ML3 - Practical Class n°4

In these practicals, we will study a special type of architecture for MLPs and CNNs called **auto-encoders** (AEs). Auto-encoders have two main use cases :

- **Representation learning** : they learn to map inputs to a new (usually lower dimensional) representation,
- **Unsupervised pre-training** : they can pre-train the first layer of a deep architecture using unlabeled data. This is very interesting when one has semi-supervised data, i.e. for a fraction of the dataset, the labels are known but they are unknown for the rest of the inputs. All inputs can be used to pre-train the network. The labeled fraction can be later used to learn the remaining layers.

The principle of AEs are briefly presented below.

AEs are meant to reconstruct inputs $\mathbf{x}^{(i)}$ so the output of such a network is a vector $\tilde{\mathbf{x}}^{(i)} \approx \mathbf{x}^{(i)}$. This is why no supervision is required to train them.

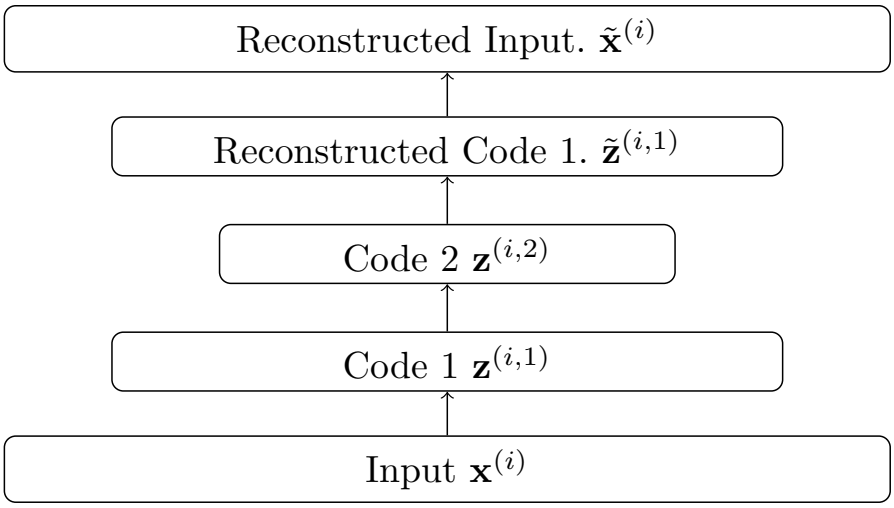
To prevent AEs from learning trivial functions (such as the identity function), it is forced to use a low dimensional intermediate representation, meaning that (in our usual notations), one of the $\mathbf{z}$ vector is much smaller than the input $\mathbf{x}$ from which it has been obtained as part of a forward pass. This representation is called a code, hence the name of the architecture.

To achieve this goal, the objective function for AEs is the mean square error (**MSE**) :

$$\frac{1}{n_{\text{train}}} \sum_{i=1}^{n_{\text{train}}} \left\| \tilde{\mathbf{x}}^{(i)} - \mathbf{x}^{(i)} \right\|_2^2. \tag{1}$$

Observe that this function is easy to differentiate so it is perfectly ok with backprop and we can train AEs by usual gradient descent techniques.

A simple way to progressively reach a low dimensional representation and reconstruct the input is to use a symmetric "butterfly" architecture. Besides, the weights from symmetrical layers can be forced to be the same (up to a transpose operation).



In the above figure, rectangles are layer input/outputs and arrows are layers. These layers can be convolutional or fully connected depending on the programmer choices. The low dimensional intermediate representation is the output of the second layer, i.e. $\mathbf{z}^{(i,2)}$ in our usual notations. The two first layers are called the **encoder** and the two last ones are called the **decoder**.

In these practicals, we will use fully connected layers. In this case, mapping a layer input to its output can be achieved using matrix calculus. For instance, to map $\mathbf{x}^{(i)}$ to $\mathbf{z}^{(i,1)}$, we use the weights $\mathbf{W}^{(1)}$ and the *bias/intercept* $\mathbf{b}^{(1)}$ as follows :

$$\mathbf{z}^{(i,1)} = f_{\text{act}}\left(\mathbf{W}^{(1)} \cdot \mathbf{x}^{(i)} + \mathbf{b}^{(1)}\right). \tag{2}$$

The number of columns of matrix $\mathbf{W}^{(1)}$ is equal to the size of the vector $\mathbf{x}^{(i)}$. The number of lines of matrix $\mathbf{W}^{(1)}$ as well as the size of vector $\mathbf{b}^{(1)}$ are equal to the number of neural units in layer $k = 1$. f_{act} is some activation function that is applied entry-wise to vector $\mathbf{W}^{(1)} \cdot \mathbf{x}^{(i)} + \mathbf{b}^{(1)}$.

As usual, our goal is to learn those weight matrices and the intercept vectors for each of the 4 layers in order to minimize the MSE loss. As mentioned above, we can make the learning task easier by tying matrices w.r.t. the architecture symmetry :

$$\mathbf{W}^{(4)} = \mathbf{W}^{(1)^T}, \tag{3}$$

$$\mathbf{W}^{(3)} = \mathbf{W}^{(2)^T}. \tag{4}$$

Generally speaking, this is not mandatory but experience shows that the model remains flexible while limiting the risk of overfitting. We are now ready to start using AEs with `tensorflow`.

Exercise 1 : Comparison with PCA

So AEs are able to map input to a lower dimensional representation, it is in this regard an unsupervised dimensionality reduction technique, just as **principal component analysis** (PCA). Actually, when an AE has a one-layer encoder, its model is pretty close to that of PCA which also minimizes MSE.

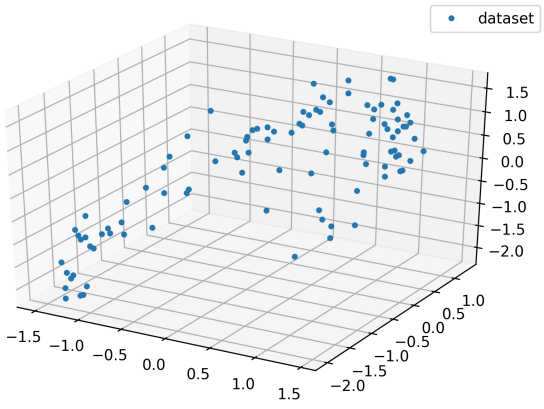
In this exercise, we will illustrate the ability of AEs to perform dimensionality reduction on a toy dataset. Unlike the previous practicals, we will not instantiate directly a layer. Instead we will use instances of `tf.Variable` for the weight matrices $\mathbf{W}^{(j)}$ and intercept vectors $\mathbf{b}^{(j)}$. The goal is twofold :

— It shows you that `tensorflow` is quite flexible and that customized layers are not difficult to create,

— We will be easily able to tie matrices as desired, see (3) and (4).

From these variables, we can complete the computation graph by using `tensorflow` calculus functions.

Let us create a first rudimentary AE and train it to learn a 2D representation from the following 3D dataset :



To complete the exercise, you must fill the gaps in the following python file : `autoenc_etu.py`. In this file, you will see :

— a first cell that creates a synthetic dataset where inputs are in 3D,

— a second cell where a new python class `basic_AE` is defined,

— a third cell where an instance of this class is created,

— a fourth cell, where the model instance is trained,

— and a final and fifth cell where we visualize results.

Remark : In these practicals, we will use the exponential linear unit (ELU) activation function which approximates ReLU.

Questions :

1. The `basic_AE` class is a child class of `tf.Module` which defines a framework for designing a customized tensorflow model. The constructor of the class is already partially written. This constructor is meant to create instances of `tf.Variable` with appropriate sizes.

   Assuming that we build an AE whose **decoder and encoder have only one layer**, fill the gaps in the constructor definition. The missing numbers are integers that are elements of the list `unit_nbrs` which contains the input dimension followed by the desired number of units per layer. In this exercise, we thus have `len(unit_nbrs)`=2.

2. There are also missing instructions in the `basic_AE.__call__` function. This function will allow an instance of the class to be also a callable object, because neural networks are functions mapping inputs to predictions. In this case the AE is meant to map $\mathbf{x}$ to $\tilde{\mathbf{x}}$. Fill the gaps in `basic_AE.__call__` by using tensorflow ops such as `tf.matmul` and `tf.transpose`.

3. Proceed to the next cell. If the constructor is correctly written, instantiation should work. You should also run unit tests on a small tensor to check if `basic_AE.__call__` works well.

4. Proceed to the next cell. In this cell, we loop on epochs and we do batch gradient descent for simplicity. The only missing instructions of the training loop are in the `with` block. In this block, you must compute the loss obtained by the AE the whole training set.

   If your code is fine, the loss should decrease at each epoch.

5. Proceed to the next cell and plot these 2D representations and check that the main geometry of the dataset is preserved.

6. Plot the evolution of the MSE loss w.r.t. time iterations.

7. Apply PCA to the same dataset and compare the results. (You may use the `sklearn` version of PCA but you will need to install the corresponding python module.)

Exercise 2 : AE for representation learning with `MNIST`

We now move to a real data problem and we will again use MNIST. In this exercise, we will create a 4 layer AE (just like in the figure from page 1). We will need to make the code from exercise 1 more modular so that we can create multi-layer encoders/decoders. Let us remind that images in MNIST have shape 28×28. There are 55000 images for training and 10000 for test error computation.

We now use the `autoenc2_etu.py` file. There are new elements in this code. In particular, an L_2 regularizer will be added to the loss. This additional loss term is embodied by function `reg`. The regularization applies to matrices $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$ only.

1. By taking inspiration from the previous exercise, fill the gaps in the constructor of the `AE` class. Observe that you must loop on layers for the encoder and loop a second time for the decoder.
2. Fill the gaps in the `AE.__call__` function. Just like before this function maps $\mathbf{x}$ to $\tilde{\mathbf{x}}$. Again, we have one for the encoder followed by another loop for the decoder.
3. You can instantiate an auto-encoder in the next cell and check of the maths work.
4. Fill the gaps in the `reg` function. In this function, we need to loop on the trainable groups of parameters of the model. In tensorflow, remember that trainable parameters are grouped by layers. We filter them by checking the first letter of their name because intercepts are not to be included in the regularization.
5. Proceed to the next cell where the training loop skeleton is available. Similarly as in the previous exercise, the missing instructions are in the `with` block. In this block, you must compute the training loss. Do not forget to add the regularization term. Also, this time, we want to do mini-batch gradient descent. Adapt the instruction to process only a subset of `x_train` in each step of one epoch. We mini-batch size is set to 10 in the code.

6. Proceed to the next cell and obtain reconstructed inputs from the *test set* and display a pair of images before/after reconstruction. The result should look like this :