Deep Learning Workshop 0368-3538
By: Shay Fux, 313452252
Nathan Bloch, 316130707

Compression By Prediction with Deep Learning

Github: https://github.com/bloch/Deep-Learning-Workshop

Theme. The theme of the project is compression by prediction, and in particular the domain of lossy image compression. In this project, our main goal was to achieve an image compression model using Deep Learning, and to try and apply this model to the domain of lossy video compression.
To achieve these goals, both our dataset and compression models were designed for image compression which can be extended to video compression.
Dataset. Our dataset consists of frames taken from self-made videos of a pacman game. We recorded(using OBS software) video gameplay of a pacman game, and we generated a frames dataset from these recorded videos. We recorded 30 short videos, with technical specs of 30FPS(frames per second) in a resolution of 1920x1080. Due to computations limitations, we converted these videos to resolution of 200x160 (with OpenCV2) in order to have a dataset of a reasonable size. The videos were converted to frames in order to train image compression models. The total length of the videos is about 70 minutes, and this results with 128K frames total. Since the technical specs include 30FPS, many consecutive frames are actually identical, and therefore we extracted only every 10th frame instead of every frame. The Dataset partition is as follows:

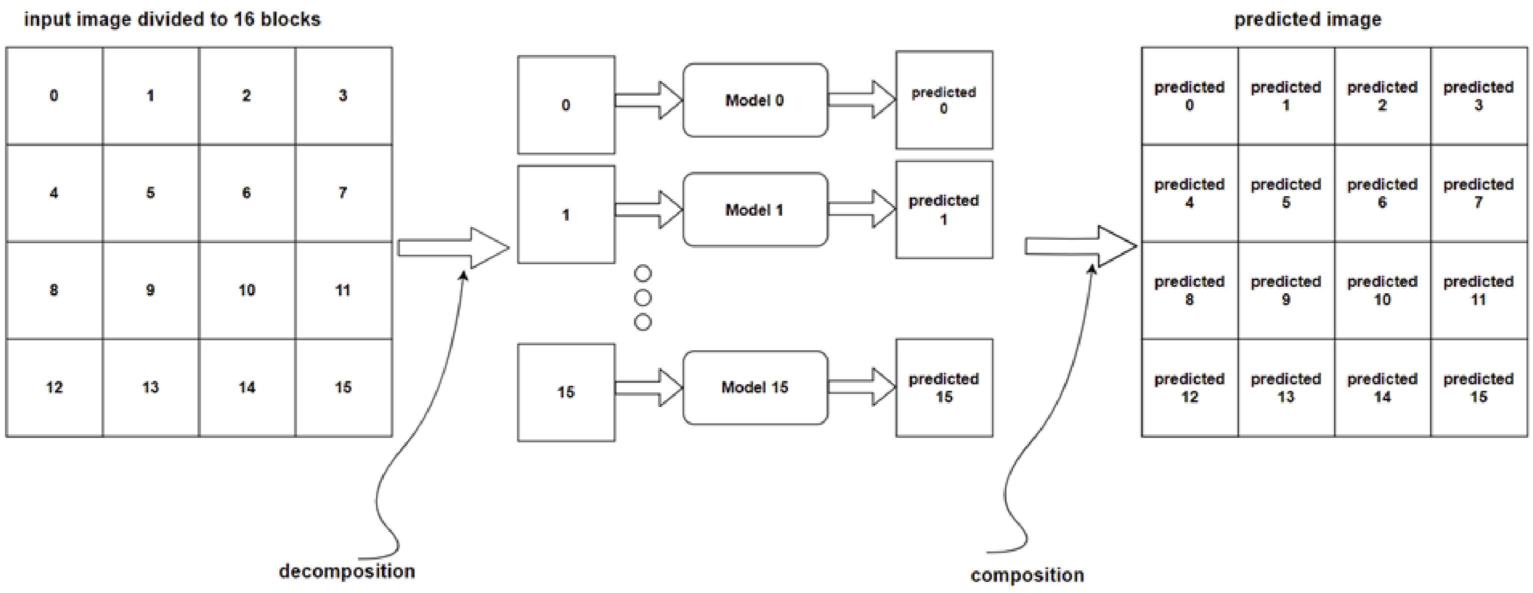
- Training dataset - 10,654 images (videos 1-25).
- Validation dataset - 1,942 images (videos 26-29).
- Test dataset - 491 images (video 30).

Models. All the models we used are based on the architecture of autoencoders. An autoencoder is a type of artificial neural network used to learn efficient codings in an unsupervised fashion. The autoencoder learns a representation(encoding) for a set of data for dimensionality reduction. We have researched several types of such autoencoders, and within each type we tried out different variations. We started off by trying simple fully-connected autoencoders, then moved to convolutional autoencoders and finished with variational autoencoders.

Fully connected Models. We have implemented two types of fully connected networks: Vanilla and Multilayer. In both of these models we encountered a computational difficulty due to the networks' size, therefore we designed a "blocks approach": each frame was divided to 16 blocks, each in a resolution of 50x40, and we trained 16 models, a model per block. However, each one of the 16 models trained only on the corresponding "block dataset" of the training dataset.

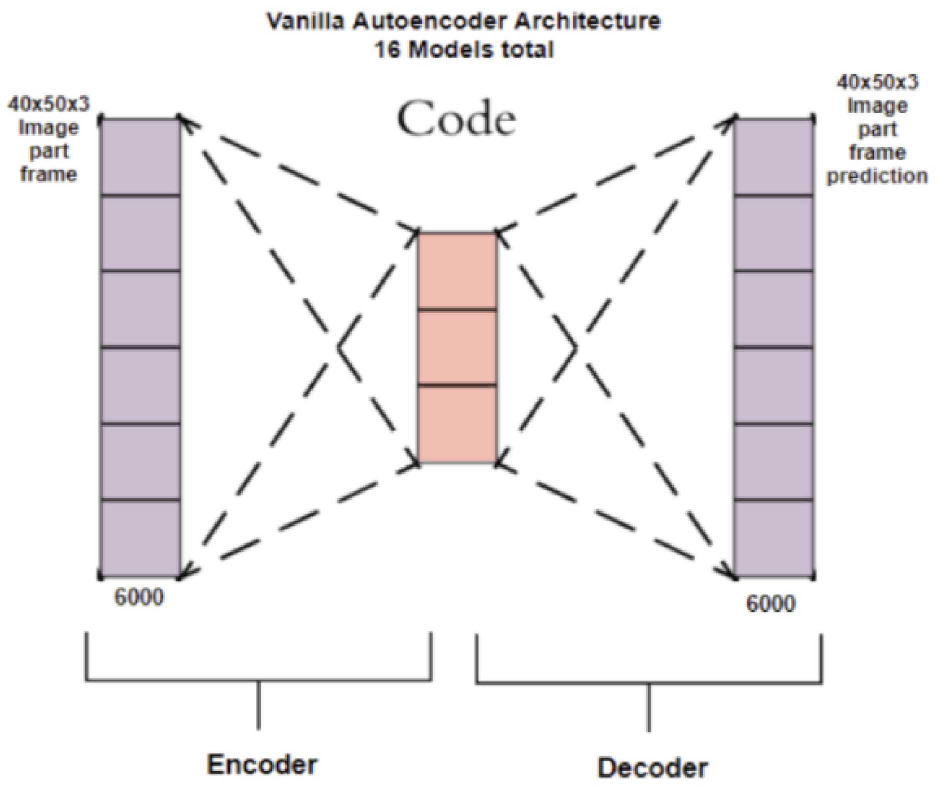
In greater details, the process is as follows:

1. Taking a frame, breaking it into blocks of frame(decomposition).
2. Running each block of frame through the corresponding model, gaining 16 reconstructed blocks of frame.
3. Merging the 16 reconstructed blocks of frames to gain a full reconstructed frame.



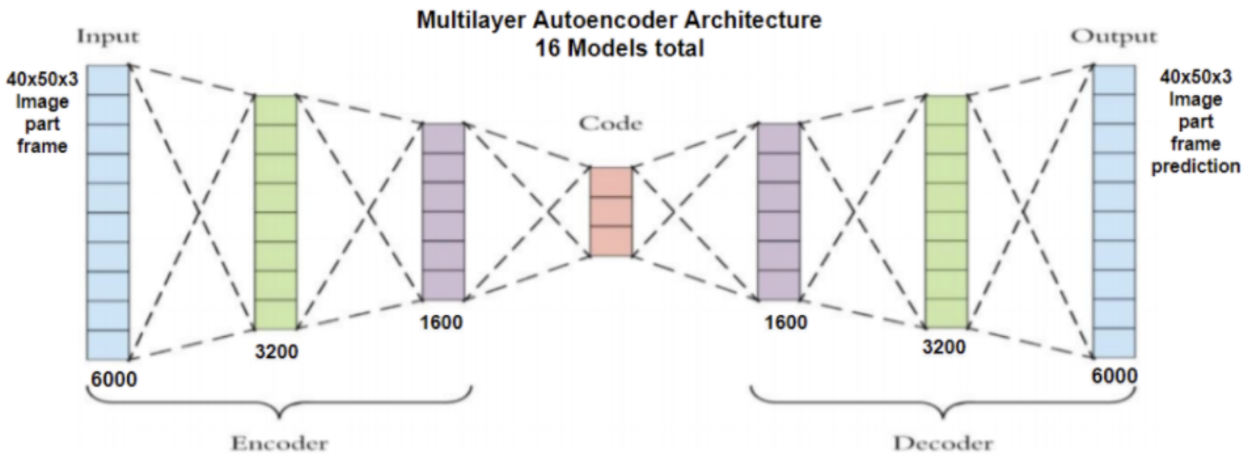
Vanilla Autoencoder. A fully connected NN autoencoder consists of one hidden layer (which is the code layer). The code layer (the compressed representation) is set to 500 or 1000(with compression rate ⅓ and ⅔ respectively, as will be explained at the compression analysis section).

Fully-connected Vanilla Autoencoder Architecture


Vanilla Autoencoder Architecture
16 Models total
40x50x3 Image part frame
Code
40x50x3 Image part frame prediction
6000
6000
Encoder
Decoder

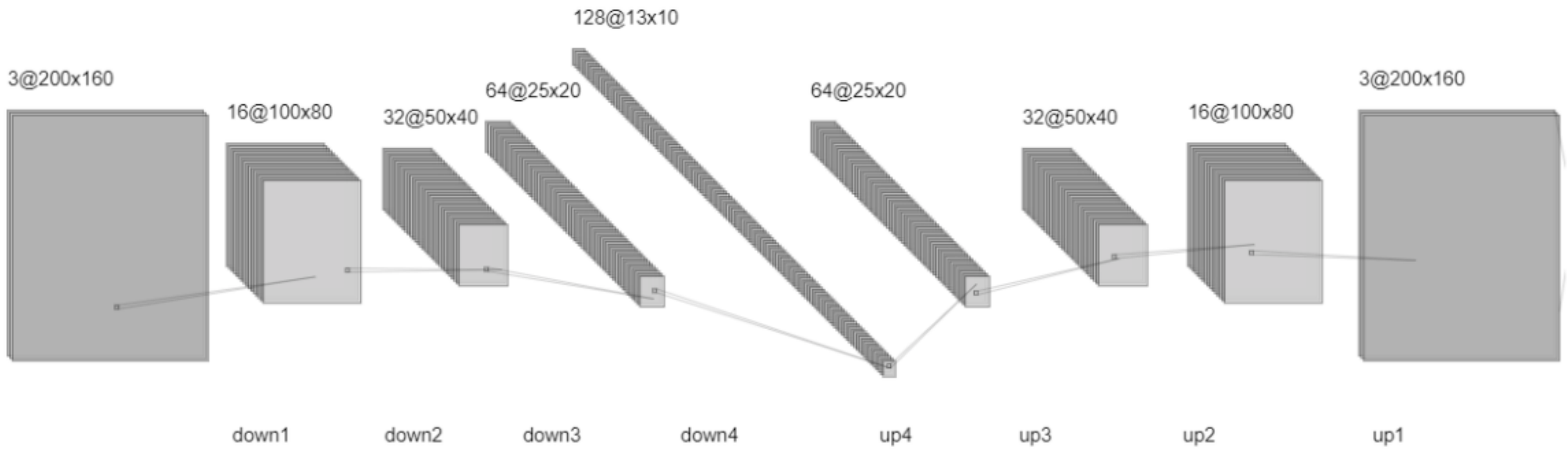
Multilayer Autoencoder. A fully connected NN autoencoder consists of 5 hidden layers where the code layer (the compressed representation) is set to 500/1000(with compression rate ⅓ and ⅔ respectively, as will be explained at the Compression analysis section).

Fully-connected Multilayer Autoencoder Architecture



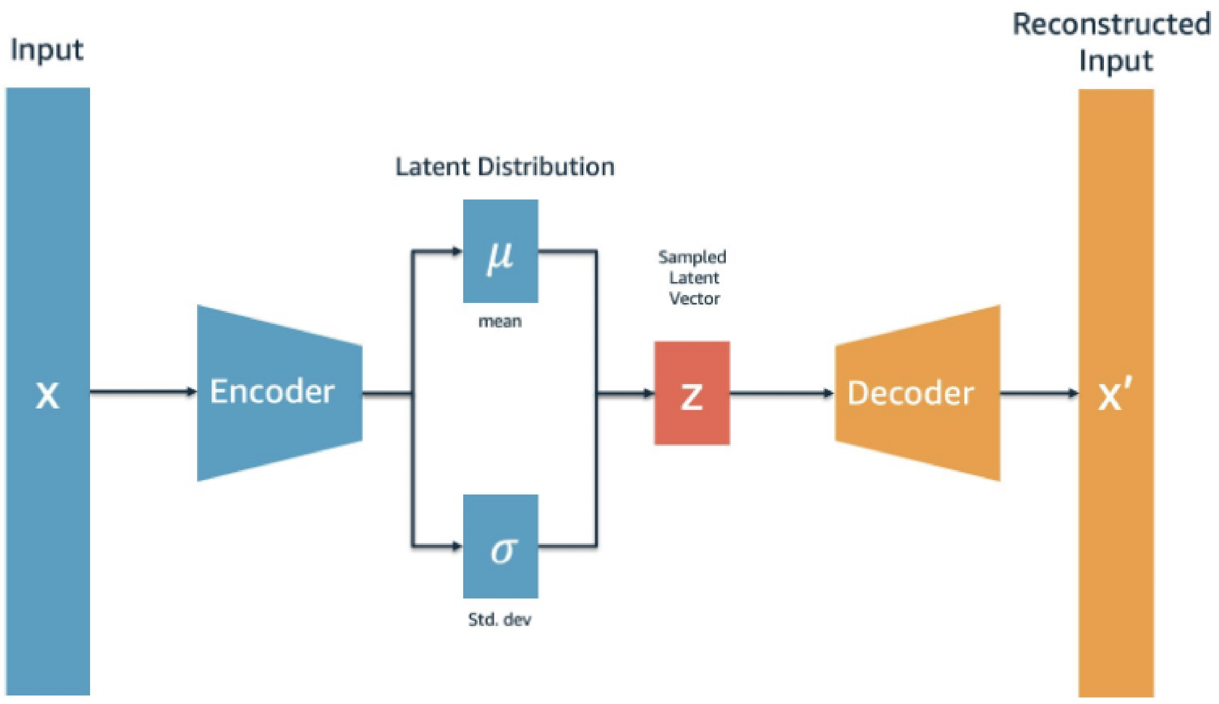
Convolutional Models. We have implemented two Convolutional autoencoders: conv4 and conv5. These models are similar but differ in the number of convolutional layers they have. The conv4 model has 4 layers of downscaling and 4 mirrored layers of upscaling. The conv5 model is an extension of the conv4, and it has one extra downscaling and upscaling layers. Each downscaling layer consists of a Conv2d-BatchNormalization-ReLu-Conv2d -BatchNormalization-ReLu sequence, and each upscaling layer is similar to the corresponding downscaling layer with a small change of Conv2dTranspose operation instead of the Conv2d operation in the downscaling.

Conv4 Autoencoder Architecture (Conv5's Architecture is similar and deeper)



Variational Models. Variational Autoencoders are autoencoders that introduce probabilistic and Bayesian methods in autoencoders. Variational autoencoders are meant to compress the input information into a constrained multivariate latent distribution(encoding) to reconstruct it as accurately as possible(decoding). The compress representation is sampled from the latent distribution , i.e from normal distribution with learned mean (μin the picture) and learned standard deviation (σin the picture). Furthemore, to implement such a model we had to use the reparameterization trick which allows generating a distribution that can be learned. The loss function we used is a combination of a binary cross entropy loss and KL divergence loss(popular loss for such models). This model is different from the previous models because this model's approach is a generative approach.

Classic Scheme of a Variational Autoencoder



Compression Analysis. As mentioned in the Models section, in each model the code layer is the representation of the compressed frame. In order to calculate an accurate compression ratio we must take under consideration the data types of both the input frame entries and both the code layer entries in each model. As mentioned before the input's size is 200x160x3, and each entry is uint8 (1 byte), meaning the size of an input frame is 200x160x3 bytes(96K bytes).
On the other hand the model's layers entries, including the code layer entries, are of type float32(4 bytes), meaning the size of the compressed representation in bytes is 4 times the number of entries in the representation layer.
Therefore the correct compression ratio calculation is:

$$compression\ ratio = ((number\ of\ entries\ in\ compressed\ representation) \cdot 4) \div 96K$$

Since the compressed representations data type gained from the different models are 4 times larger than the input data type, then the challenge of compression becomes harder, and for example, if we compress an image to a representation of ⅓ of the original size, but turn the data type to float32 instead of uint8, then the size of the representation is actually bigger than the input size with a multiplicative factor of 4/3. This problem can be solved in different ways, and we tried out two of them. The first is making the model's compressed representation size small enough(i.e. at least 4 times smaller).

The second option is using quantization. The first approach is a straight-forward solution in which we designed several models with a small enough representation size, but this leads to additional loss as a result of a bigger gap between dimensions of the input and the representation size. The second approach is more tricky and ML algorithmic. There are many techniques for quantization, and we tried several of them:

1. The naive technique: multiplying by 255 and rounding. By statistical observation, in all models the compressed representation entries were numbers between zero to one, so by this assumption we can apply this technique and receive integers in the domain of 0-255, which suit the uint8 data type.
2. Less naive technique: A quantization scheme that will simply be a shifting and scaling of the real number line to a quantized number line. For a given set of real values, we want the minimum/maximum real values in this range [*rmin*,*rmax*] to map to the minimum/maximum integer values [0,2^B-1] respectively, with everything in between linearly distributed. In our case B = 8, as we want to fit in a uint8 type.
   The formula is as follows:

$$r = \frac{r_{max} - r_{min}}{2^B - 1} \cdot q$$

   where r is the real number, and q is the quantized representation. We tried applying this quantization, but this technique didn't yield as good results as the naive one.

3. The third quantization technique was a one that we proposed, and is based on the k-means clustering algorithm, which is a method of vector quantization which tries to partition the data into k clusters. In this method, we applied the k-means algorithm(with k=256) to receive 256 centroids and labels for each entry in the representation layer. The representation layer includes the labels alongside the 256 centroids. This method also didn't yield good results and the computations required made this method infeasible.

Training and Inference Schemes.
In the following table we will summarize our results on the different models on the test set with respect to different quantization methods and compression ratios. As expected, having a smaller representation layer results in a better compression ratio but worse visual results(reconstruction is harder, and the compression is more lossy). All models(except for VAE) were trained with MSE loss, a suitable pixel-wise loss for image reconstruction. We used Adam optimizer, and many PyTorch APIs along the training process for ease of use of datasets and model loading.
As mentioned, the full-sized frames are 200x160x3, and each block is of size 50x40x3, and entries are of uint8 type. That means, each full-sized frame is 96000 bytes and each block is 6000 bytes. These calculations are the explanation to the compression ratios calculated in the table.

A table summarizing the results of all compression models on the test dataset, regarding different parameters such as quantization method and representation layer size as well as reconstruction quality measurements like PSNR and SSIM.

Type	quantization method	Epochs trained	Model size	Representation layer size	Compression ratio	Average PSNR	Average SSIM
Vanilla FC	no quantization	60	768Mb	1000 per block	⅔ (float32 entries)	34.248	0.947
Vanilla FC	naive quantization (method 1)	60	768Mb	1000 per block	⅙ (uint8 entries)	36.013	0.9442
Vanilla FC	scale quantization (method 2)	60	768Mb	1000 per block	⅙ (uint8 entries)	24.751	0.790
Vanilla FC	no quantization	60	385Mb	500 per block	⅙ (float32 entries)	33.714	0.945
Multilayer FC	no quantization	30	3.3Gb	1000 per block	⅔ (float32 entries)	32.230	0.947
Multilayer FC	naive quantization (method 1)	30	3.3Gb	1000 per block	⅙ (uint8 entries)	32.251	0.943
Multilayer FC	scale quantization (method 2)	30	3.3Gb	1000 per block	⅙ (uint8 entries)	24.369	0.790
Multilayer FC	no quantization	30	3.2Gb	500 per block	⅙ (float32 entries)	32.359	0.949
CNN 4 layers	no quantization	320	2.4Mb	16640	~ ⅔ (float32 entries)	30.364	0.874
CNN 5 layers	no quantization	320	9.5Mb	8960	~ ⅓ (float32 entries)	29.022	0.867
VAE	no quantization	220	410Mb	2048	~$\frac{1}{48}$(float32 entries)	27.436	0.908

The number of epochs that each model trained was decided by observation of the loss decay and the saturation of the improvement in the quality of reconstruction(both visually and in terms of PSNR & SSIM).

Lessons Learned.

1. Fully-connected networks might be simple architecture models, but yield good results(especially the Vanilla model). On the other hand, these models are expensive in terms of storage and running time. The inference time is slower compared to CNNs as our data size is considered relatively large. The blocks approach can become handy in terms of running time(parallelizing the process) or RAM usage(work on a different block each time).
2. CNNs models are tricky on the aspect of compression. In order to get a better comparison ratio, it is required to use deeper networks. These deep CNNs did not achieve good results as they lost a lot of information during the encoding. Therefore, many famous architectures(such as UNET, SegNet, etc) use extra data from early encoding layers in order to reconstruct better. In such approaches, there is no actual compression as the representation layer is bigger than the input size. This was our main challenge with CNNs.
3. As observed from training the different models it has been clear that there is a tradeoff between CNN models and fully connected models in terms of computational performance (CNN models outperform in inference speed and memory requirements) and in terms of compression performance(FC models yield much better visual results).
4. As mentioned in the comparison analysis section, quantization is a crucial factor in terms of compression ratio and because it's not part of the learning process it's become harder to train models that include quantization.
5. Generalizing a deep learning compression model to fit arbitrary images is still an unresolved task, and we faced that difficulty as we took a dataset that is very concrete and found out that this mission is not that easy even on such a dataset.

Future Work.

1. Quantization - research and implementation of new techniques that includes traditional ML algorithms(PCA and others) and try to implement quantization through deep learning networks.
2. Our focus was image compression that was used to compress videos. Regarding video compression, other models can be considered and be more suitable for videos, for example, models that rely on time series. For example, such models include LSTMs, RNNs and GANs.

Third Party Tools.

- Video game application(https://www.pacman1.net/)
- Video capturing application - for data collection(OBS software).
- CV & ML libraries: OpenCV2, numpy, pytorch, scikit-learn, scikit-image, matplotlib.

Compute and storage requirements. We used our personal computers for the whole workshop and didn't use any external computational resources.

Models and related works. We didn't base on a specific paper during this project, we however used the web to learn about classic architectures of autoencoders. All models were implemented by us with inspiration to popular ML\Deep Learning websites.
Some(but not all) of the tutorials we have used are -

Medium: General info about Autoencoders
Towards Data Science: Basic tutorial on Autoencoders
Stanford CS231n: Convolutional Neural Networks for Visual Recognition
Jaan.io: VAE autoencoder

Github links to the workshop: https://github.com/bloch/Deep-Learning-Workshop
This github repo contains all code of the project and a detailed readme file with premade demo showcases and instructions for running a live demo on your machine.
Remark: running the live demo takes some effort in downloading & installing everything and therefore we uploaded the premade demo showcases if problems occur with the live demo.
If any problems occur, please contact us.
Nathan Bloch: nathanbloch@mail.tau.ac.il, **052-5403886**
Shay Fux: shayakivafux@mail.tau.ac.il, **054-6602231**