Ecole Polytechnique fédérale de Lausanne

Semester project LCAV Laboratory EPFL

Clustering Autoencoders for Re-Identification

Francesco Ballestrazzi

Supervisor: Adam James Scholefield

Contents

1 Introduction and motivation **2**

2 Autoencoders **3**
2.1 Architecture . . . 3
2.2 Latent space . . . 3
2.3 Autoencoders and clustering . . . 4

3 Clustering losses **5**

4 My setup **6**
4.1 Framework . . . 6
4.2 Network structure . . . 6
4.3 Learning . . . 6
4.4 Clustering . . . 7
4.5 Pre-training . . . 7

5 Results **8**
5.1 Latent space structuring . . . 8
5.2 Clustering new categories . . . 8
5.3 Importance of reconstruction loss . . . 9

6 Conclusions **10**
6.1 Future work . . . 10
6.2 Discussions . . . 10

1 Introduction and motivation

The goal of this project has been, in a first phase, the implementation of an automatic statistics extractor from basketball game videos. Ideally the system would have needed to take as input a video of a full game and output a group of statistics, like distance traveled or heat map of positions on the court, for each player. Since the format of the games could change a lot among videos (e.g different illumination, camera position and rotation), no assumptions of any kind have been made.

Using existing algorithms for object detection and tracking, such as YOLO[6] and DeepSort[8], good results have been obtained for the tracking of the players on the court.

However, to be able to extract full game statistics of a player, it's important to identify each of them uniquely during the game, even if they have been on the bench for some time or they are seen from different angles while on the court.

Since this is a challenging task, crucial for the completion of the original task, the remaining semester time has been spent on it.

The new system would then take as input detection "boxes" that contain a single player at some frame during the video and output an unique id that remain the same for each distinct player during the entire game.

This new task is typically called the Re-Identification problem[1] and many solutions have been proposed in the last years. However, for this project, a new approach has been tested.

The idea here is to use clustering to differentiate among players. Clustering on high dimensionality data (e.g. the players "boxes" images) is not appropriate in this context because only pixels values would be used to distinguish players. For this reason, a pre-processing step is needed to reduce the dimensionality of the data and abstract from pixels values.

Such low-dimensional representation have been obtained using a particular neural network commonly used to learn compressed representations, called Autoencoder[2]. Some custom modifications have been implemented in order to help the subsequent clustering step.

An important thing to keep in mind is that only a subset of images can be seen during training, thus the network need to be able to generalise and map also unseen images to different representations. For example, if only images of players A, B and C have been used for training, the goal is that images of a new player D are mapped to different low-dimensional representations than the ones of A, B and C.

In the first section of this report, the typical Autoencoder structure is described.
In the second one, the custom modifications are presented.
In the third, the setup (e.g. hyper-parameters) used for the experiments is shown, while in the fourth the results are described.
Finally, in the last section, some conclusions and thoughts about future improvements are presented.

The code used for the experiments can be found at https://github.com/freballe/ClusteringAutoencoder.

2 Autoencoders

2.1 Architecture

Autoencoders are composed by two neural networks, an encoder and a decoder. The encoder maps the input image to a low-dimensional vector (the latent variables), while the decoder take this vector and attempts to reconstruct the initial image.

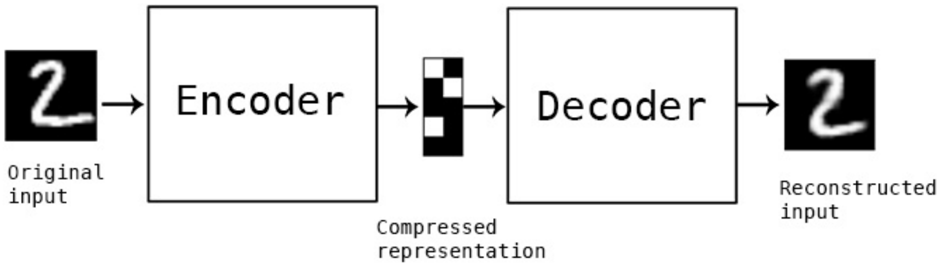

Figure 1: Autoencoder structure. Source: blog.keras.io

Typically, encoder and decoder are symmetric, meaning that they contain the same number and type of layers, but in inverse order. Common choices for such layers are fully connected and convolutional ones.

The two networks are trained to learn mappings that minimize the difference between the original and the reconstructed images. Such difference will be referred as "reconstruction loss".

2.2 Latent space

An interesting aspect of Autoencoders is their ability to learn compressed representations that are well distributed in the low-dimensional space, called the latent space.

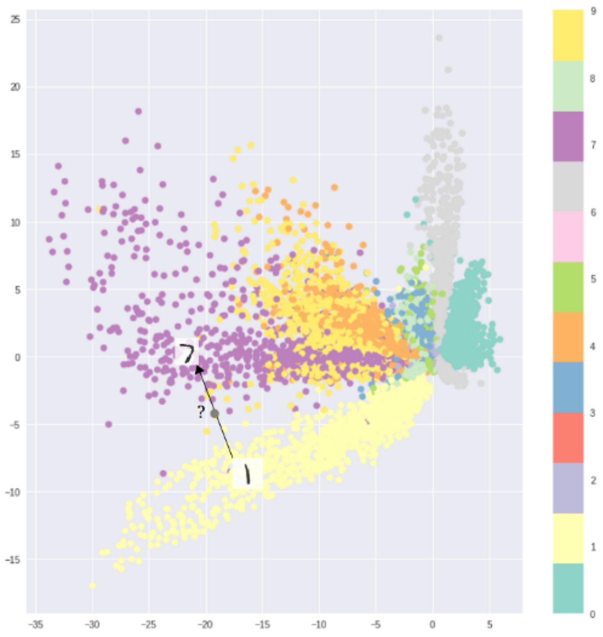

Figure 2: Example of the latent space structure of an Autoencoder trained on the MNIST dataset. Source: towardsdatascience.com

2.3 Autoencoders and clustering

Even if the latent space is already well structured, clustering on it is still unsatisfying since all representations are close and not well separated.

Typically, the way to improve this consists in defining some loss function that pushes learned encodings to be separated into clusters. Many approaches of this type are present in literature[7][3][9], but many of them were designed to work in unsupervised settings to improve clustering accuracy and not to be able to distinguish a new category as a new cluster.

Since for this project the network must not only be able to enforce a different structuring of the space, but also to spread apart unseen categories, two additional metrics have been added to train the network. This ideally should guarantee more freedom in the structure.

3 Clustering losses

The basic Autoencoder training minimise the reconstruction loss

$$\sum_i ||y_i - x_i|| \tag{1}$$

where i is the index of an image in the dataset, x_i is the original image and y_i is the output of the network for such image.

However, since we want to enforce the latent space to assume a clustering structure, two more losses have been added.

One has the goal to spread apart representations of images belonging to different categories. Representations of images of two different players need to be far in the latent space. This one will be referred as the "inter cluster loss".

The other is used to compact together representations of images belonging to the same category. Two images of the same player need to produce similar low-dimensional vectors to facilitate the clustering step. Referred to as "intra cluster loss".

Thus, to train the network, the following loss has been used

$$\sum_{c \in C} \sum_{i \in I_c} (\underbrace{||y_{i,c} - x_{i,c}||^2}_{\text{reconstruction loss}} + \underbrace{\lambda \sum_{\bar{i} \in I_c} ||z_{i,c} - z_{\bar{i},c}||^2}_{\text{intra cluster loss}} \underbrace{- \gamma \sum_{\bar{c} \in C \setminus c} \sum_{\bar{i} \in I_{\bar{c}}} ||z_{i,c} - z_{\bar{i},\bar{c}}||^2)}_{\text{inter cluster loss}} \tag{2}$$

where C represents the set of all categories in the dataset (in our case, all the possible players present at the game), R_c represent all images of category c (all the images of a certain player), z represents the output of the encoder (i.e. the low-dimensional representation) and γ and λ are positive weights that are used to balance the two clustering losses.

4 My setup

4.1 Framework

The network has been implemented in Python, with the library PyTorch[5].

The MNIST dataset [4] has been used in the prototyping phase to verify if the network was able to perform correctly on simple data. The results shown in the next section are the ones obtained with this dataset. Since more experiments still need to be done on the network, no testing has been done on the original players dataset.

4.2 Network structure

The neural network is a standard Dense autoencoder, its implementation in Pytorch is shown here.

```python
class AE(nn.Module):
    def __init__(self, **kwargs):
        super().__init__()
        h1 = 500
        h2 = 64
        latent = 10
        self.encoder = nn.Sequential(
            nn.Linear(kwargs["input_shape"], h1),
            nn.ReLU(),
            nn.Linear(h1, h2),
            nn.ReLU(),
            nn.Linear(h2, latent),
            nn.ReLU()
        )
        self.decoder = nn.Sequential(
            nn.Linear(latent, h2),
            nn.ReLU(),
            nn.Linear(h2, h1),
            nn.ReLU(),
            nn.Linear(h1, kwargs["input_shape"]),
            nn.Sigmoid()
        )

    def forward(self, features):
        code = self.encoder(features)
        reconstructed = self.decoder(code)
        return reconstructed, code
```

4.3 Learning

Equation 2 describe, in theory, which function to optimize to achieve our task. However, in practice, some problems occurred.

Inter cluster loss, the one that maximise distances among representations with different labels, appeared to have much more influence than other losses, pushing all representations to the borders of the latent space.

Moreover, since latent vectors of all images needed to be computed for each image, learning turned out to be very slow.

To overcome the first issue, intra and inter losses have been normalized by the number of samples in the respective sets and additional weights have been added to balance the three losses.

To speed up learning, losses are computed on batches of images and no more on each image individually. Each batch correspond to the set containing all the images with a particular label/category.

The new loss function has been implemented as follows.

```python
# compute encodings
outputs, z = model(images)

# partition encodings and images by label
z_same = z[labels == label]
z_diff = z[labels != label]
outputs_same = outputs[labels == label]
images_same = images[labels == label]

# compute training reconstruction loss
recon_loss = criterion(outputs_same, images_same)
intra_loss = intra_cluster_loss(z_same)
inter_loss = inter_cluster_loss(z_same, z_diff)

inter_loss_weight = intra_loss.item() / inter_loss.item()
alpha = 0.99

total_loss = alpha * recon_loss + (1 - alpha) * (intra_loss -
    inter_loss_weight * inter_loss)
```

where `intra_cluster_loss` and `inter_cluster_loss` are defined as follows and `criterion` correspond to the reconstruction loss.

```python
def intra_cluster_loss(z_same):
    dist = torch.cdist(z_same, z_same)
    dist = torch.mean(dist,dim=1)
    loss = torch.sum(dist)
    loss = torch.div(loss, z_same.
    shape[0])
    return loss
```

```python
def inter_cluster_loss(z_same, z_diff):
    dist = torch.cdist(z_diff, z_same)
    dist = torch.mean(dist,dim=1)
    loss = torch.sum(dist)
    loss = torch.div(loss, z_diff.shape
    [0])
    return loss
```

4.4 Clustering

Once the representations are computed, the Kmeans algorithm is used to cluster them. As a result each low-dimension representation is associated with a cluster number. There is no relation between this number and the original categories identifiers.

The relation between the two is chosen to be, among all possible mappings, the one that maximise the accuracy of the results.

4.5 Pre-training

To facilitate learning, the network has been pre-trained on reconstruction loss alone. The two new losses have been added only after this first phase.

5 Results

In this section the clustering results are presented. The metric used for evaluation is the accuracy. Since the clustering algorithm output values are unrelated to the values used to identify categories, the mapping between the two has been chosen to be the one, among all possible mappings, that maximise the accuracy.

The network has been trained using only the first 5 MNIST digits. Then, unseen categories are the last 5 digits, from 5 to 9.

5.1 Latent space structuring

The new losses had a positive impact on the space structuring and consequently on clustering. In fact, clustering on low-dimension representations, produced by the Autoencoder trained only on reconstruction loss, achieved a 66% accuracy, while when the losses were added the accuracy increased to 76%, on training data.

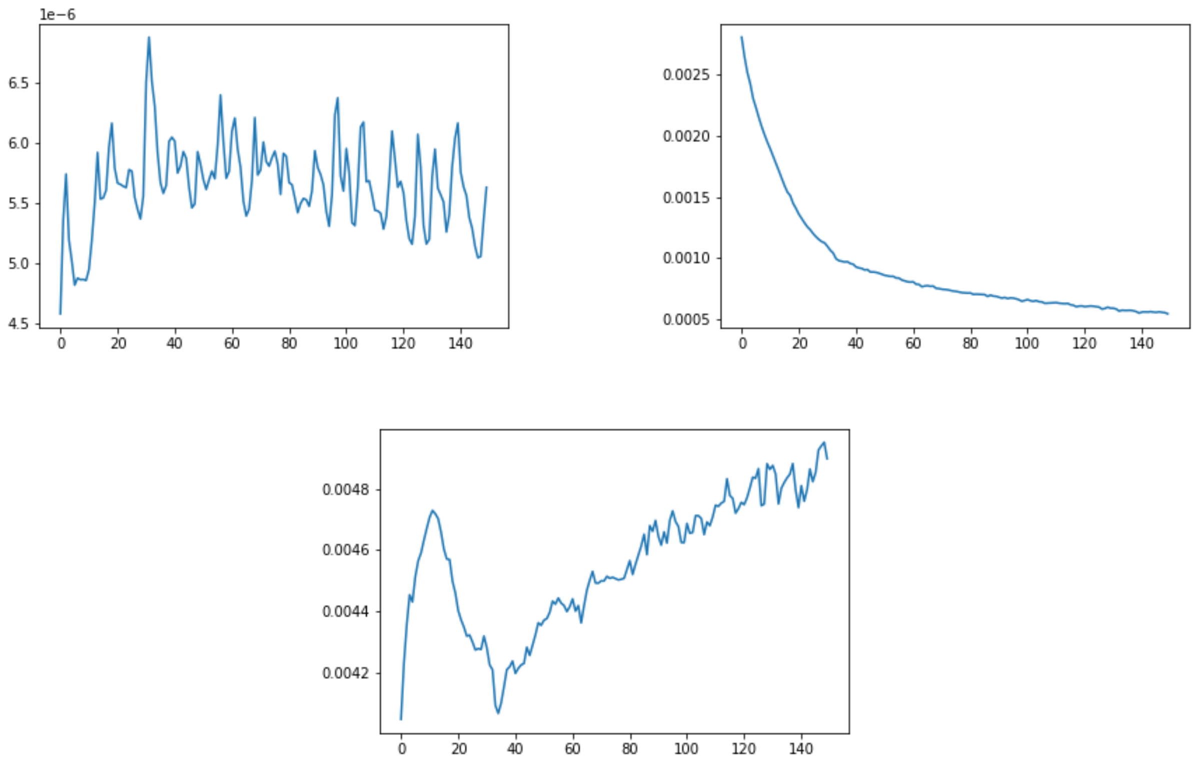

Figure 3: Training losses at each epoch. Reconstruction loss (left), Intra cluster loss (right) and Inter cluster loss (center). As expected, the intra cluster loss decreases, while the inter cluster loss increases.

5.2 Clustering new categories

However, when testing clustering on unseen categories results are quite disappointing. With the same configuration, 45% of accuracy was obtained with cluster losses and 44% with only reconstruction loss.

This is a clear sign of overfitting and lack of generalization power. Even if the losses are useful to further separate different training categories in the latent space, there is no evidence of the ability of the network in doing the same also for new data.

5.3 Importance of reconstruction loss

The use of the reconstruction loss, even while training with the clustering losses, was really important to maintain a structured latent space and to avoid “trivial” encodings. This choice also helped to achieved better results. In fact, the network trained with also reconstruction loss obtained 45% accuracy, while the one with only clustering losses achieved 26% on unseen categories. On categories used for training, accuracy dropped from 76% (with reconstruction loss) to 31% (without reconstruction).

6 Conclusions

6.1 Future work

Given the time constraints and the change of focus during the semester many aspect of the project probably still need to be considered. Some ideas for future work are presented here.

The first aspect to be improved concern the weights to balance losses. Many experiments have been done but no “universal” solution has been found. Weights that depend on the losses intensities could be a good way to get dynamic and adaptable weights that are flexible to various datasets and hyper-parameters.

Another interesting point to explore would be the influence of the reconstruction loss on the performance of the network. After some quick tests it appeared that pre-training (only) on reconstruction helped clustering, but additional tests would be needed to confirm this hypothesis.

Finally, since the last layer of the encoder has a lot of impact on the latent space, various “classical” layers (e.g. Sigmoid, ReLu, Tanh) have been tried. All of those seemed to impose on the space too many constraints, or not enough. In the end, ReLu has been used, but this is definitely something that could be improved. An idea was to use a layer capable of maintaining a fixed variance of the representations, but without imposing strict limits on their values. Maybe imposing a loss between the distribution of representations and a known distribution (a Gaussian for example, similarly to Variational Autoencoders) could be a good starting point.

6.2 Discussions

In this report an Autoencoder approach to the re-identification problem has been presented.

Two new losses have been designed to push the low-dimensional representations of images into clusters. And then clustering algorithms have been applied to cluster data and obtain the category of unseen images.

As a starting point, this approach has been tested on a simple dataset, MNIST, and not on the original one with basketball players.

From the results obtained, this idea worked well to impose a structure on the latent space. However, the generalization power of the network was not sufficient to produce good clustering results on unseen categories. For this reason the code has not been tested on the players dataset.

Given the complexity of the Re-Identification problem, such approach is probably too simple to be able to achieve really competitive results. However, the technique demonstrated some merit in imposing constraints on the Autoencoder learning and it was interesting to explore this possibility.

References

[1] Shaogang Gong et al. “The re-identification challenge”. In: *Person re-identification*. Springer, 2014, pp. 1–20.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[3] Xifeng Guo et al. “Deep clustering with convolutional autoencoders”. In: *International conference on neural information processing*. Springer. 2017, pp. 373–382.

[4] Yann LeCun and Corinna Cortes. “MNIST handwritten digit database”. In: (2010). URL: http://yann.lecun.com/exdb/mnist/.

[5] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[6] Joseph Redmon et al. “You Only Look Once: Unified, Real-Time Object Detection”. In: *CoRR* abs/1506.02640 (2015). arXiv: 1506.02640. URL: http://arxiv.org/abs/1506.02640.

[7] Chunfeng Song et al. “Auto-encoder based data clustering”. In: *Iberoamerican congress on pattern recognition*. Springer. 2013, pp. 117–124.

[8] Nicolai Wojke, Alex Bewley, and Dietrich Paulus. “Simple Online and Realtime Tracking with a Deep Association Metric”. In: *CoRR* abs/1703.07402 (2017). arXiv: 1703.07402. URL: http://arxiv.org/abs/1703.07402.

[9] Junyuan Xie, Ross Girshick, and Ali Farhadi. “Unsupervised deep embedding for clustering analysis”. In: *International conference on machine learning*. 2016, pp. 478–487.