EE-556: Mathematics of data: from theory to computation
Homework Exercise 2

Natalie Bolón Brun

11/2018

1 Autoencoder for Representational Learning

The loss function of the autoencoder given N datapoints is the mean squared error between the input and output data:

$$L = \frac{1}{N}\sum_{i=1}^{N} L^{(i)} = \frac{1}{2N}\sum_{i=1}^{N}\left\|\boldsymbol{x^{(i)}} - \varphi(\phi(\boldsymbol{x^{(i)}}))\right\|_2^2 \tag{1}$$

In our case both encoder and decoder are given by a one-layer network without bias term and are parameterized by $\boldsymbol{W^1}$ and $\boldsymbol{W^2}$ respectively:

$$\boldsymbol{z^{(i)}} = \phi(\boldsymbol{x^{(i)}}) = \sigma(\boldsymbol{W^1}\boldsymbol{x^{(i)}}) \tag{2}$$

$$\boldsymbol{x'^{(i)}} = \varphi(\boldsymbol{z^{(i)}}) = \boldsymbol{W^2}\boldsymbol{z^{(i)}}) \tag{3}$$

where σ is an activation function, $\boldsymbol{x^{(i)}} \in \mathrm{R}^d$, $\boldsymbol{z^{(i)}} \in \mathrm{R}^k$, $\boldsymbol{W^1} \in \mathrm{R}^{kxd}$ and $\boldsymbol{W^2} \in \mathrm{R}^{dxk}$.

a) Write down the loss function $L^{(i)}$ in terms of $\boldsymbol{x^{(i)}}$ and parameters $\boldsymbol{W^1}$, $\boldsymbol{W^2}$. Calculate the gradient of $L^{(i)}$ with respect to both parameters.

The loss for a datapoint is given by

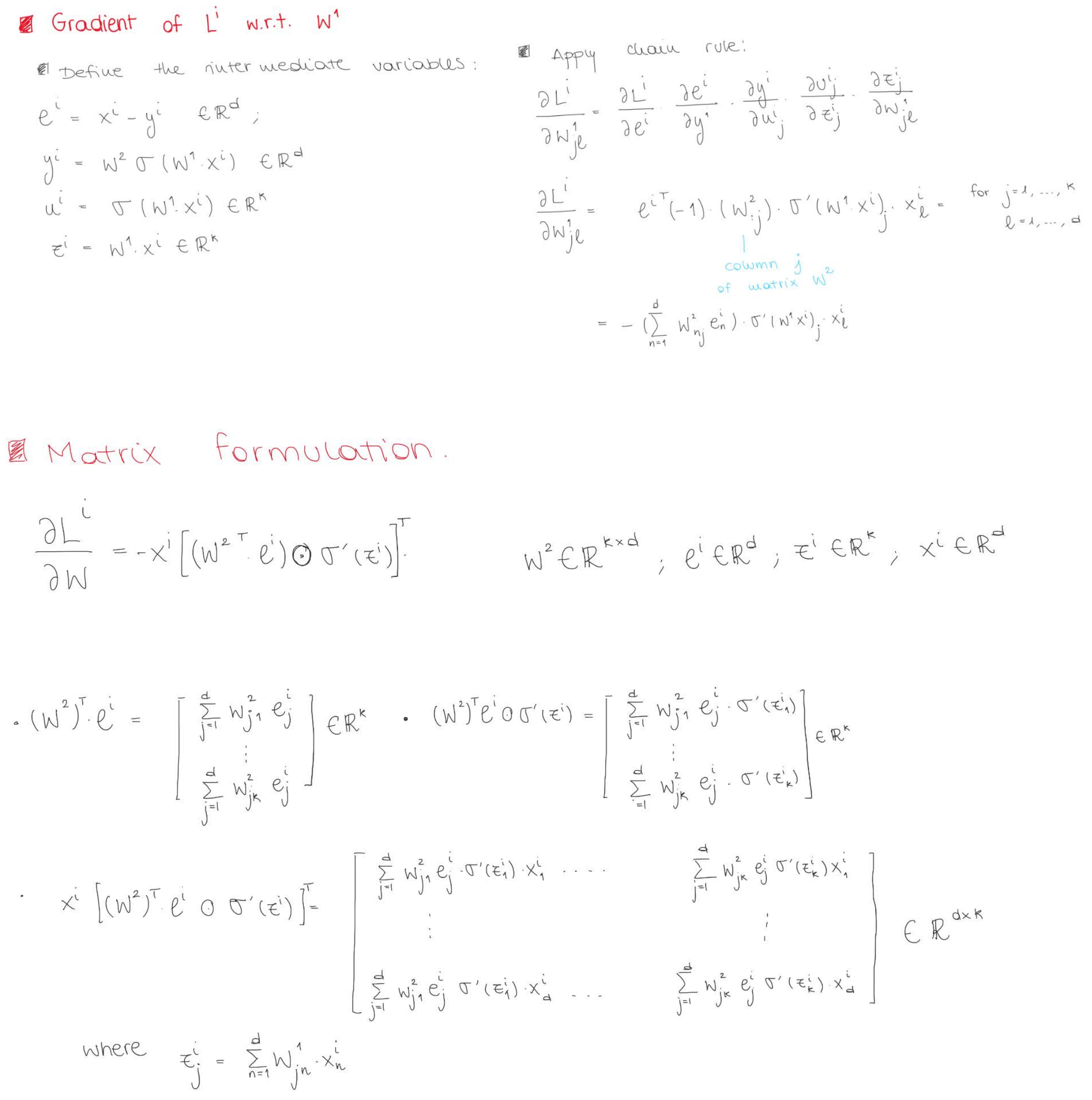
$$L^{(i)} = \left\|\boldsymbol{x^{(i)}} - \varphi(\phi(\boldsymbol{x^{(i)}}))\right\|_2^2 = < \boldsymbol{x^{(i)}} - W^2\sigma(W^1\boldsymbol{x^{(i)}}), \boldsymbol{x^{(i)}} - W^2\sigma(W^1\boldsymbol{x^{(i)}}) > \tag{4}$$

$$L^{(i)} = \sum_{j=1}^{d}\ [x_j^i - \sum_{l=1}^{k} W_{jl}^2\ \ \sigma(\sum_{m=1}^{d} W_{lm}^1 x_m^i)]^2 \tag{5}$$

where σ represents the activation function, $\boldsymbol{b}$ is the dimension of the input and $\boldsymbol{k}$ is the dimension of the hidden layer.

The gradient with respect the different weights is given by:

- Gradient w.r.t. $\boldsymbol{W^1}$:



Gradient of L^i w.r.t. W^1

Define the intermediate variables:

$$e^i = x^i - y^i \in \mathbb{R}^d ;$$

$$y^i = W^2 \sigma(W^1 \cdot x^i) \in \mathbb{R}^d$$

$$u^i = \sigma(W^1 \cdot x^i) \in \mathbb{R}^k$$

$$z^i = W^1 \cdot x^i \in \mathbb{R}^k$$

Apply chain rule:

$$\frac{\partial L^i}{\partial W^1_{jl}} = \frac{\partial L^i}{\partial e^i} \cdot \frac{\partial e^i}{\partial y^i} \cdot \frac{\partial y^i}{\partial u^i_j} \cdot \frac{\partial u^i_j}{\partial z^i_j} \cdot \frac{\partial z^i_j}{\partial W^1_{jl}}$$

$$\frac{\partial L^i}{\partial W^1_{jl}} = e^{i\,T}(-1) \cdot \underbrace{(W^2_{:j})}_{\text{column } j \text{ of matrix } W^2} \cdot \sigma'(W^1 x^i)_j \cdot x^i_l = \quad \text{for } j = 1, \dots, k \quad l = 1, \dots, d$$

$$= -\left(\sum_{n=1}^{d} W^2_{nj} e^i_n\right) \cdot \sigma'(W^1 x^i)_j \cdot x^i_l$$

Matrix formulation.

$$\frac{\partial L^i}{\partial W} = -x^i \left[(W^{2\,T} e^i) \odot \sigma'(z^i) \right]^T \qquad W^2 \in \mathbb{R}^{k \times d} ; \; e^i \in \mathbb{R}^d ; \; z^i \in \mathbb{R}^k ; \; x^i \in \mathbb{R}^d$$

- $$(W^2)^T \cdot e^i = \begin{bmatrix} \sum_{j=1}^{d} W^2_{j1} e^i_j \\ \vdots \\ \sum_{j=1}^{d} W^2_{jk} e^i_j \end{bmatrix} \in \mathbb{R}^k$$

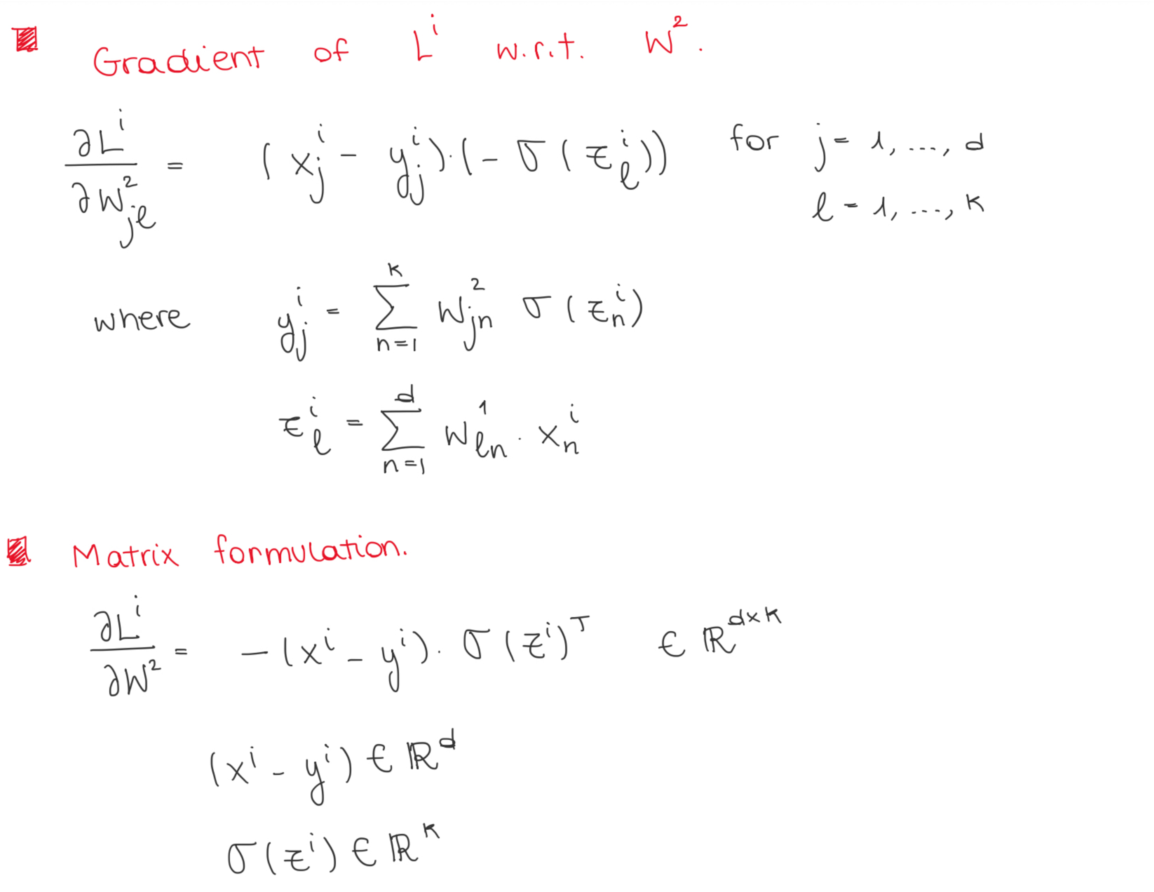
- $$(W^2)^T e^i \odot \sigma'(z^i) = \begin{bmatrix} \sum_{j=1}^{d} W^2_{j1} e^i_j \cdot \sigma'(z^i_1) \\ \vdots \\ \sum_{j=1}^{d} W^2_{jk} e^i_j \cdot \sigma'(z^i_k) \end{bmatrix} \in \mathbb{R}^k$$

- $$x^i \left[(W^2)^T e^i \odot \sigma'(z^i) \right]^T = \begin{bmatrix} \sum_{j=1}^{d} W^2_{j1} e^i_j \cdot \sigma'(z^i_1) \cdot x^i_1 & \cdots & \sum_{j=1}^{d} W^2_{jk} e^i_j \sigma'(z^i_k) x^i_1 \\ \vdots & & \vdots \\ \sum_{j=1}^{d} W^2_{j1} e^i_j \sigma'(z^i_1) \cdot x^i_d & \cdots & \sum_{j=1}^{d} W^2_{jk} e^i_j \sigma'(z^i_k) \cdot x^i_d \end{bmatrix} \in \mathbb{R}^{d \times k}$$

where $z^i_j = \sum_{n=1}^{d} W^1_{jn} \cdot x^i_n$

Figure 1: Gradient derivation with respect to W_1

- Gradient w.r.t. $\boldsymbol{W^2}$:



Gradient of L^i w.r.t. W^2.

$$\frac{\partial L^i}{\partial W^2_{j\ell}} = (x_j^i - y_j^i)\cdot(-\sigma(z_\ell^i)) \quad \text{for } j = 1, \dots, d \quad \ell = 1, \dots, K$$

where

$$y_j^i = \sum_{n=1}^{K} W^2_{jn}\, \sigma(z_n^i)$$

$$z_\ell^i = \sum_{n=1}^{d} W^1_{\ell n} \cdot x_n^i$$

Matrix formulation.

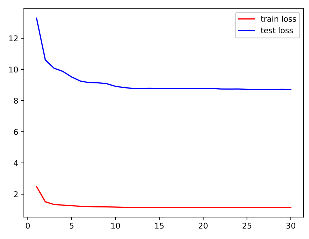
$$\frac{\partial L^i}{\partial W^2} = -(x^i - y^i)\cdot \sigma(z^i)^T \quad \in \mathbb{R}^{d \times K}$$

$$(x^i - y^i) \in \mathbb{R}^d$$

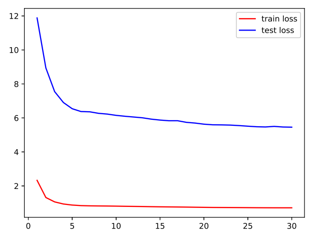
$$\sigma(z^i) \in \mathbb{R}^K$$

Figure 2: Gradient derivation with respect to W$_2$

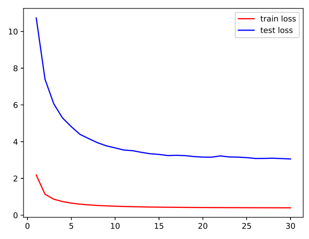
b) Training of the autoencoder on MNIST. Learning curve for the model generated using different sizes for the hidden layer $\boldsymbol{\{25, 50, 100\}}$, different activation functions $\boldsymbol{\{Relu, \sigma, tahn, Identity \;\; and \;\; Negative\}}$ as activation function and step-size of $\mathbf{0.01} \;\; \boldsymbol{or} \;\; \mathbf{0.005}$ depending on the activation function. Autoencoder trained with **30** epochs.



(a) 25 hidden units

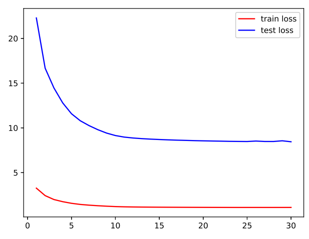


(b) 50 hidden units

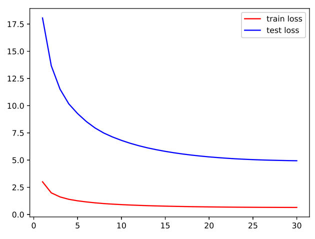


(c) 100 hidden units

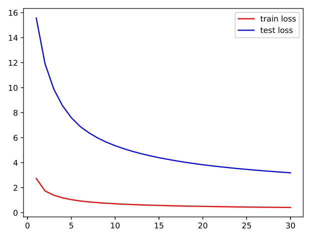
Figure 3: Loss curves for different sizes of hidden layer using Relu as activation function of the hidden layer and Identity in the output layer. Step size value set to 0.005 constant.



(a) 25 hidden units

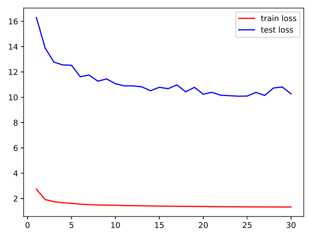


(b) 50 hidden units

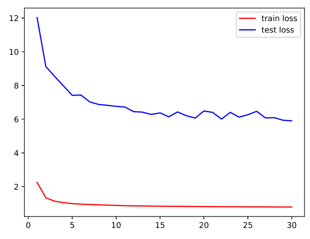


(c) 100 hidden units

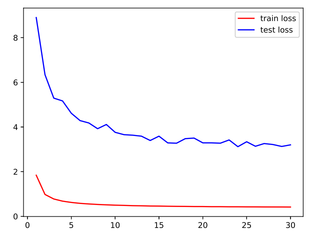
Figure 4: Loss curves for different sizes of hidden layer using σ as activation function of the hidden layer and Identity in the output layer. Step size value set to 0.005 constant.



(a) 25 hidden units

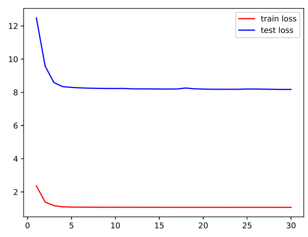


(b) 50 hidden units

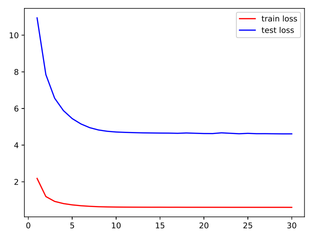


(c) 100 hidden units

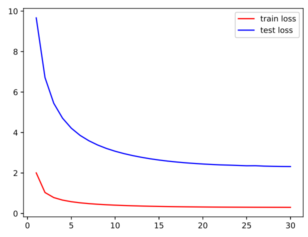
Figure 5: Loss curves for different sizes of hidden layer using $\boldsymbol{tanh}$ as activation function of the hidden layer and Identity in the output layer. Step size value set to 0.01 constant.



(a) 25 hidden units

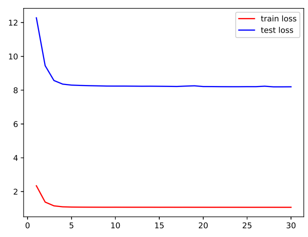


(b) 50 hidden units

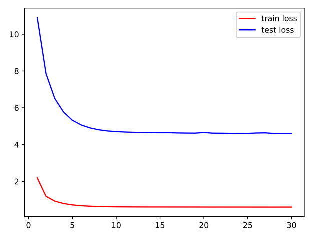


(c) 100 hidden units

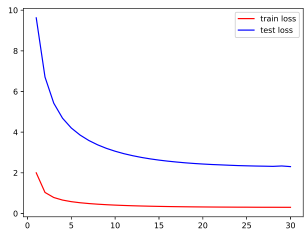
Figure 6: Loss curves for different sizes of hidden layer using Identity as activation function of the hidden layer and Identity in the output layer. Step size value set to 0.005 constant.



(a) 25 hidden units



(b) 50 hidden units



(c) 100 hidden units

Figure 7: Loss curves for different sizes of hidden layer using Negative as activation function of the hidden layer and Identity in the output layer. Step size value set to 0.005 constant.

The final loss values are:

hidden units	**25**		**50**		**100**	
	Train loss	**Test loss**	**Train loss**	**Test loss**	**Train loss**	**Test loss**
Relu	1.135	8.717	7.154e-01	5.457	4.023e-01	3.059
Sigmoid	1.109	8.450	6.475e-01	4.940	4.207e-01	3.194
Tanh	1.329	1.027e+01	7.932e-01	5.906	4.162e-01	3.200
Identity	1.069	8.172	6.043e-01	4.616	3.029e-01	2.320
Negative	1.070	8.201	6.053e-01	4.603	3.019e-01	2.304

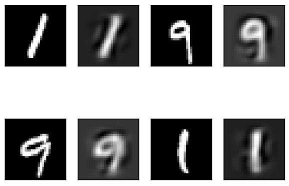
Table 1: Loss values after training with 30 epochs for different activation functions and sizes of hidden layer. As expected, loss is decreasing as the size of the hidden layer increases.

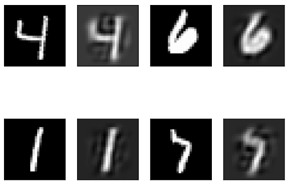
In all cases, the small hidden layers (25 units) the loss remains constant after the first 10 epochs or earlier. For higher number of hidden units, the two linear functions achieve a stationary loss earlier while the non-linear functions still keep decreasing the loss in the test set.

The comparison of the loss within the activation functions shows that the best results are achieved independently of the size of the hidden layer with the linear activation functions. It is also remarkable that the relative difference between the usage of linear and non-linear activation functions decreases as the size of the hidden layer decreases (higher compression rate).

c) Change the dimension of the hidden layer as well as the activation function and plot the corresponding reconstructed images.

The following images compare original and reconstructed images generated by different models. Along rows it is shown the results for different sizes of the hidden layer and the same activation function. Along columns, the size of the hidden layer is fixed while the activation function is varying within the five possible functions: Identity, Rectified Linear Unit, Hyperbolic Tangent, Sigmoid and Negative.


(a) 25 hidden units


(b) 50 hidden units

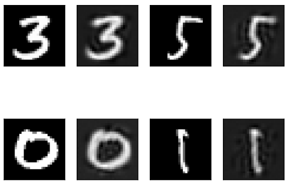
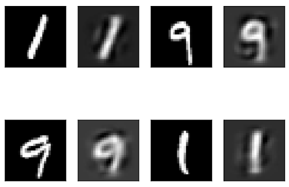
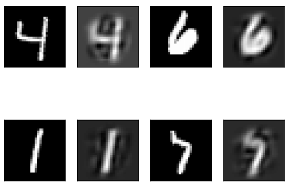

(c) 100 hidden units

Figure 8: Reconstruction of images generated with different sizes for the hidden layer; Identity as activation function; All models were trained with 0.005 as step size and for 30 epochs.


(a) 25 hidden units


(b) 50 hidden units

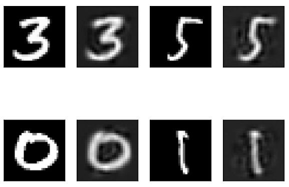
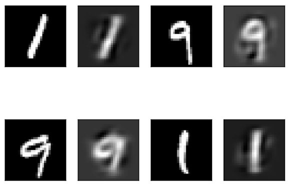
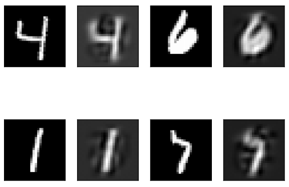

(c) 100 hidden units

Figure 9: Reconstruction of images generated with different sizes for the hidden layer; Relu as activation function; All models were trained with 0.005 as step size and for 30 epochs.


(a) 25 hidden units


(b) 50 hidden units

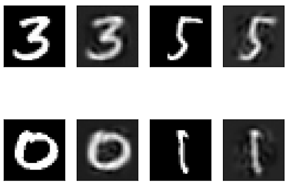
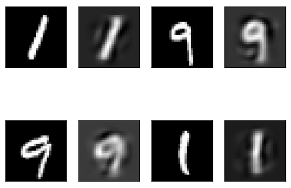
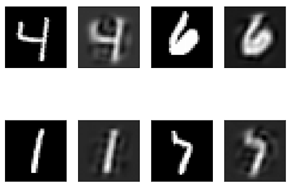

(c) 100 hidden units

Figure 10: Reconstruction of images generated with different sizes for the hidden layer; Hyperbolic tangent as activation function; All models were trained with 0.005 as step size and for 30 epochs.


(a) 25 hidden units


(b) 50 hidden units

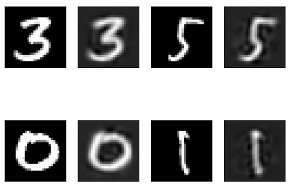
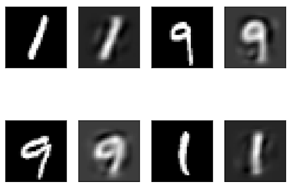
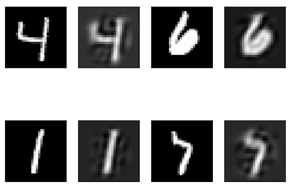

(c) 100 hidden units

Figure 11: Reconstruction of images generated with different sizes for the hidden layer; Sigmoid as activation function; All models were trained with 0.01 as step size and for 30 epochs.


(a) 25 hidden units


(b) 50 hidden units

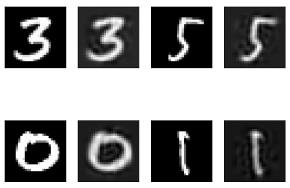
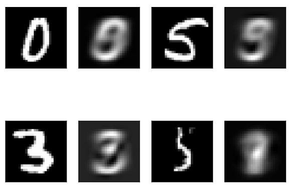

(c) 100 hidden units

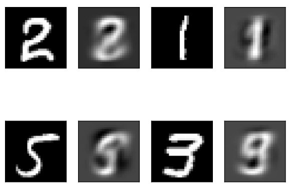
Figure 12: Reconstruction of images generated with different sizes for the hidden layer; "Negative" as activation function; All models were trained with 0.005 as step size and for 30 epochs.

As shown in the previous images, the effect of the activation function is not very remarkable. Only the case when the activation function is set to Hyperbolic Tangent seems to have poorer results than the rest of activations.

In terms of size of the hidden layer, as expected, the results improve as the size of the hidden layer increases for a fixed number of epochs. Nevertheless, in all cases the quality achieved by the reconstruction still allows the reader to recognize the original number.

If the size of the hidden layer decreases too much, the quality of the results decreases drastically generating a reconstruction that does not allow the interpret the original number without doubt. Under 15 hidden units the reconstruction cannot give enough information about the original image while for 15 hidden units it starts to become more loyal to the original.


(a) 5 hidden units


(b) 10 hidden units

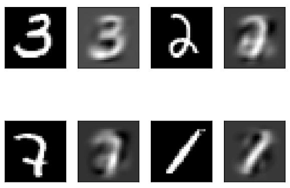

(c) 15 hidden units

Figure 13: Reconstruction of images generated with different sizes for the hidden layer; "Identity" as activation function; All models were trained with 0.005 as step size and for 20 epochs.

d) Show how can be chosen the activation function σ such that the resulting autoencoder resembles PCA. Are there any additional constrains on weights $\boldsymbol{W_1}$ and $\boldsymbol{W_1}$?

If the activation function σ is chosen to be unitary, the output of the encoder becomes:

$$\boldsymbol{x}'^{(i)} = \boldsymbol{W_2 W_1 x^{(i)}} \tag{6}$$

In this case, the problems turns into:

$$\inf_{W^2 W^1} \left\| \boldsymbol{x^{(i)}} - W^2 W^1 \boldsymbol{x^{(i)}} \right\|_2^2 = \sum_{j=1}^{d} \; [x_j^i - \sum_{l=1}^{k} W_{jl}^2 \; (\sum_{n=1}^{d} W_{ln}^1 x_n^i)]^2 \tag{7}$$

In this case, the matrix $\boldsymbol{W_2} * \boldsymbol{W_1} \in \mathbb{R}^{bxb}$ is a square matrix that results in the same output as the one given by PCA.
The projection in the space of reduced dimension is given by $\boldsymbol{W_1} * \boldsymbol{x^{(i)}}$.
The difference with respect to PCA is that both weight matrices produced by training the autoencoder to reduce the mean squared error can be non orthogonal. The matrix $P = W_2 * W_1$ can have non normalized and/or non orthogonal columns, characteristic that differs from the matrix $P * P^T$ produced by choosing the first k eigenvectors of the covariance matrix.
To obtain closer projections to PCA, the weights of the compression and reconstructions matrices ($\boldsymbol{W_1}$ and $\boldsymbol{W_2}$) should be such that the norm of the columns of $\boldsymbol{W_2}$ and the norm of the rows of $\boldsymbol{W_1}$ is equal to one.

2 Optimizers of Neural Network

a) Given different optimizers, run the neural network for 15 epochs 3 times per optimizer and compare the average accuracy obtained. Plot the obtained training loss. State how the performance of adaptive learning-rate methods vs SGD and SGD with momentum methods is compared. Use step sizes: 0.5, 10^{-2} and 10^{-5}.

Step Size	**SGD**	**SGD with Momentum**	**RMSProp**	**Adam**	**AdaGrad**
0.5	98.52%	9.93%	9.50%	10.95%	80.88%
10^{-2}	93.74%	97.78%	96.21%	97.93%	98.11%
10^{-5}	11.24%	31.36%	91.67%	87.33%	51.34%

Table 2: Average accuracy for optimizer and step size.

In Table 2 it be can observed the accuracy obtained for five different optimizers and three different values of step size.

In terms of accuracy, the highest accuracy is achieved by ***SGD*** with a step-size of 0.5 obtaining a final value of 98.52%.

The big step-size values, the adaptive methods and the incorporation of momentum does not lead towards better solutions but instead decrease produce results far from optimal.

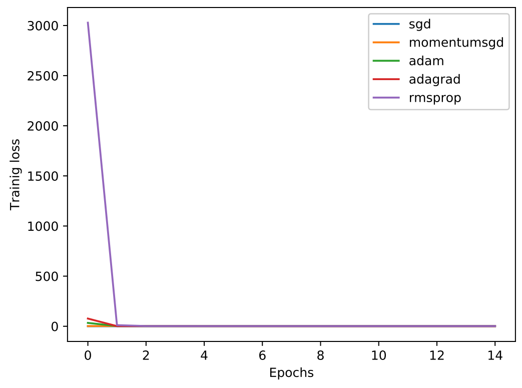
When decreasing the step-size to 1e-2, all algorithms achieve quite good accuracy values (higher than 90%). The performance of Adam and AdaGrad is quite similar obtaining an accuracy of around 98% in both cases. RMSProp as third adaptive method achieves also high accurate results but remains worse than SGD with Momentum. Finally, SGD is the one performing worst as it evolves much slower and will require a higher number of epochs to increase its accuracy and achieve values such as the ones obtained with the other optimizers.

Finally, for a very small initial step size, the performance of all optimizers decrease. However, RMSProp and Adam are still capable of achieving high values of accuracy. In this case, the incorporation of momentum does not allow an increase of the speed of convergence similar to the one obtained by the adaptive methods.

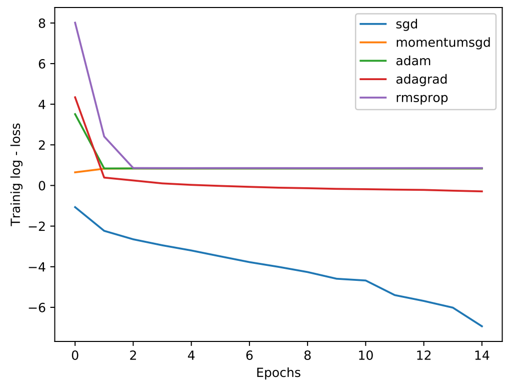
The comparison about the evolution of loss for the different optimizers is shown in Figure 16.

For a step size of 0.5, it can be seen that only SGD is capable of decreasing the loss after the first iteration. All other methods get stuck in a non-optimal solution. Apart from SGD, AdaGrad is the only algorithm able to achieve a reasonable accuracy in contrast with the three remaining algorithms that incorporate momentum and whose performance is under 15% of accuracy. This can be remarked in the loss plot (Figure 13) where the loss for AdaGrad is lower than SGD with Momentum, RMSProp and Adam. Nevertheless, it still remains far from the performance achieved by SGD.

In conclusion, with big step sizes, the incorporation of momentum does not accelerates convergence but leads to non-optimal solutions such as local minimums.

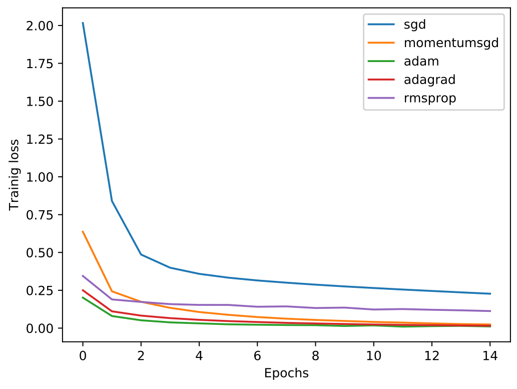


(a) Loss evolution for step size= 0.5

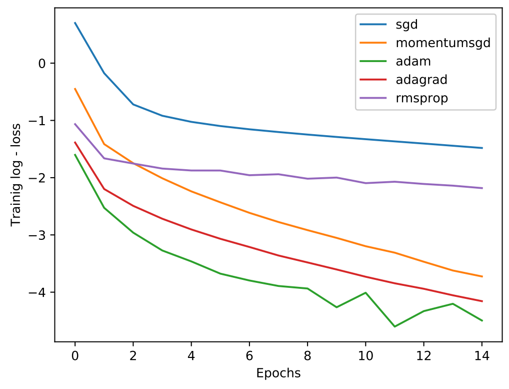


(b) Logarithmic loss evolution for step size =0.5

Figure 14: Loss evolution for step size of 0.5

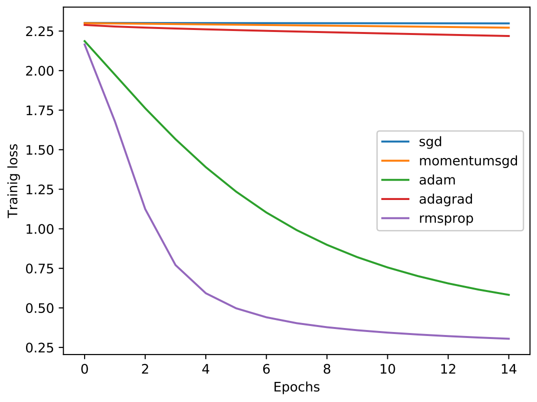


(a) Loss evolution for step size= 1e-2



(b) Logarithmic loss evolution for step size =1e-2

Figure 15: Loss evolution for a step size of 0.01



(a) Loss evolution for step size= 1e-5

Figure 16: Loss evolution for a step size of 1e-5

In Figure 14 a), it is shown the evolution of the loss using different optimizers for an initial step size of 0.01 while in Figure 14 b) it is plot its logarithm to ease the analysis of the different performances. It can be seen that all algorithms evolve properly and the best solution is achieved by Adam optimizer. In this case, SGD evolves much slower than the adaptive and momentum methods. Comparing momentum and adaptive methods, although SGD with momentum has an initially a higher loss, its decrease is similar to the one achieved by AdaGrad and Adam at the end of the training. In the case of RMSProp, its improvement is very slow and its loss remains almost constant after the first iteration through the whole dataset. This can be caused by a momentum term very big that causes the step size to decrease drastically after the first epoch. Finally, it can be stated that Adam presents oscillations when approaching an optimal solution.

For small values of initial step size, the best performance is achieved by RMSProp followed by Adam. SGD, SGD with Momentum and AdaGrad have almost constant loss value, it means its evolution is very slow and they remain far from an optimal solution after the whole training is completed. On the other hand, RMSProp reduces the loss by a factor of 4 in just 4 epochs, allowing to obtain a nearly optimal solution very fast in terms of required epochs. Nevertheless, after 8 epochs its decrease becomes much slower as it approaches a region with small gradient. Finally, Adam performs a less aggressive descent initially but remains quite constant along the epochs, slowing less than RMSProp.