# Deep Learning Assignment 9 in FS 2022

## Convolutional Auto-Encoder

Microsoft Forms Document:
https://forms.office.com/r/ugv3L3jv8i

Manuel Günther

Distributed: Friday, May 6, 2022
Discussed: Friday, May 13, 2022

In this assignment, we show that it is possible to learn from unlabeled data using a convolutional auto-encoder network. We will make use of the images of MNIST which we encode into a 10-dimensional deep feature representation, which we then decode back to the original size of $28 \times 28$ pixels.

To show that the auto-encoder network has learned something useful, we will play around with the encoded deep feature representations and compute averages of all the 10 digits classes. We reconstruct images using the decoder part and see, what happens.

## 1 Dataset

We will make use of the default implementation of the `torch.nn.MNIST` dataset as used in many lectures. However, besides the last two tasks, we do not make use of the labels of the dataset, but we only utilize the images. We instantiate the training and test sets of MNIST – again we will use the test set for validation purposes.

**Datasets** Instantiate the training and validation datasets of MNIST. Select a simple `ToTensor` transform. Instantiate a training data loader using a batch size of $B = 32$, and a validation data loader with 100 samples in a batch.

1. 2D convolutional layer with $Q_1$ channels, kernel size $5 \times 5$, **stride 2** and padding 2
2. activation function ReLU
3. 2D convolutional layer with $Q_2$ channels, kernel size $5 \times 5$, **stride 2** and padding 2
4. activation function ReLU
5. flatten layer to convert the convolution output into a vector
6. fully-connected layer with the correct number of inputs and $K$ outputs

(a) Encoder Network

1. fully-connected layer with $K$ inputs and the correct number of outputs
2. activation function ReLU
3. reshaping to convert the vector into a convolution input
4. 2D **fractionally-strided convolutional** layer with $Q_2$ channels, kernel size $5 \times 5$, stride 2 and padding 2
5. activation function ReLU
6. 2D **fractionally-strided convolutional** layer with $Q_1$ channels, kernel size $5 \times 5$, stride 2 and padding 2

(a) Decoder Network

Table 1: Network configurations of the (a) encoder and (b) decoder networks

## 2 Auto-Encoder Network

The auto-encoder network is composed of two parts: the encoder that transforms the input image to a deep feature representation; and the decoder that produces an image from such a deep feature.

For the encoder $\mathcal{E}$, we will basically use the same convolutional network topology as in the last assignment. An exception is that we perform our down-sampling via striding and not via pooling. After each convolution, we apply the ReLU activation. The output of the encoder is a $K = 10$ dimensional deep feature representation. The complete encoder network topology can be found in Table 1(a).

The decoder $\mathcal{D}$ performs the inverse operations of the encoder. A fully-connected layer is used to increase the number of samples to the same size as the output of the flattening of the encoder. Then, the flattening needs to be undone by reshaping the vector into the correct dimensionality, followed by a ReLU activation. A fractionally-strided convolutional layer increases the intermediate representation by a factor of 2. Note that the fractionally-strided convolution is implemented in `torch.nn.ConvTranspose2d`, and the `stride` parameter should have the same value as for the encoder. Additionally, the `torch.nn.ConvTranspose2d` has a parameter `output_padding` which needs to be adapted to reach the correct output shape (see Test 1). After this layer, we perform another ReLU activation and another fractionally-strided convolution to arrive at the original input dimension. The complete decoder network topology can be found in Table 1(b).

Finally, we combine the two sub-networks into one auto-encoder network. While there exist several possibilities of doing this, we will implement a third `torch.nn.module` that contains an instance of the encoder and an instance of the decoder.

**Task 2: Encoder Network** Implement the encoder network for given parameters $Q_1$, $Q_2$ and $K$ as given in Table 1(a). Implement a network class that derives from `torch.nn.Module` and implements the `__init__` and the `forward` methods.

**Task 3: Decoder Network** Implement the decoder network for given parameters $Q_1$, $Q_2$ and $K$ as given in Table 1(b). Implement a network class that derives from `torch.nn.Module` and implements the `__init__` and the `forward` methods. The output of the decoder network is supposed to have values in range $[0, 1]$, similarly to the input values. We need to make sure that only these values can be achieved. How can we implement this?

**Task 4: Joint Auto-Encoder Network** Implement the auto-encoder network by combining the encoder and the decoder. In the `__init__` function, instantiate an encoder from Task 2 and a decoder from Task 3. In forward, pass the input through the encoder and the decoder: $\mathbf{Y} = \mathcal{D}(\mathcal{E}(\mathbf{X}))$

**Test 1: Output Sizes** Instantiate the auto-encoder network with $Q_1 = Q_2 = 32$ and $K = 10$. Create an input $\mathbf{X}$ in the size that the auto-encoder requires. Provide that input to the (untrained) encoder part of the auto-encoder network to extract the deep feature representation. Check that the deep feature is in the desired size. Provide the deep feature to the (untrained) decoder part of the auto-encoder network. Check that the output is of dimension $28 \times 28$, and its values are between 0 and 1.

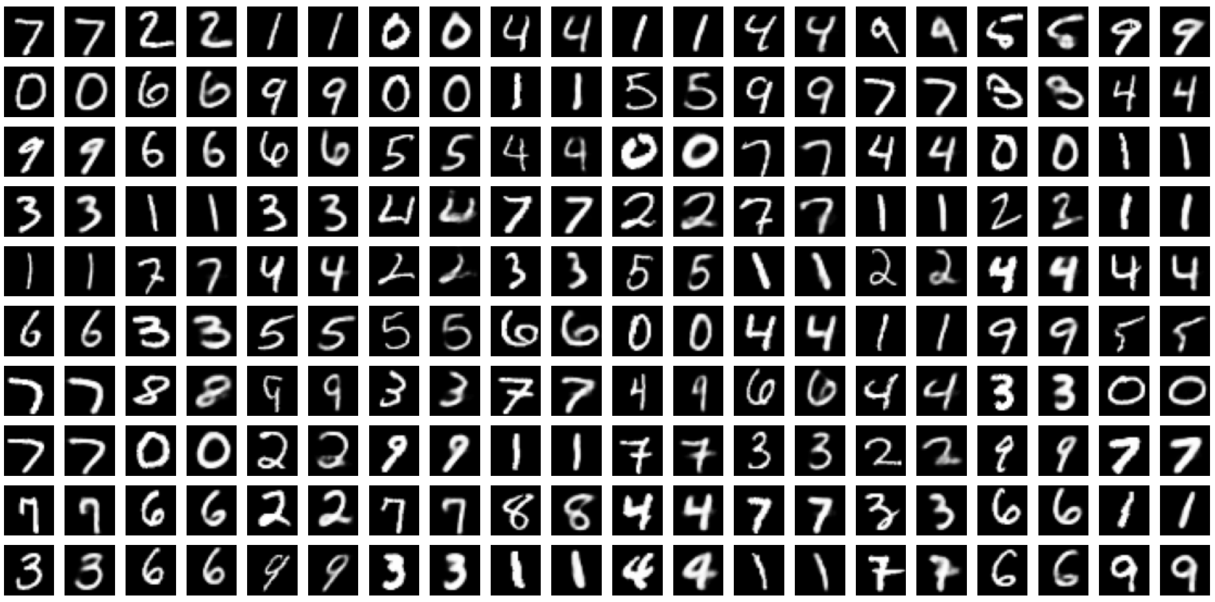
Figure 1: 100 Pairs of original and reconstructed samples.

## 3 Training and Evaluation

To train the auto-encoder network, we will use the $L_2$ distance between the output and the input of the network as a loss function, which is implemented in `torch.nn.MSELoss`:

$$\mathcal{J}^{L_2}(\mathbf{X},\mathbf{Y}) = \|\mathbf{X}-\mathbf{Y}\|^2 \tag{1}$$

For optimization, we will make use of the Adam optimizer with a learning rate of $\eta = 0.001$. We will run the training for 10 epochs and compute training and validation set loss after each epoch.

For evaluation, we will first check whether some of the validation set samples are correctly reconstructed. Then, we see if the deep learned feature representations of the encoder are anything useful. For each of the 10 classes, we compute the deep feature representation $\vec{\mu}_o$ of that class by averaging over the encoding of the validation set samples of that class:

$$\vec{\mu}_o = \frac{1}{N_t}\sum_{n=1}^{N} \mathbb{I}\big(t^{[n]} = o\big)\mathcal{E}\big(\mathbf{X}^{[n]}\big) \tag{2}$$

where $N_t$ is the number of validation samples of class $t$.

Finally, we reconstruct images of these averages by decoding them into images. Additionally, we reconstruct images from of deep features of pairs of classes.

**Task 5: Training and Validation Loop** Instantiate the loss function and the optimizer. Train the auto-encoder network for 10 epochs. Compute the running average of the training loss for each epoch. At the end of each epoch, also compute the validation set loss.

**Task 6: Reconstruction Result** Take the first batch of the validation set and provide it to the auto-encoder network to extract their reconstructions. Create a plot that shows pairs of original and reconstructed samples in 10 rows and 10 columns, as can be seen in Figure 1.

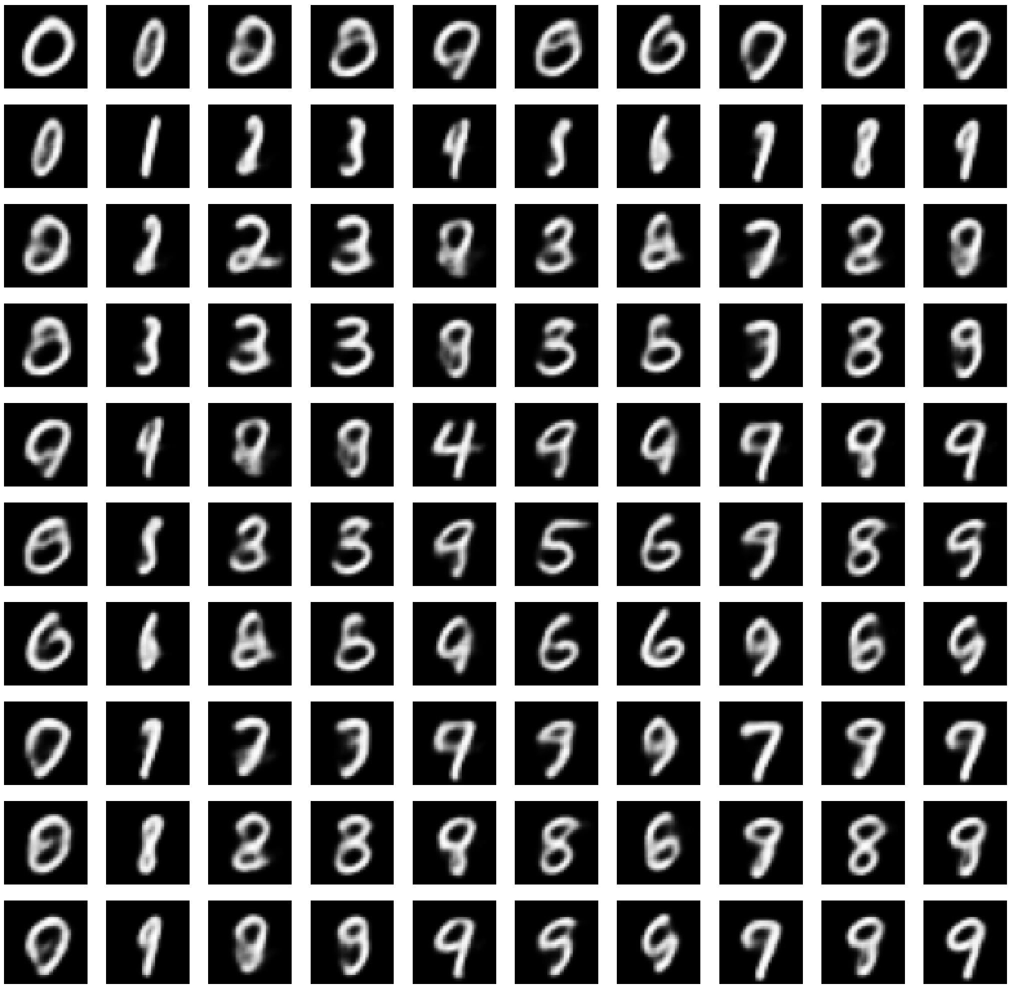

Figure 2: Reconstructed samples of mixtures of two classes each.

**Task 7: Mean Vector per Class** Use the encoder part of the trained auto-encoder network to extract deep feature representations for each validation set sample. For each of the 10 classes, select all samples of that class and compute the average of the deep features for that class according to (2).

**Task 8: Decode Mixtures of Classes** For each pair $o_1, o_2$ of classes, decode the mean of the two deep feature representations:

$$\mathbf{Y}_{o_1,o_2} = \mathcal{D}\left(\frac{\vec{\mu}_{o_1} + \vec{\mu}_{o_2}}{2}\right) \tag{3}$$

Plot the results in a matrix representation, where the diagonal presents the results of the average of one class, while non-diagonal elements shows reconstructed mixtures of two classes. An according plot can be found in Figure 2.