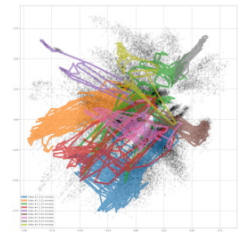


Image and video clustering with an autoencoder

Description	Indexing 100.000 photos or 30 hours of video for discovery.
Period	Fall 2021
Languages & Libs	Python, Keras
Tags	Computer Vision, Autoencoder



This article describes a neural network which automatically projects a large collection of video frames (or images) into 2D coordinates, based on their content and similarity. It can be used to find content such as explosions from Arnold's movies, or car scenes from Bonds. It was originally developed to organize over 6 hours of GoPro footage from Åre bike trip from the summer of 2020, and create a high-res poster which shows the beautiful and varying landscape (Figure 9).

The model's data pipeline and architecture are fairly simple, and its overview shown in Figure 1. The high-res, high-bandwidth video is first downsampled by FFMPEG, keeping only 0.2% of the raw data. This is done by scaling down the video resolution by 80% and dropping 95% of the frames. "Raw" video frames are kept in a [memory-mapped NumPy array](#), for fast random access during visualizations. Although 99.8% of the data is already discarded, 6.2 hours of content still consists of 67k frames which take 32 GB of disk space in a RGB uint8 format.

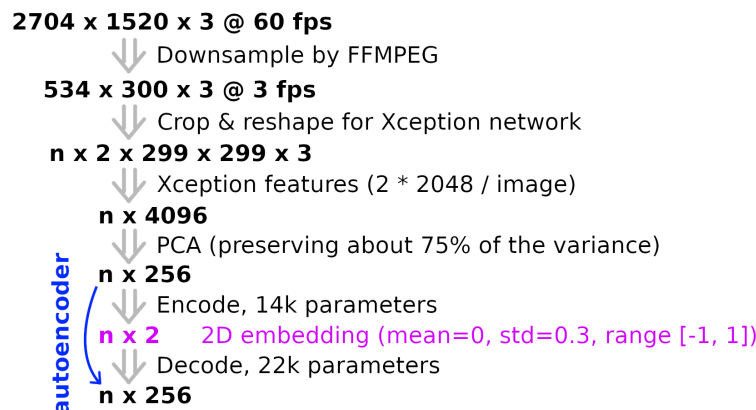


Figure 1: Overview of the data pipeline. The 2D representation (autoencoder's "code" or "embedding") is used to organize the images on a browsable UI, or as descriptors for video clustering & similarity search.

A pre-trained [Xception](#) network is used to extract descriptive features from each frame. It takes in a $299 \times 299 \times 3$ RGB image, and outputs its predictions of what kind of image it is. It has been trained to identify 1000 different categories, such as "lion", "ski", "desk" and "barber chair". Classes are exclusive, so predicted class probabilities add up to 100%. In this case the final prediction layer is dropped, so the model outputs a 2048 dimensional "descriptor" of the image. This is fairly robust to changes in the input image, since its goal is to identify these object categories from very different looking images. Its architecture is shown on Figure 2, and more details can be found from the [publication](#).

The source video has an aspect ratio of 16:9 but the Xception network's input is rectangular. One could just squeeze the input image into a rectangular shape, but here a different approach was used. The input 534×300 image is cropped to left and right halves of size 300×300 and they are scaled to 299×299 . Pixels near the image center are present in both inputs, but that doesn't matter. This reduces the amount of data per image by 99.1%, compressing it down to a representative vector of 4096 dimensions.

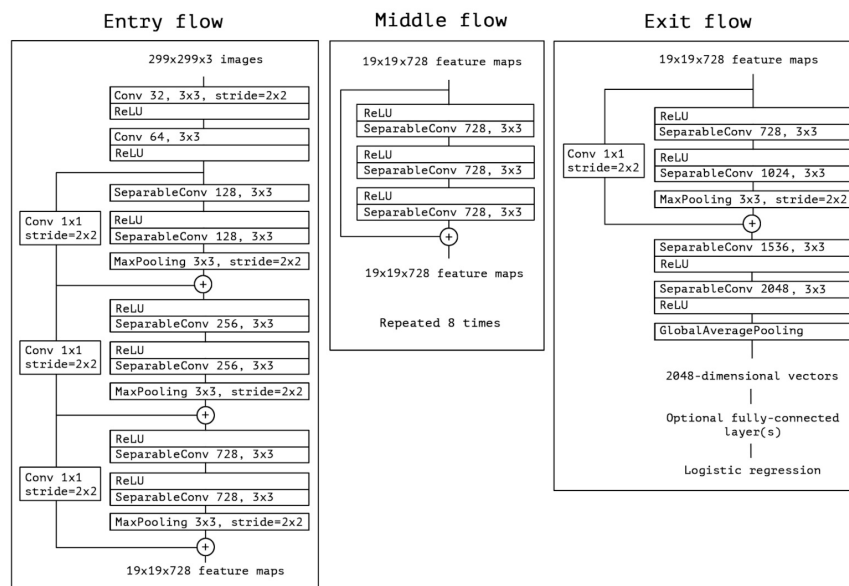


Figure 2: The architecture of the [Xception network](#) (copied from the publication).

The next step is to compress the representation down even further, to just two dimensions. This is desirable since a 2D space is very easy to visualize, and the [autoencoder](#)'s reproduction-accuracy is not relevant in this application. The model can be done more lightweight by preprocessing the data even further before passing it to the autoencoder. One easy solution is to use [Principal Component Analysis](#). It finds linear correlations between features, and determines a linear projection which preserves as much of the variance as possible. It was used to compress the features from 4096 to 256 dimensions (-93.75%) but dropping only about 25% of the "information" (variance).

The final step is to reduce the number of dimensions from 256 to 2 and back to 256, while matching the original input as well as possible in the output. An other solution would have been to use t-SNE here, but an autoencoder gives more control to the result since one can use various regularization methods.

The autoencoder uses only Dense and BatchNormalization layers, but there are quite a few hyperparameters to choose. At the encoding stage, all Dense layers use the elu activation function, except the last one uses tanh which constrains the output to a range of -1 to 1. Elu is used because it is found to work well on regression problems. There are two Dense layers between the input and output layers, and their sizes are determined by weighted geometric means: approximately $(256^2 * 2^1)^{1/3}$ and $(256^1 * 2^2)^{1/3}$.

Layer (type)	Output Shape	Param #
input_132 (InputLayer)	[(None, 256)]	0
dense_183 (Dense)	(None, 50)	12850
batch_normalization_134 (Batch Normalization)	(None, 50)	200
dense_184 (Dense)	(None, 10)	510
batch_normalization_135 (Batch Normalization)	(None, 10)	40
dense_182 (Dense)	(None, 2)	22
Total params: 13,622		
Trainable params: 13,502		
Non-trainable params: 120		

The decoder is very similar, except the dimension gets progressively larger. All Dense layers use the `elu` activation function again, except the last one uses just the `linear` one. It uses more parameters, to ease up the reconstruction. The encoder shouldn't be too complex, because it may lead fairly similar inputs to be far apart in the 2D plane. Intermediate layer sizes are again calculated via a weighted geometric mean.

Layer (type)	Output Shape	Param #
input_133 (InputLayer)	[(None, 2)]	0
dense_185 (Dense)	(None, 6)	18
batch_normalization_136 (Batch Normalization)	(None, 6)	24
dense_186 (Dense)	(None, 22)	154
batch_normalization_137 (Batch Normalization)	(None, 22)	88
dense_187 (Dense)	(None, 76)	1748
batch_normalization_138 (Batch Normalization)	(None, 76)	304
dense_188 (Dense)	(None, 256)	19712
Total params: 22,048		
Trainable params: 21,840		
Non-trainable params: 208		

Additional regularizations are applied to guide the resulting 2D embedding (aka. "coding" or "code"). It is desirable that the point cloud's mean is zero and the standard deviation is "sufficiently" large. Here an std of 0.3 was used, which is sufficiently small that a Gaussian distribution with this std has only 0.1% of samples being outside the range of the tanh activation function. This regularization is easily done in Keras with a [activity regularizer](#).

The second "regularization" technique is similar to a [triplet loss](#) or [contrastive loss](#). The main loss function is still the [MSE](#) of the autoencoder's reconstruction, but the model has a second output as well. The network actually takes three inputs (A, B and C), only the first one of which is used to train the autoencoder. The three images are consecutive frames from a video, so it is expected that they share some similarity with each other. So the model is guided to also minimize $(\text{encode}(A) + \text{encode}(C))/2 - \text{encode}(B)$, meaning that B's 2D embedding should be located between the embedding of A and C.

However this didn't end up working too well, especially with movies where the scene consists of clips from several camera angles. A better solution was to calculate $\log((\epsilon + \text{sum}((A - B)^2))/(\epsilon + \text{sum}((B - C)^2)))$ on a random triplets of A, B & C. The encoder is guided to structure the coding so that the log-ratio applies also to values of $\text{encode}(A)$, $\text{encode}(B)$ & $\text{encode}(C)$. This worked well for biking videos and also for movies.

Example embeddings are shown in Figure 3. The left one has a standard deviation of 0.3, and the right one has 0.6. The latter option didn't result in the desired uniform distribution, rather it pushed lots of points to the margin of the tanh function. The "normalized" version in the middle is explained later in this article.

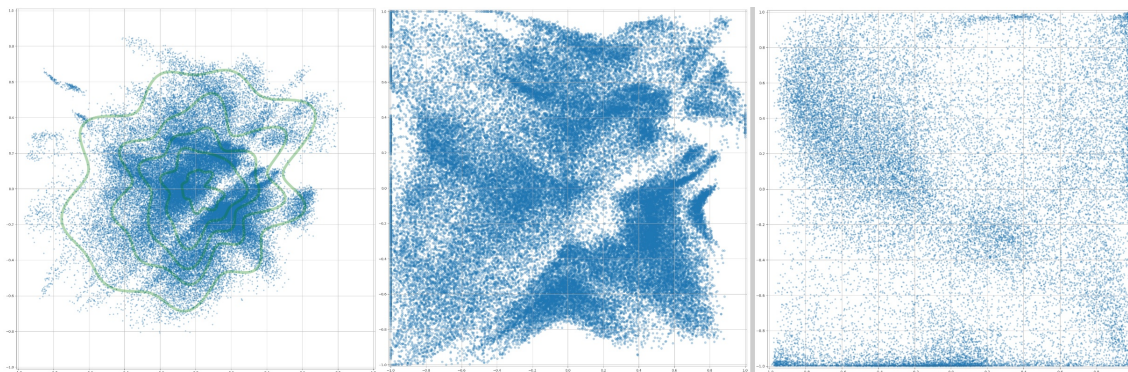


Figure 3: Example distributions from the autoencoder: standard deviation = 0.3 (approximating a Gaussian distribution), a "stretched" version of it (approximating an uniform distribution), and standard deviation = 0.6 (approximating an uniform distribution).

Each point on the scatter plot corresponds to a specific frame from a specific video. This means that we can trace a specific video's "path" within the 2D space, and use it as a fingerprint. Figure 4 shows nine

such paths, which have been chosen to be as dissimilar from each as possible. Their corresponding video frame sequences are shown on the right side, each column corresponding to a specific video. The images are rather small, but it is clear that they have very different scenery. Because of this, there is relatively little overlap between different videos' paths. It is possible to check the small legend on the bottom left to determine how line colors match the shown videos, for example the left-most video is plotted as a blue line and the right-most video is plotted in yellow.

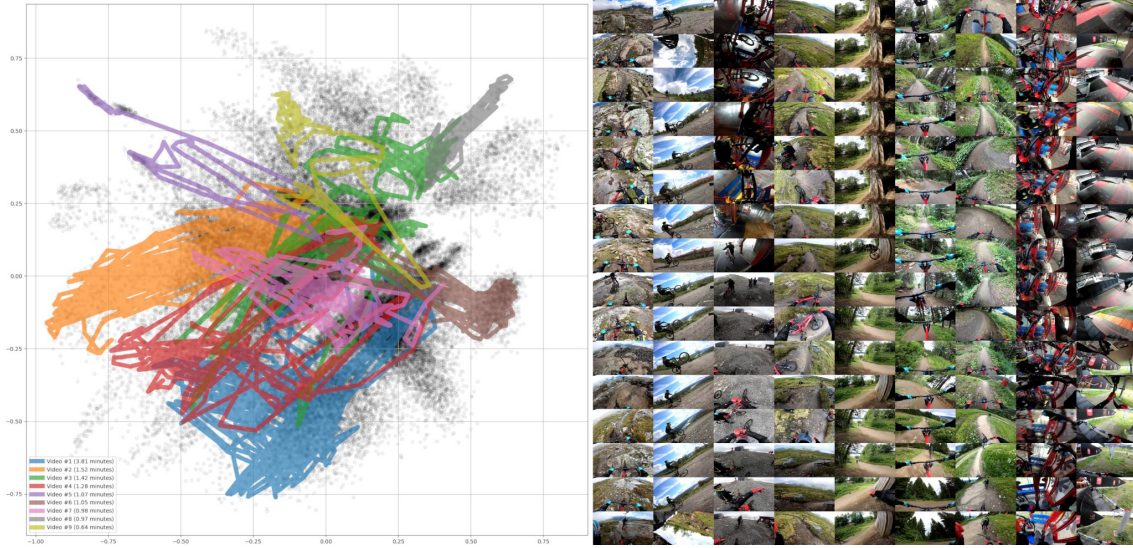


Figure 4: Most dissimilar video clips from the Åre dataset. Each plotted "path" corresponds to a specific video, which are shown on the right.

It is also possible to find as similar video clips as possible, given a query video. These kinds of examples are shown in figures 5 - 8. Videos of Figure 5 have their embeddings mostly at the $y < 0$ region, and they seem to represent a very harsh and rocky terrain. They are filmed at the higher region of the Åre mountain, some frames even show some snow although it was in the middle of the summer. The small blue dots on the images are biking gloves.

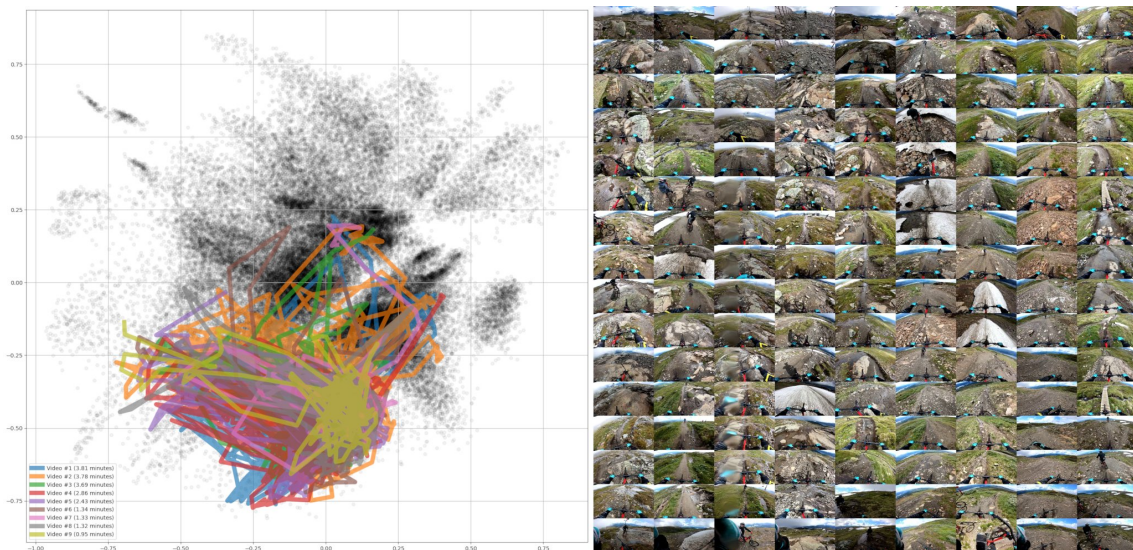


Figure 5: Examples of rocky terrain, from high up in the Åre mountain.

Figure 6 shows videos which have their embeddings at the $|y| < 0.4$ range, which seem to have a bike trail in the middle and greenery on the sides. These are from various locations on the bike park, but of course at a lot lower altitude than the previous ones.

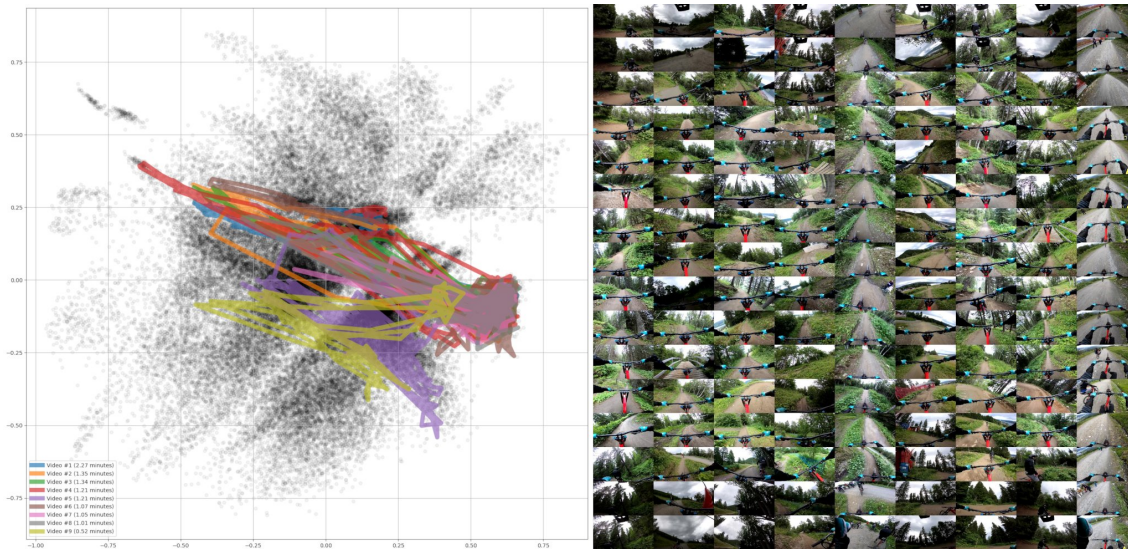


Figure 6: Examples of bike trails, with greenery on both sides.

The final example Figure 7 has quite different scenes. They have an exceptional red color in common, but it originates from a variety of objects (bikes, condola, ...). Videos #1, #3, #6 and #7 are about entering or exiting a red condola, and it also shows our red bike frames. Videos #4, #5, #8 and #9 are from a chair lift, again showing red bike frames. The video #2 is from a queue, showing a red fence.

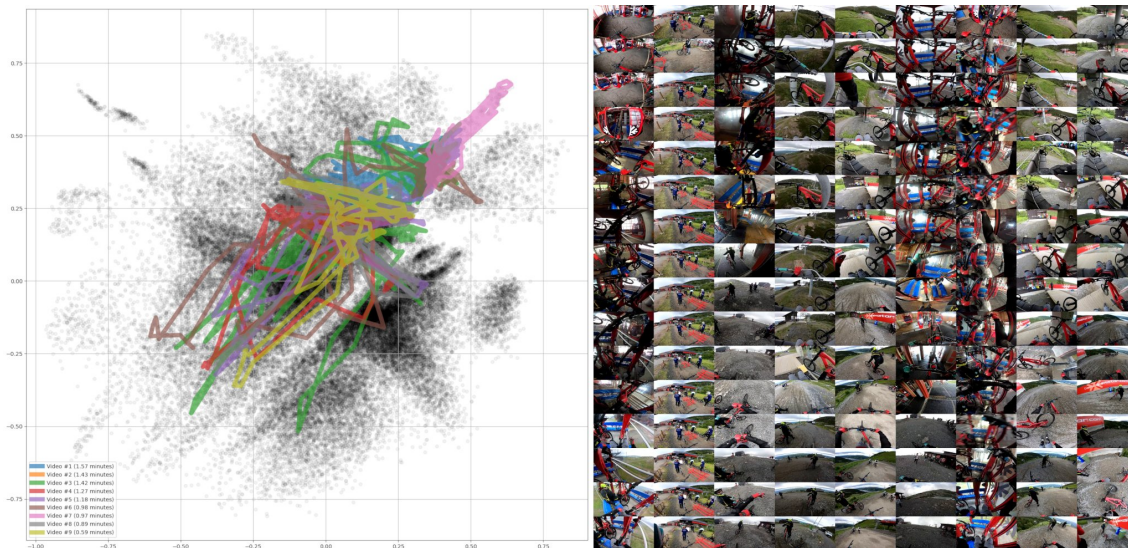


Figure 7: Example of videos from condolas and chair lifts, showing a lot of red bike frames as well.

An uniform distribution at the interval of -1 to 1 has a standard deviation of 0.6, and it was also tested as an activation regularization target. However the result was a bit unexpected, and the 2D embedding is quite sparse in the middle. It pushed a lot of samples to the extremes of the tanh activation function. Most dissimilar videos in this space are shown in Figure 8.

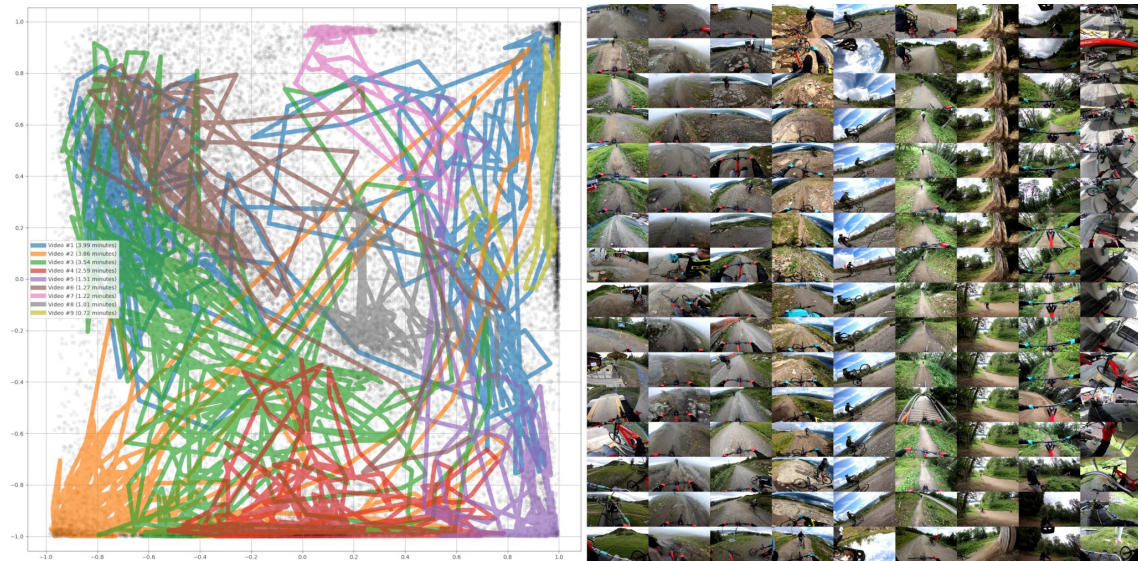


Figure 8: Alternative regularization of the encoder activation, having a stanrad deviation of 0.6.

If we want to visualize the whole embedding space as images, it is desirable that the whole space is covered with images. The left side of Figure 3 has green lines plotted on it, showing the estimated "percentile distance" from the origin. These estimates can then be used to stretch the distribution, so that it would cover the whole plane more equally. The end result is shown in the middle of Figure 3. The result isn't ideal, but better than the starting distribution and better than the distribution on the right (an attempt at uniform distribution by having $\text{std}=0.6$).

Images are tiled on a grid based on their position in the 2D space, and this is shown on the left side of Figure 9. The image is parameterized by four values: middle point's x and y coordinates, the width of the "window" and the number of shown images. The right side shows a zoomed in version of the middle.



Figure 9: Grouped images based on their similarity (autoencoder's 2D coordinates).

Even at the current low-res image cache of 534×300 pixels, this compilation of 11×18 tiles has a resolution of 5874×5400 , which is sufficient for a 0.5 - 1.0 meter wide print. To get a higher DPI we can increase the cached images' resolution, tile more images or seek original frames from high-res MP4 video files.

Random seek on video is fairly slow, but it doesn't matter on a one-time effort. And before generating it we can still use the low-res in-memory cached ones for fast image previews.

If we don't like the composition, we can always re-train the autoencoder and let it converge to a different representation. Although it is expected that it always organizes the images in a more or less stable manner. An other option is to directly manipulate the embeddings, as we already did when we scaled them away from the origin. Also an arbitrary rotation can be applied before scaling.

The final examples at figures 10 - 12 show image clusters from the movie categories: [Arnold Schwarzenegger](#), [James Bond](#) and [Uuno Turhapuro](#). Everybody knows Arnold and Bond, but Uuno isn't well known outside Finland. He is a Finnish comedy character, created in the early 1970s. Each dataset was run separately through the dataflow shown in Figure 1, and the encoded embeddings were re-scaled to fill the 2D plane in a more uniform way (as in the middle scatter plot of Figure 3). Then an UI was used to navigate the embedding space, and representative regions were chosen as examples of grouped images from each film category.

The used Arnold movies are from 1982 - 2002, and in total there are 15 of them. The full compilation is shown on the top left corner of Figure 10, along with zoomed in examples. The top middle image has mainly explosions and pyrotechnics, along with the logo of [Columbia Pictures](#). Arguably the clouds on its background resemble a fire ball. Images on the top right are close-ups of a single face. The bottom left corner has airplanes and helicopters, which are clearly a very distinct and recognizable category. The bottom middle has people in a jungle-like environment. And finally the bottom right has again people, but this time not so close-up but more of a side-pose when compared to the images at top right.

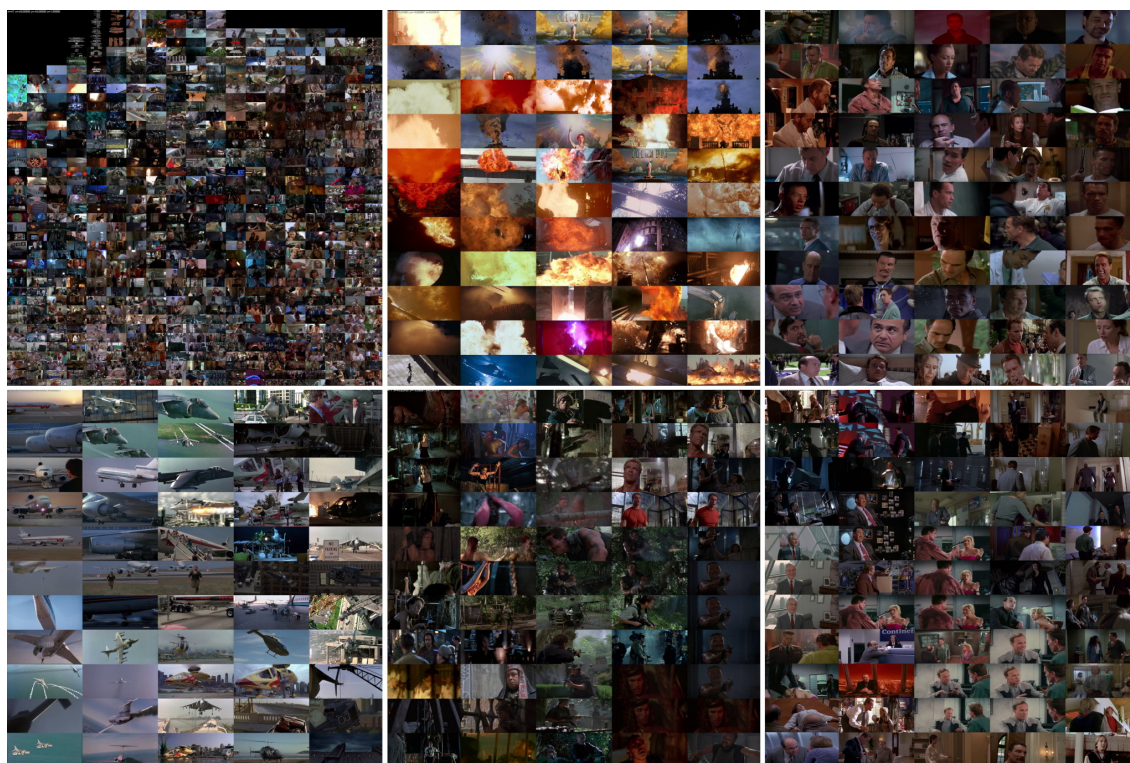


Figure 10: Grouped frames from 15 Arnold Schwarzenegger films.

The used Bond movies are from 1974 - 1999, and in total there are 11 of them. The full compilation is shown on the top left corner of Figure 11, and the other are zoomed in examples. The top middle image has a jail theme, showing bars or a fence. The top right shows close-ups of hands, holding several kinds of objects such as a phone, playing cards, photos or jewelry. The bottom left corner has one or more persons in it, many times showing some bare skin. In many of them Mr. Bond is in bed, but there is also a picture of a sword swallower in there. The bottom middle has close-ups on cars from various angles, but includes

also some helicopters and airplanes. Those images are so zoomed in that it is a bit hard to tell these vehicles apart. The bottom right images are boat & ocean themed.



Figure 11: Grouped frames from 11 James Bond films.

The used Turhapuro movies are from 1984 - 2004, but most of them are from the 1980s. The films from 1973 - 1983 were skipped, since they are black & white. In total 13 movies were used. This dataset didn't have clear distinct image categories, but some beach-themed are shown on the right of Figure 12.

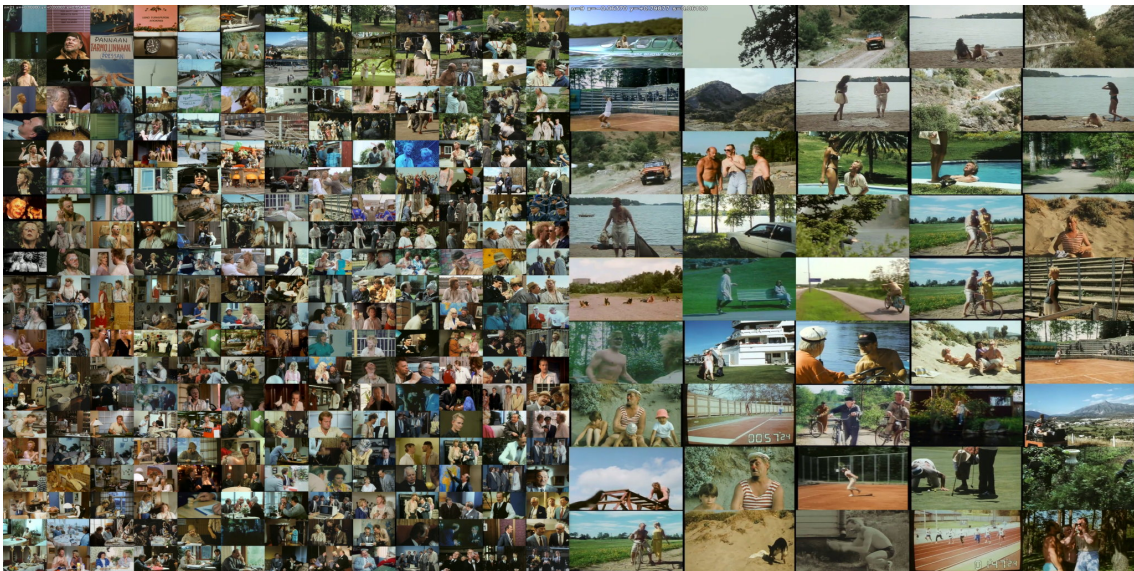


Figure 12: Grouped frames from 13 Uuno Turhapuro films.