# Mathematics of data: homework 2

Marco Pietro Abrate 292996

November 23, 2018

## 1 Autoencoder

The loss of the autoencoder is the mean squared error between the input and output data:

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i = \frac{1}{2N}\sum_{i=1}^{N} ||\mathbf{x}_i - \varphi(\phi(\mathbf{x}_i))||_2^2 \tag{1}$$

where $\mathbf{x}_i$ is one sample point, N is the dimension of the data set, $\varphi$ and $\phi$ are the encoder and the decoder, respectively. They are parameterized by $\mathbf{W}_1$ and $\mathbf{W}_2$ as follow:

$$\mathbf{z}_i = \phi(\mathbf{x}_i) = \sigma(\mathbf{W}_1\mathbf{x}_i), \tag{2}$$

$$\mathbf{x}'_i = \varphi(\mathbf{z}_i) = \mathbf{W}_2\mathbf{z}_i \tag{3}$$

where $\sigma$ is the elementwise activation function such as sigmoid and ReLU. We assume the dimensions $\mathbf{x}_i \in \mathbb{R}^d$ and $\mathbf{z}_i \in \mathbb{R}^k$ and $\mathbf{W}_1 \in \mathbb{R}^{kxd}$ and $\mathbf{W}_2 \in \mathbb{R}^{dxk}$.

It is now possible to rewrite the loss function:

$$L_i = \frac{1}{2}||\mathbf{x}_i - \mathbf{W}_2\sigma(\mathbf{W}_1\mathbf{x}_i)||_2^2 \tag{4}$$

### 1.1 Computing the gradient

The *i-th* gradient with respect to $\mathbf{W}_1$ and $\mathbf{W}_2$ of the loss function $L_i$ can be computed as:

$$\frac{\partial L_i}{\partial \mathbf{W}_2} = -(\mathbf{x}_i - \mathbf{W}_2\sigma(\mathbf{W}_1\mathbf{x}_i))(\sigma(\mathbf{W}_1\mathbf{x}_i))^T \tag{5}$$

$$\frac{\partial L_i}{\partial \mathbf{W}_1} = -\mathbf{W}_2^T(\mathbf{x}_i - \mathbf{W}_2\sigma(\mathbf{W}_1\mathbf{x}_i))\mathbf{1} \odot \sigma'(\mathbf{W}_1\mathbf{x}_i)\mathbf{x}_i^T \tag{6}$$

where $\mathbf{1} \in \mathbb{R}^{1xd}$ and $\sigma'$ is the first order derivative of the activation function.

## 1.2 Training the autoencoder

After implementing the training of the autoencoder on the MNIST dataset using minibatch stochastic gradient descent (dimension of the batch is of 100 data points), I tested different stepsizes using the ReLU activation function. The most suitable one appears to be 0.003, considering the training and test losses after 20 epochs. Finally, I created a model with the optimal stepsize on 50 epochs. The model was saved for the successive question. The results are shown in Figure 1.

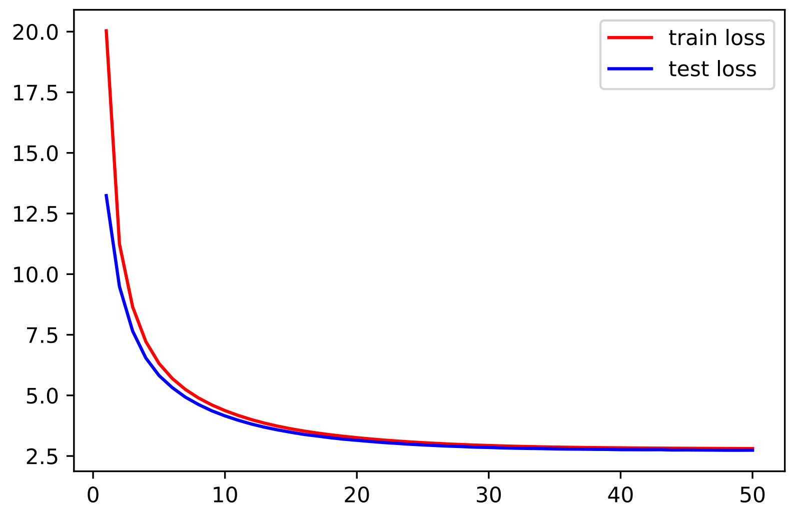


Figure 1: Training and test loss of the autoencoder.

## 1.3 Changing dimension of code and activation function

Using the formerly trained model, I run the algorithm for reconstructing the images using different dimensions of the hidden layer. It is clear from Figure 2 that there is not a big difference when using 100, 50 or 1 as the dimension of the hidden layer, using the ReLU activation function. This means that the samples present in the MNIST dataset can be projected on a well defined line without the loss of much information.
Then, I fixed the dimension of the code to 100 and I changed the activation function, training the models on 20 epochs and varying the stepsize from case to case. The results are shown in Figure 3, 4, 5 and 6.

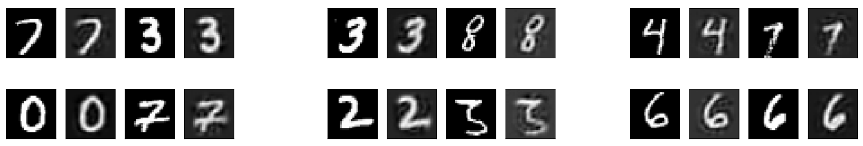


Figure 2: Reconstructed images using the ReLU activation function and 100, 50 and 1 as code dimension, from left to right.

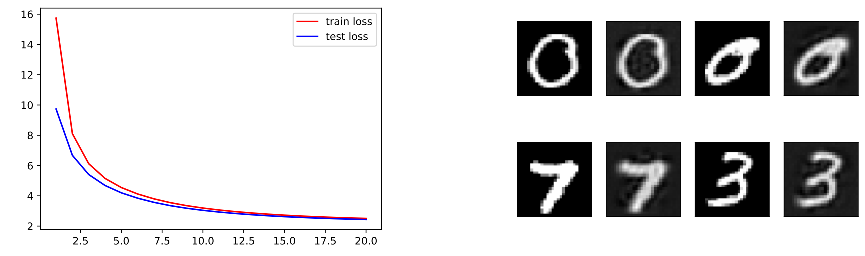


Figure 3: Losses (left) and reconstructed images (right) using a stepsize of 0.005 and the identity activation function. The code dimension is fixed to 100.

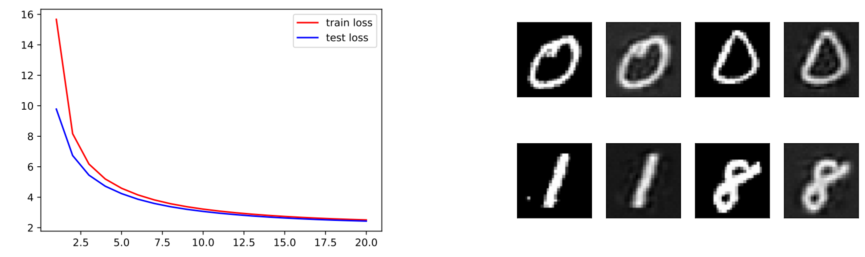


Figure 4: Losses (left) and reconstructed images (right) using a stepsize of 0.005 and the negative activation function. The code dimension is fixed to 100.

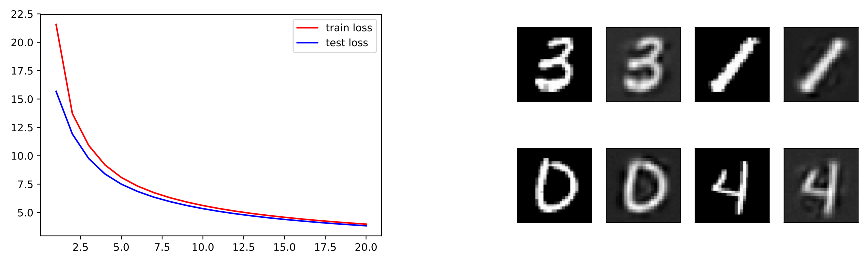
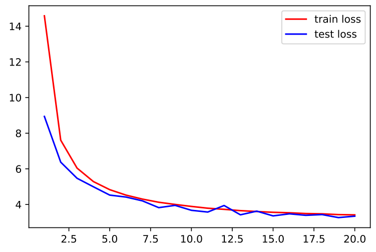


Figure 5: Losses (left) and reconstructed images (right) using a stepsize of 0.01 and the sigmoid activation function. The code dimension is fixed to 100.



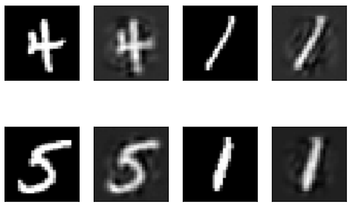


Figure 6: Losses (left) and reconstructed images (right) using a stepsize of 0.01 and the hyperbolic tangent activation function. The code dimension is fixed to 100.

### 1.4 Resembling Principal Component Analysis

Assuming that the input data $\{\mathbf{x}_i\}_{i=1}^N$ is normalized, the procedure of PCA is as follow:

1. Calculate covariance matrix $\Sigma = \frac{1}{N}\sum_{i=1}^N \mathbf{x}_i\mathbf{x}_i^T$;
2. Eigenvector decomposition $\Sigma = \mathbf{UDU}^T$ where $\mathbf{D}$ is a diagonal matrix and $\mathbf{U}$ is an orthogonal matrix;
3. Pick top $k$ eigenvectors from $\mathbf{U}$ to construct a projection matrix: $\mathbf{P} = (\mathbf{u}_1, ..., \mathbf{u}_k)$;
4. Calculate low-dimension projection: $\mathbf{z}_i = \mathbf{P}^T\mathbf{x}_i$.

Setting the activation function $\sigma$ to the identity and substituting $\mathbf{W}_1$ and $\mathbf{W}_2$ with $\mathbf{P}^T$ and $(\mathbf{P}^T)^{-1}$ respectively, the encoded and reconstructed signals are calculated as follow:

$$\mathbf{z}_i = \phi(\mathbf{x}_i) = \sigma(\mathbf{W}_1\mathbf{x}_i) = \mathbf{P}^T\mathbf{x}_i \tag{7}$$

$$\mathbf{x}_i' = \varphi(\mathbf{z}_i) = (\mathbf{P}^T)^{-1}\mathbf{z}_i \tag{8}$$

## 2 Optimizers of Neural Network

Different optimizers where implemented for a handwritten digit classifier using the MNIST dataset:

- Stochastic Gradient Descent;
- SGD with Momentum;
- Adagrad;
- RMSProp;
- Adam.

Then, I plotted the curves of the training losses and accuracies for each optimizer trained on 15 epochs, using 0.05, $10^{-2}$ and $10^{-5}$ as stepsize. The results are shown in Figure 7, 8 and 9, respectively.
Stochastic Gradient Descent works fine except for the smallest stepsize, because it might need more epochs to reach the minimum.
As expected, the SGD with momentum tents to overshoot with a stepsize of 0.05, which is too big for this method, and approaches the stationary point very slowly with $10^{-5}$.
AdaGrad adapts the learning rate in different directions, thus it works fine with every stepsizes, even if it learns slowly with the smallest one, but still faster than the SGD techniques.
RMSProp works badly with the highest stepsize, it might be because it gets stuck in a local minimum. This method works fine with the remaining stepsizes, even with the smallest one, due to the weight it gives to newly computed gradients.
Adam, mixing both the momentum and adaptive learning techniques, does not succeed to obtain a reasonable accuracy with the 0.05 stepsize, because it might overshoot. However, It is the most accurate with the smallest stepsize, because it starts slowly (avoiding overshoot), adapts the learning rate and efficiently escapes local minimum thanks to the momentum.
To conclude, I can say that adaptive learning rate methods are faster and more accurate than standard Stochastic Gradient Descent ones, though one must pay attention to the starting stepsize, as it can heavily influence the results.

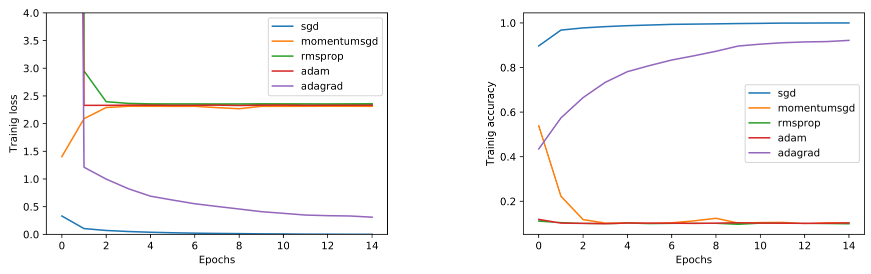


Figure 7: Training losses and accuracies setting the stepsize to 0.05.

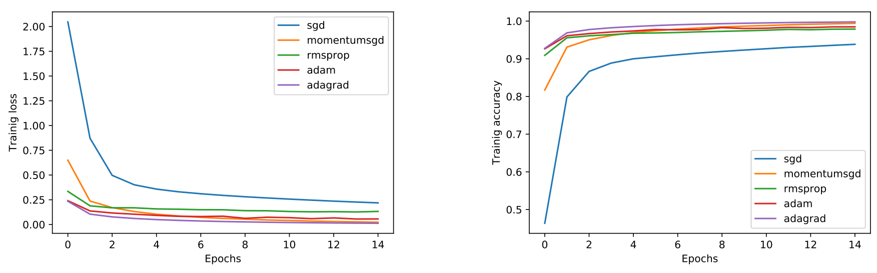
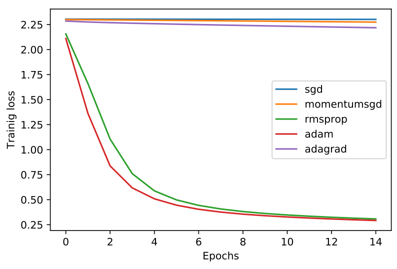


Figure 8: Training losses and accuracies setting the stepsize to $10^{-2}$.



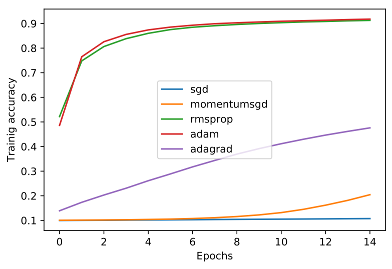


Figure 9: Training losses and accuracies setting the stepsize to $10^{-5}$.