Artificial Neural Networks and Deep Architectures, DD2437

Short report on lab assignment 4
Deep neural network architectures with autoencoders

Robert Cedergren, Viktor Törnegren and Lingxi Xiong

February 27, 2019

1 Main objectives and scope of the assignment

- To implement autoencoder by using deep learning libraries and explore the impact of the key parameters
- To design and train multi-layer autoencoders by using greedy layer-wise pretraining, and understand the ideas underlying the learning process
- To study the functionality through testing the performance of classification problems.

2 Method

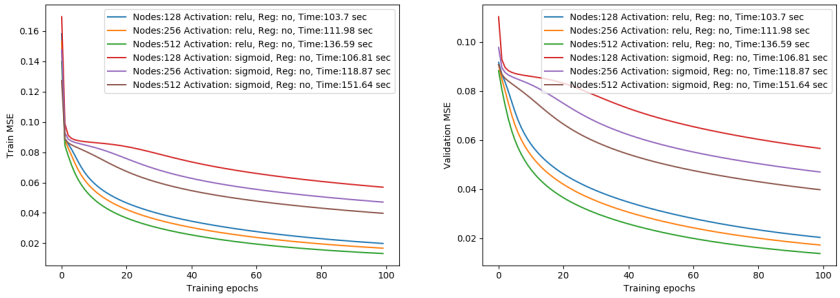
Python 3.6 and Keras 1 and 2 is used to conduct the tasks and experiments in this lab. Matplotlib is used for visualization.

3 Results and discussion

3.1 Autoencoder for binary-type MNIST images

For one layer autoencoder, the mean square error between the training input and the reconstructed input from the decoder was computed for different number of nodes (under-complete case) and with ReLu respectively the sigmoid as activation functions. From Figure 1 we can clearly see that the ReLu function converges faster compared to the sigmoid function, which is expected since ReLu is a is non-saturating function whereas the sigmoid is an saturating function. Moreover, from Figure 2 it can be seen that the autoencoder with ReLu respectively the sigmoid as activation function has no problem at all to reconstruct the images for the under complete case.

Further we also created autoencoders for the over-complete case, that is when we have more hidden nodes than input data, in our case we have 1024 hidden nodes. To cope with the problem of over-fitting we use the two standard regularization terms, L1-norm respectively L2-norm. In addition to this we also implement denoising autoencoders, this means that we add Gaussian noise to the input data in every epoch to prevent the model from simply learn to copy the input and instead force it to learn more robust features. From Figure 3 and 4 we can once again conclude that the ReLu function converges faster than the sigmoid function and from Table 1 it can be seen that a to low λ in our case $\lambda = 1e-5$ for both the L1 and L2 norm forces almost all (99%) of the nodes to be deactivated. Hence, to prevent overfitting one can either use L1 or L2 norm with $\lambda = 1e-7$ or apply Gaussian noise with $\sigma = 0.3$ this can further be confirmed from Figure 5.



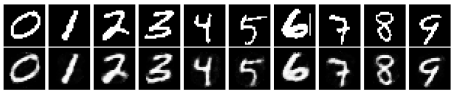
(a) Training error

(b) Validation error

Figure 1: Training and validation error for the under complete case.

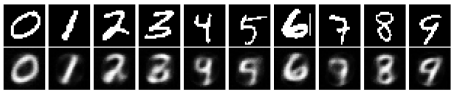
(a) Acivation = ReLu, nodes = 128

(b) Acivation = ReLu, nodes = 256

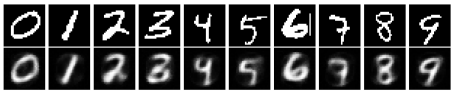


(c) Acivation = ReLu, nodes = 512

(d) Acivation = sigmoid, nodes = 128

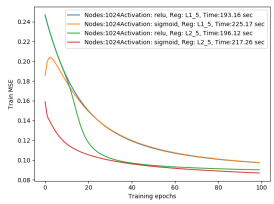


(e) Acivation = sigmoid, nodes = 256

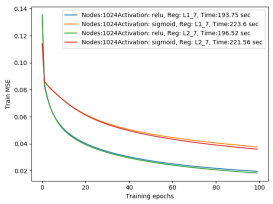


(f) Acivation = sigmoid, nodes = 512

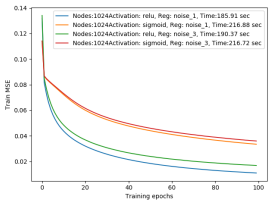
Figure 2: First row is original image, second row is reconstructed image, for under-complete case.



(a) Training error, with L1 and L2 norm, $\lambda = 1e-5$

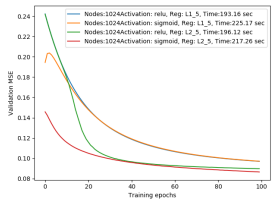


(b) Training error, with L1 and L2 norm, $\lambda = 1e-7$

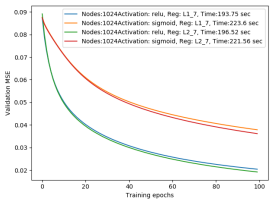


(c) Training error, with Gaussian noise

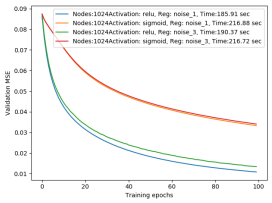
Figure 3: Training error for the over complete case with 1024 hidden nodes.



(a) Validation error, with L1 and L2 norm, $\lambda = 1e-5$



(b) Validation error, with L1 and L2 norm, $\lambda = 1e-7$

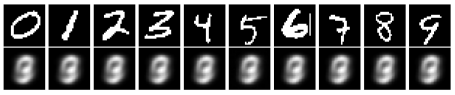


(c) Validation error, with Gaussian noise

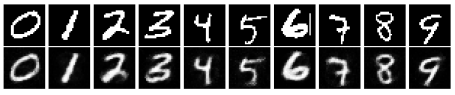
Figure 4: Validation error for the over complete case.

Activation	Regularisation	Parameter	Sparseness
ReLu	L1 norm	λ = 1e-5	0.99
Sigmoid	L1 norm	λ = 1e-5	0.99
ReLu	L1 norm	λ = 1e-7	0.75
Sigmoid	L1 norm	λ = 1e-7	0.25
ReLu	L2 norm	λ = 1e-5	0.99
Sigmoid	L2 norm	λ = 1e-5	0.94
ReLu	L2 norm	λ = 1e-7	0.57
Sigmoid	L2 norm	λ = 1e-7	0.12
ReLu	Gaussian noise	$\sigma = 0.1$	0.16
Sigmoid	Gaussian noise	$\sigma = 0.1$	0.02
ReLu	Gaussian noise	$\sigma = 0.3$	0.21
Sigmoid	Gaussian noise	$\sigma = 0.3$	0.02

Table 1: Average sparseness of hidden layer representations, over complete case, with 1024 hidden nodes.

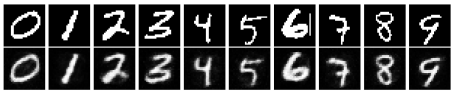


(a) Acivation = ReLu, L2-norm, $\lambda = 1e-5$

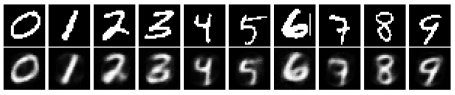


(b) Acivation = ReLu, L1-norm, $\lambda = 1e-7$

(c) Acivation = sigmoid, L1-norm, $\lambda = 1e-5$



(d) Acivation = ReLu, L2-norm, $\lambda = 1e-5$

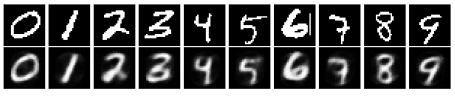


(e) Acivation = sigmoid, L2-norm, $\lambda = 1e-5$

(f) Acivation = ReLu, Gaussian noise, $\sigma = 0.1$

(g) Acivation = sigmoid, Gaussian noise, $\sigma = 0.1$

(h) Acivation = ReLu, Gaussian noise, $\sigma = 0.3$



(i) Acivation = sigmoid, Gaussian noise, $\sigma = 0.3$

Figure 5: First row is original image, second row is reconstructed image, for over-complete case

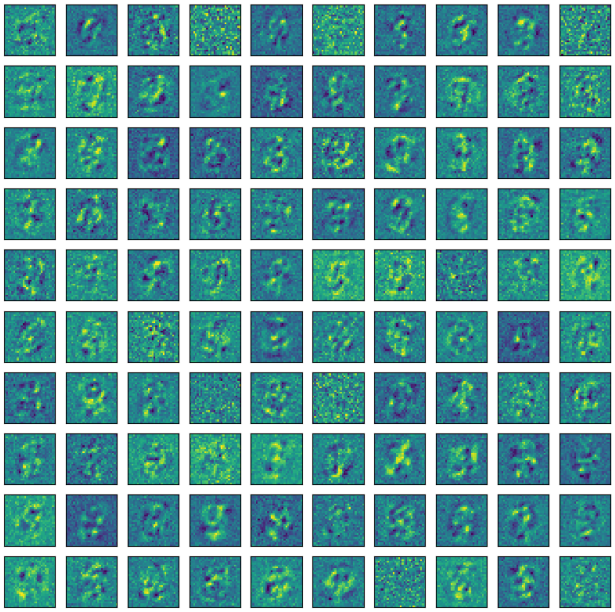


Figure 6: 784(28*28) bottom-up final weights for each hidden unit (100 nodes)

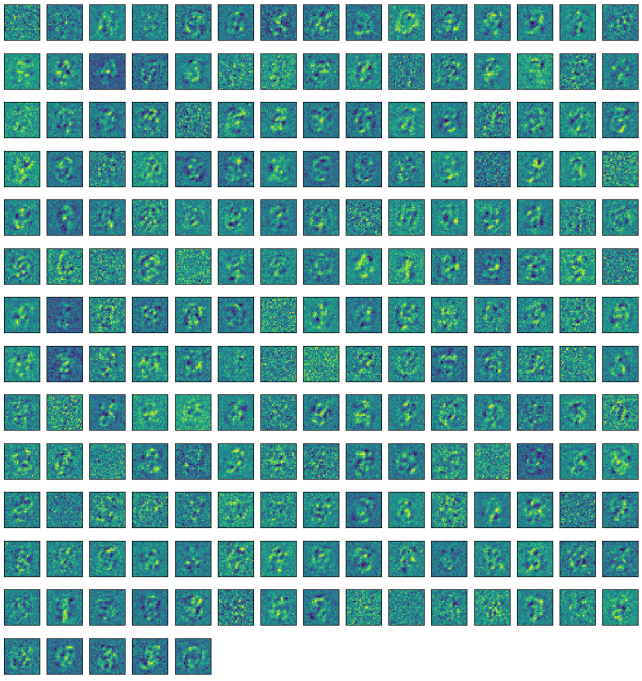


Figure 7: 784(28*28) bottom-up final weights for each hidden unit (100 nodes)

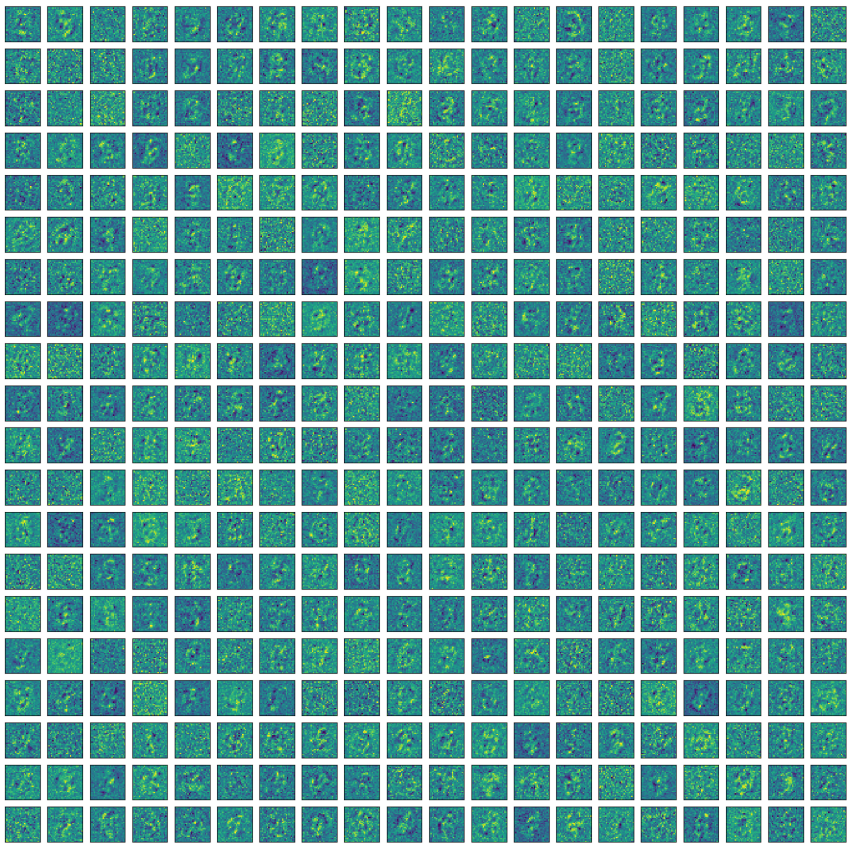


Figure 8: 784(28*28) bottom-up final weights for each hidden unit (100 nodes)

The weights for each hidden unit is shown in Figure 3, 4, and 5 with the configuration of one hidden layer with 100, 200, and 400 hidden nodes respectively. With undercomplete autoencoders, we are able to capture the most salient features of the training data, and along with this, the autoencoder learned the principal subspace of the training data having mean squared error as loss function.

3.2 Stacked autoencoders for MNIST digit classification (without noise)

We extended the single-layer autoencoder to a stacked autoencoder to perform a comparison of classification performance with different number of hidden layers (1, 2 and 3) as well as a no hidden nodes, i.e. a network that performed classification directly on the raw inputs. Since the undercomplete layers with 128, 256 and 512 seemed to perform good according to validation MSE the hidden layers constituted of 128, 256 or 512 units. The networks were initialized by perform-

ing greedy layer-wise pre-training. Learning rate and momentum used in both pre-training and for training the final network was 0.1 and 0.9, respectively. For the pre-training the weights were initialized randomly from a zero mean Gaussian with 0.05 as standard deviation and the biases were initialized with zeros. When using the ReLu activation function for training the full networks batch normalization was used to normalize the input of the hidden layers by adjusting and scaling the activations. The classification performances for the different specifications are presented in Table 2 below. 80 % of the training data (6400 observations) are used for training and thus 20 % of the training data (1600 samples) are used as validation data for monitoring the training. To prevent the networks form overfitting early stopping was used. More specifically, for both pre-training and training, the training was stopped when the validation loss did not decrease more than 0.000001.

For lower learning rate the convergence took longer time and due to the early stopping criterion the training sometimes stopped when it actually could possibly have benefitted from continuing.

Regardless of the pre-training configurations the greedy layer-wise pre-trainings always converged according to the early stopping criterion after 4 or 5 epochs. When training the full networks all the trainings more or less converged between 5-15 epoch and there was no systematic between architectures or configurations. However, the more amount of nodes and layers the more computation time. Regarding the classification errors it can be seen in Table 2 below that the networks with the ReLu activation functions always performs best on the test data. I.e. networks that were trained with the ReLu activation function generalizes better than its sigmoid dittos. Furthermore, it does not seem that more nodes and layers necessarily means better classification performance. Taking classification performance, model complexity and computation time into account, one does not really have a reason to use more than two layers. Depending on the application one could settle with a network using one or two hidden layers. The networks with two hidden layers performs better than the networks with one hidden layers, however, just slightly.

It is also worth noting that a no hidden layer network, i.e. a network that performed classification directly on the raw inputs had an average classification performance rate of 0.895. A very simple model in comparison to the networks with hidden layers, yet a classifier with relatively good results.

Hidden nodes	Accuracy (ReLu)	Accuracy (sigmoid)
128	0.92	0.91
256	0.93	0.92
512	0.92	0.91
128-128	0.93	0.89
256-256	0.94	0.89
512-512	0.94	0.88
128-256	0.94	0.88
128-512	0.94	0.90
256-128	0.92	0.89
256-512	0.94	0.90
512-128	0.94	0.90
512-256	0.94	0.89
512-512-512	0.94	0.80
256-256-256	0.94	0.30
128-128-128	0.93	0.20
128-256-512	0.92	0.30
512-256-128	0.93	0.82
256-512-256	0.94	0.76
256-128-256	0.92	0.31
512-128-512	0.94	0.11
512-256-512	0.93	0.64
128-256-128	0.93	0.29
128-512-128	0.92	0.2

Table 2: Classification performance for 1, 2 and 3 hidden layer networks.

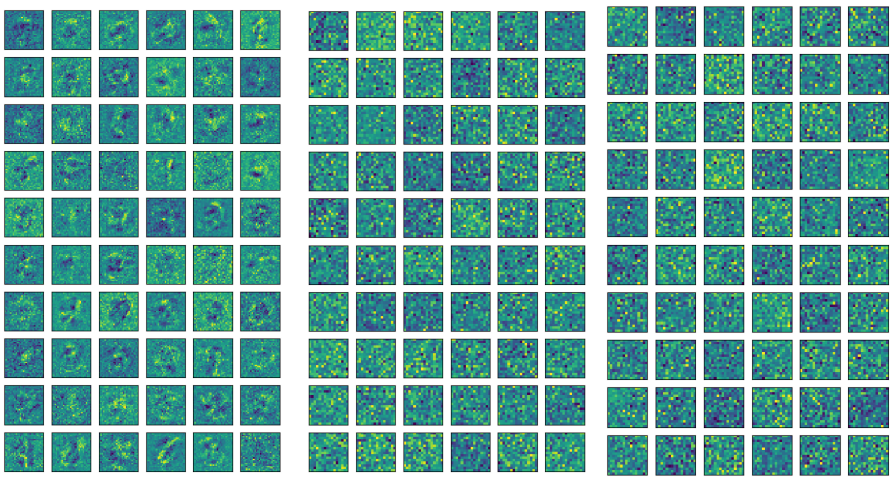


Figure 9: weight in first, second and third hidden layers

In Figure 7, if we visualize the weight of each hidden layer, we can see the

hierarchical structure of 'features' we extract is more and more abstract or 'deep'.