances.cpu().data.numpy()
        top_negatives = np.argpartition(negative_distances, len(positive_pairs))[
            : len(positive_pairs)
        ]
        top_negative_pairs = negative_pairs[torch.LongTensor(top_negatives)]

        return positive_pairs, top_negative_pairs


class TripletSelector:
    """
    Implementation should return indices of anchors, positive and negative samples
    return np array of shape [N_triplets x 3]
    """

    def __init__(self):
        pass

    def get_triplets(self, embeddings, labels):
        raise NotImplementedError
```

```
class AllTripletSelector(TripletSelector):
    """
    Returns all possible triplets
    May be impractical in most cases
    """

    def __init__(self):
        super(AllTripletSelector, self).__init__()

    def get_triplets(self, embeddings, labels):
        labels = labels.cpu().data.numpy()
        triplets = []
        for label in set(labels):
            label_mask = labels == label
            label_indices = np.where(label_mask)[0]
            if len(label_indices) < 2:
                continue
            negative_indices = np.where(np.logical_not(label_mask))[0]
            anchor_positives = list(
                combinations(label_indices, 2)
            )  # All anchor-positive pairs

            # Add all negatives for all positive pairs
            temp_triplets = [
                [anchor_positive[0], anchor_positive[1], neg_ind]
                for anchor_positive in anchor_positives
                for neg_ind in negative_indices
            ]
            triplets += temp_triplets

        return torch.LongTensor(np.array(triplets))


def hardest_negative(loss_values):
    hard_negative = np.argmax(loss_values)
    return hard_negative if loss_values[hard_negative] > 0 else None


def random_hard_negative(loss_values):
    hard_negatives = np.where(loss_values > 0)[0]
    return np.random.choice(hard_negatives) if len(hard_negatives) > 0 else None


def semihard_negative(loss_values, margin):
    semihard_negatives = np.where(
        np.logical_and(loss_values < margin, loss_values > 0)
    )[0]
    return np.random.choice(semihard_negatives) if len(semihard_negatives) > 0 else None


class FunctionNegativeTripletSelector(TripletSelector):
    """
    For each positive pair, takes the hardest negative sample (with the greatest triplet loss value) to create a triplet
    Margin should match the margin used in triplet loss.
    negative_selection_fn should take array of loss_values for a given anchor-positive pair and all negative samples
    and return a negative index for that pair
    """

    def __init__(self, margin, negative_selection_fn, cpu=True):
        super(FunctionNegativeTripletSelector, self).__init__()
        self.cpu = cpu
        self.margin = margin
        self.negative_selection_fn = negative_selection_fn

    def get_triplets(self, embeddings, labels):
        if self.cpu:
            embeddings = embeddings.cpu()
        distance_matrix = pdist(embeddings)
        distance_matrix = distance_matrix.cpu()

        labels = labels.cpu().data.numpy()
        triplets = []

```

```
        for label in set(labels):
            label_mask = labels == label
            label_indices = np.where(label_mask)[0]
            if len(label_indices) < 2:
                continue
            negative_indices = np.where(np.logical_not(label_mask))[0]
            anchor_positives = list(
                combinations(label_indices, 2)
            )  # All anchor-positive pairs
            anchor_positives = np.array(anchor_positives)

            ap_distances = distance_matrix[
                anchor_positives[:, 0], anchor_positives[:, 1]
            ]
            for anchor_positive, ap_distance in zip(anchor_positives, ap_distances):
                loss_values = (
                    ap_distance
                    - distance_matrix[
                        torch.LongTensor(np.array([anchor_positive[0]])),
                        torch.LongTensor(negative_indices),
                    ]
                    + self.margin
                )
                loss_values = loss_values.data.cpu().numpy()
                hard_negative = self.negative_selection_fn(loss_values)
                if hard_negative is not None:
                    hard_negative = negative_indices[hard_negative]
                    triplets.append(
                        [anchor_positive[0], anchor_positive[1], hard_negative]
                    )

        if len(triplets) == 0:
            triplets.append(
                [anchor_positive[0], anchor_positive[1], negative_indices[0]]
            )

        triplets = np.array(triplets)

        return torch.LongTensor(triplets)


def HardestNegativeTripletSelector(margin, cpu=False):
    return FunctionNegativeTripletSelector(
        margin=margin, negative_selection_fn=hardest_negative, cpu=cpu
    )


def RandomNegativeTripletSelector(margin, cpu=False):
    return FunctionNegativeTripletSelector(
        margin=margin, negative_selection_fn=random_hard_negative, cpu=cpu
    )


def SemihardNegativeTripletSelector(margin, cpu=False):
    return FunctionNegativeTripletSelector(
        margin=margin,
        negative_selection_fn=lambda x: semihard_negative(x, margin),
        cpu=cpu,
    )
```

Appendix C

Project plan

COMP0029: Project Plan
Anomaly detection in parcel X-ray images using a Siamese neural network

Martynas Janonis
Supervisor: Lewis Griffin

13th November 2019

Contents

1 Aims and objectives

1.1 Aims

The aim is to develop a working solution that is able to detect anomalies in parcel X-ray images with reasonable accuracy and performance.

1.2 Objectives

- Review the previous approaches to anomaly detection.
- Train a Siamese network to decide if two parcel X-ray images were different views of the same parcel or not.
- Assess the resulting representation as effective or not for anomaly detection.
- Measure how well humans find a pair for an image (i.e. given an image Z of one parcel, does image X or image Y show the same parcel but from a different view).

2 Deliverables

- A fully documented and functional anomaly detection solution.
- A performance and accuracy comparison between the different iterations of the anomaly detection model.
- A strategy for testing and evaluating the solution.

3 Work plan

- Project start to end of October (4 weeks): Literature search and review. Analysing previous anomaly detection solutions.

- Mid-October to mid-November (4 weeks): Refining objectives and starting work on the baseline solution.
- Mid-November to end of November (2 weeks): Completing and evaluating the baseline model.
- End of November to beginning of January (4 weeks): Completing increasingly complex iterations of the model and evaluating their performance.
- Mid-December to mid-February (8 weeks): Have all iterations of the model finished and tested. Fix bugs that were found during testing. Measure human performance on finding image pairs.
- Mid-February to end of March (6 weeks): Work on the Final Report.

Appendix D

Interim report

COMP0029: Interim Report
Anomaly detection in parcel X-ray images using a Siamese neural network

Martynas Janonis
Supervisor: Lewis Griffin

7th February 2020

Contents

1 Progress made and current status

1.1 Baseline solution

Gathered results from the baseline models (DenseNet-201, ResNext-101). Evaluated other architectures but decided not to continue working on them due to their relatively large top-5 error on the ImageNet dataset.

1.2 Model training

Trained a ResNext-101 (32x8d) model on the parcel dataset. Unfortunately, the model stopped learning after $\sim$20 epochs (i.e. the loss stopped decreasing).

The problem was that one of the parameters for the contrastive loss function (the margin) wasn't correct. I am currently training the same model but with the margin set to 2 instead of 1.

2 Remaining work to be done

2.1 Measuring human performance

I am now working on a script that would show three images: an anchor sample, a positive sample and a negative sample. A human would then have to find the correct pair for the anchor sample (i.e. given an image Z of one parcel, does image X or image Y show the same parcel but from a different view). I will then compare human and neural network performance.

2.2 Experimenting with triplet networks

My current approach uses a Siamese network. However, most modern literature describe triplet networks as being superior for similar tasks (e.g. facial recognition). I will train a triplet network and compare its performance to a similar Siamese one.

2.3 Online pair/triplet mining

Currently, during training the network is fed random pairs. After a while, the network can easily differentiate between the easy pairs, but struggles when given a more difficult example.

Online pair/triplet mining would allow the network to learn from only the useful pairs/triplets without having to waste time on pairs/triplets it already knows how to differentiate. This, in theory, should greatly improve the network's performance.