



Katholieke  
Universiteit  
Leuven

**Department of  
Computer Science**

# **ARTIFICIAL NEURAL NETWORKS**

## **FINAL REPORT**

Daniel Andrés Pérez Pérez - r0605947 - MCS

# Contents

<b>1 S1: Supervised learning and generalization</b>	<b>3</b>
1.1 Backpropagation in feedforward multi-layer networks . . . . .	3
1.2 Bayesian inference of networks hyperparameters . . . . .	3
<b>2 S2: Recurrent neural networks</b>	<b>5</b>
2.1 Hopfield networks . . . . .	5
2.2 Elman network . . . . .	6
<b>3 S3: Unsupervised learning: PCA and SOM</b>	<b>7</b>
3.1 Principal Component Analysis . . . . .	7
3.2 Competitive learning with SOM's . . . . .	7
<b>4 S4: Deep Learning: stacked autoencoders and convolutional Neural networks</b>	<b>9</b>
4.1 Stacked autoencoders . . . . .	9
4.2 Convolutional neural networks . . . . .	10
<b>5 Final project</b>	<b>11</b>
5.1 P1.1: Nonlinear regression . . . . .	11
5.2 P1.2: Classification . . . . .	11
5.3 P2: Character recognition with Hopfield networks . . . . .	13
<b>A Code: Nonlinear regression</b>	<b>15</b>
<b>B Code: Classification</b>	<b>18</b>
<b>C Code: Character recognition with Hopfield networks</b>	<b>21</b>

## List of Figures

1	Results of the FFNN prediction using different training algorithms in a noisy data. . . . .	3
2	Results of the FFNN prediction using <i>Bayesian regularization backpropagation</i> and <i>Levenberg-Marquardt</i> algorithms in a noisy data using 5 neurons in the hidden layer. . . . .	4
3	Results of the FFNN prediction using <i>Bayesian regularization backpropagation</i> and <i>Levenberg-Marquardt</i> algorithms in a noisy data using 30 neurons in the hidden layer . . . . .	4
4	Results of the <i>hopdigit</i> function with noise=4 and numiter=10. . . . .	5
5	Results of the <i>hopdigit</i> function with noise=4 and numiter=100. . . . .	5
6	Results of the <i>hopdigit</i> function with noise=100 and numiter=10. . . . .	5
7	Results of the <i>hopdigit</i> function with noise=100 and numiter=100. . . . .	5
8	Left: Image of the mean vector of the whole data set. Right: Image of the first element of the dataset with different reconstructions. . . . .	7
9	Plots of PCA exercise. . . . .	7
10	Top: Trajectory of the neurons using Manhattan distance. Bottom: Trajectory of the neurons using Link distance. . . . .	8
11	Top: Plots of SOM using grid topology and <i>linkdist</i> . Middle: Plots of SOM using hexagonal topology and <i>linkdist</i> . Bottom: Plots of SOM using hexagonal topology and <i>mandist</i> . . . . .	8
12	Surface of training set with 1000 elements . . . . .	11
13	Plots of FFNN performances with different number of neurons. . . . .	11
14	Plot of the ground truth and prediction surface. . . . .	12
15	Results of the classification tasks using different number of neurons . . . . .	12
16	Results of the classification tasks using different number of neurons . . . . .	12
17	Results of the classification tasks using different number of neurons . . . . .	12
18	Set of letter in the dataset (attractors) . . . . .	13
19	Top: original letter (attractor state). Middle: distorted letter by 3 pxeles. Bottom: reconstruction using the Hopfield network. . . . .	13
20	Average of absolute error and number of spurious patterns found with differen P values. . . . .	14
21	Theoretical loading capacity using the Hebb-rule. . . . .	14
22	Top: Original letter. Middle: distorted letter by 3 pxeles. Bottom: prediction of the FFNN. . . . .	14

## List of Tables

1	Results of time-series predictions using an Elman network with different parameters . . . . .	6
2	Results of DeepNet before fined-tuning. (# layer, max epochs, hidden units)	9
3	Results of DeepNet after fined-tuning. (# layer, max epochs, hidden units)	10
4	Results using a FFNN. . . . .	10

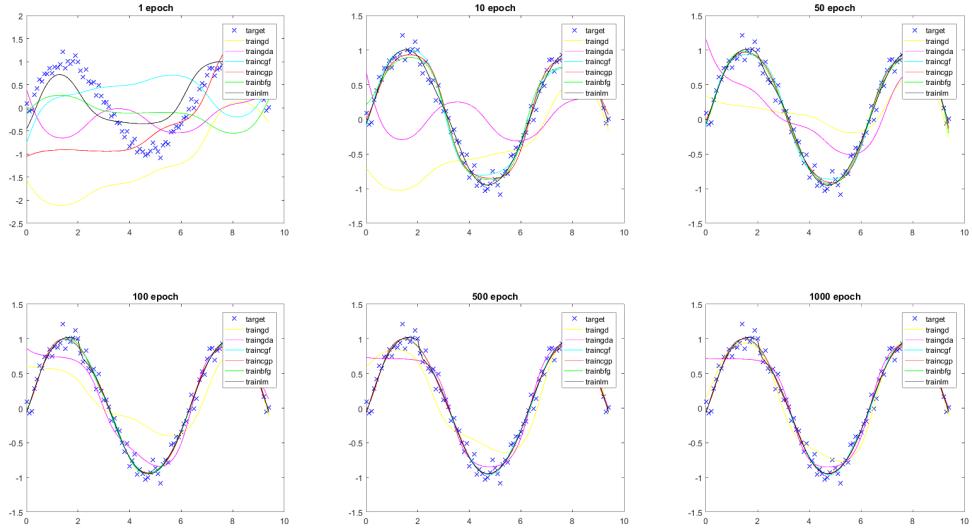
# 1 S1: Supervised learning and generalization

## 1.1 Backpropagation in feedforward multi-layer networks

All training algorithms were tested using a FFNN with 5 neurons in the hidden layer and 1 neuron in the output layer with different epochs. Two main experiments were executed on a free-noisy data and a noisy data. Due to space constraint, only results on noisy data are shown in 1.

The results clearly indicate that *Levenberg-Marquardt* algorithm (*trainlm*) converges quicker than the other learning algorithms. As we expected, *Gradient descent* algorithm (*traingd*) is the slowest to converge. One interesting point in figure 1 is that *trainlm* tends to overfit the curve, which illustrate that one should consider an early stop criteria before *trainlm* starts to overfit in order to increase the generalization of the overall model.

Figure 1: Results of the FFNN prediction using different training algorithms in a noisy data.



## 1.2 Bayesian inference of networks hyperparameters

*Bayesian regularization backpropagation* algorithm (*trainbr*) was tested on a noisy data, similar to the previous section, as well as *trainlm*. Different numbers of neurons were tested. Figures 2 and 3 show the extreme results of the experiments.

It is shown that, at the beginning of the learning phase, *trainbr* has a very bad model compared to *trainlm*. However after a relatively small number of epochs *trainbr* converges. According with the experiments, the speed rate to obtain a fairly good model from the *trainbr* is pretty similar to *trainlm*. The number of neurons in the hidden layer clearly affect both algorithms. The bigger the number of neurons has, the bigger the overfitting

is; which implies that the generalization of the model is small. To conclude, a good trade-off between number of neurons and epochs must be set properly in order to have a good model with high generalization.

Figure 2: Results of the FFNN prediction using *Bayesian regularization backpropagation* and *Levenberg-Marquardt* algorithms in a noisy data using 5 neurons in the hidden layer.

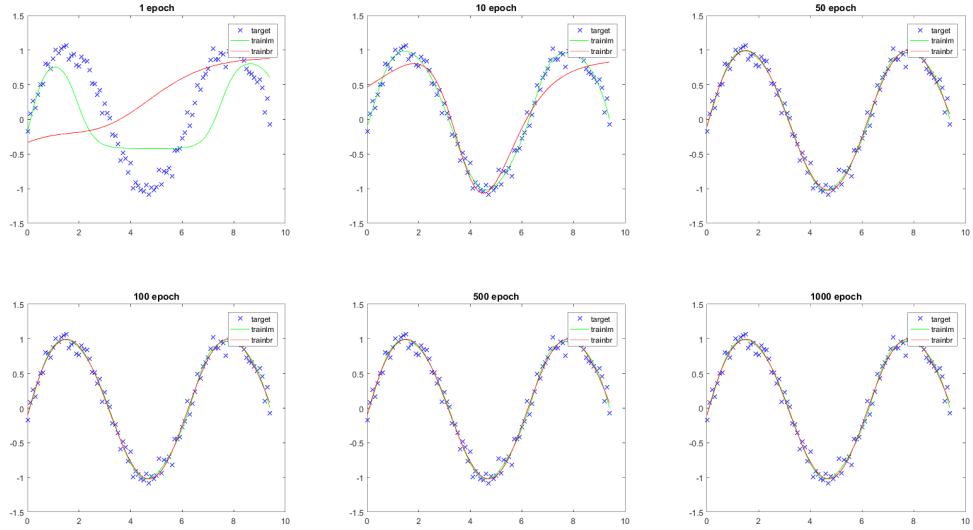
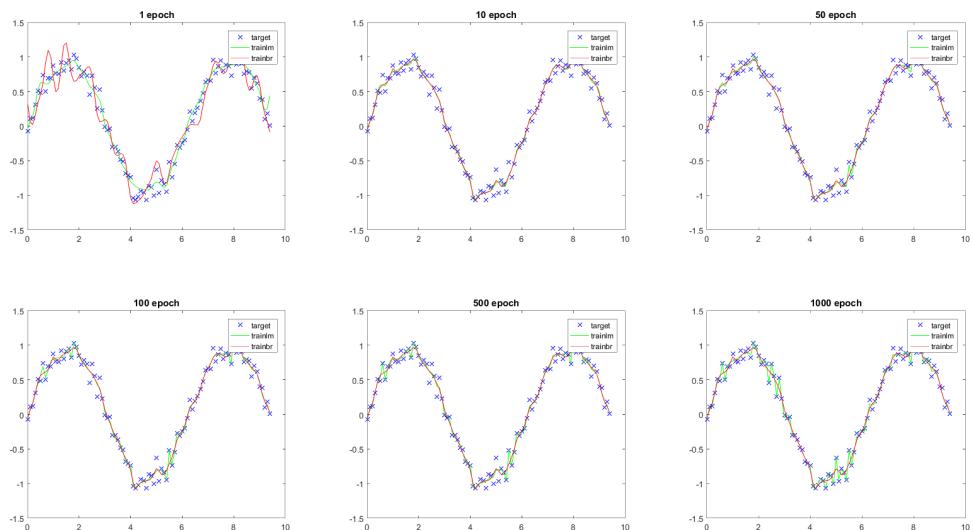


Figure 3: Results of the FFNN prediction using *Bayesian regularization backpropagation* and *Levenberg-Marquardt* algorithms in a noisy data using 30 neurons in the hidden layer



## 2 S2: Recurrent neural networks

### 2.1 Hopfield networks

Extreme cases of the noise parameter and number of iterations were tested on the *hopdigit* example. Figures 4 to 7 show the results of 4 extreme experiments.

Results in figures 4 and 6 show unstable states due to the limited number of iterations during the process. Hence the results did not converge. Nevertheless, the unstable states are relatively *closed* to the stable states shown in figures 5 and 6.

Results in figures 5 and 7 after a bigger amount of iterations. It is clear that the noise (which implies the initial position of the input) affects the final result, a bigger noise (bigger distance from the target attractor) results in an unexpected result, as shown in figure 7.

After an exhaustive iteration, no spurious states were found.

Figure 4: Results of the *hopdigit* function with noise=4 and numiter=10.



Figure 5: Results of the *hopdigit* function with noise=4 and numiter=100.



Figure 6: Results of the *hopdigit* function with noise=100 and numiter=10.



Figure 7: Results of the *hopdigit* function with noise=100 and numiter=100.



## 2.2 Elman network

Several experiments were done using different training points, epochs and neurons. To avoid a straightforward conclusion, three iterations per setup were run. Table 1 shows the results.

This experiments suggests that the number of neurons is a critical hyperparameter in the Helman network. There is a correlation between the recall and the neurons. There is a clearly negative effect when the number of neurons are increased. The number of epochs and training points increase the Recall and decrease the MSE in all the settings.

Table 1: Results of time-series predictions using an Elman network with different parameters

Training points	Epochs	Neurons	R1 - MSE1	R2 - MSE2	R3 - MSE3	Avg R - Avg MSE
300	100	50	0.8046 - 0.28165	0.86376 - 0.19918	0.82022 - 0.21337	0.82952 - 0.23140
300	100	100	0.25316 - 0.74488	0.84409 - 0.22219	0.74083 - 0.24923	0.61269 - 0.40543
300	100	500	0.066352 - 4.1895	-0.37543 - 1.7456	0.36863 - 2.6571	0.01985 - 2.86406
300	500	50	0.87958 - 0.17177	0.89757 - 0.14329	0.919663 - 0.13059	0.89893 - 0.44560
300	500	100	0.83692 - 0.19721	0.77193 - 0.35833	0.8484 - 0.21193	0.81908 - 0.25582
300	500	500	0.26619 - 2.4972	0.20328 - 2.8426	0.77812 - 0.33735	0.41586 - 1.89238
300	1000	50	0.90988 - 0.1029	0.92109 - 0.089881	0.89314 - 0.14156	0.90803 - 0.11144
300	1000	100	0.79292 - 0.34087	0.85052 - 0.23548	0.93913 - 0.095492	0.86085 - 0.22394
300	1000	500	0.88905 - 0.13257	0.77324 - 0.2473	0.74208 - 0.37589	0.80145 - 0.25192
500	100	50	0.91346 - 0.70402	0.89814 - 0.12147	0.86020 - 0.17218	0.89363 - 0.33255
500	100	100	0.31444 - 1.0704	0.73214 - 0.35368	0.75021 - 0.26852	0.59893 - 0.56346
500	100	500	0.058904 - 4.2432	0.082859 - 2.4775	-0.028816 - 1.9908	0.03764 - 2.9038
500	500	50	0.66104 - 0.83886	0.77711 - 0.29763	0.82016 - 0.23519	0.75277 - 0.45722
500	500	100	0.88826 - 0.15023	0.7454 - 0.28408	0.8295 - 0.24462	0.82105 - 0.22631
500	500	500	0.3947 - 2.0466	0.71847 - 0.30259	0.41324 - 2.136	0.50880 - 1.49506
500	1000	50	0.96242 - 0.053706	0.94135 - 0.082144	0.8931 - 0.12993	0.93229 - 0.08859
500	1000	100	0.81748 - 0.25344	0.89584 - 0.14696	0.82683 - 0.25421	0.84671 - 0.21820
500	1000	500	0.54997 - 1.2887	0.88085 - 0.15332	0.9128 - 0.11874	0.78120 - 0.52015

### 3 S3: Unsupervised learning: PCA and SOM

#### 3.1 Principal Component Analysis

The PCA algorithm was implemented using the Matlab functions provided in the exercise document. Figure 8 shows the mean of the whole data set as well as the first element and its corresponding reconstructed vector using different  $k$ . Figures in 9 shows the eigenvalues of the components and the reconstruction error using different  $k$  values.

Comparing figures in 9, there exists a relation between the reconstruction error and the eigenvalues. This implies that the number of  $k$  components directly influence the quality of the reconstruction. This can be verified on figure 8, the reconstructed image from  $k=1$  to  $k=2$  clearly shows an aggressive improvement.

Figure 8: Left: Image of the mean vector of the whole data set. Right: Image of the first element of the dataset with different reconstructions.

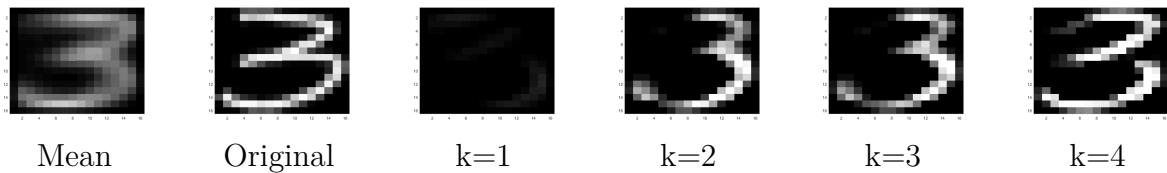
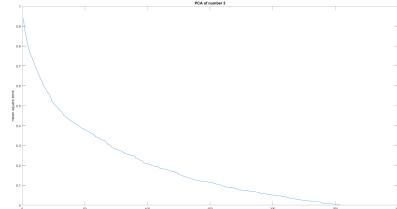


Figure 9: Plots of PCA exercise.

(a) Plot of the eigenvalues of the US Postal Service dataset. Axis X represents the  $i$ th eigenvalue. Axis Y represents the eigenvalues.



(b) Plot of the reconstructions error (MSE) from  $k=1$  to  $k=256$



#### 3.2 Competitive learning with SOM's

Figure 10 shows the results of different experiment in **SOM\_concentric\_cylinders.m** script. The model used a hexagonal topology with a grid size of  $5 \times 5 \times 5$ . *Link distance* (`linkdist`) and *Manhattan distance* (`mandist`) were tested with different epochs.

It is concluded that the `linkdist` simulates a better cylinder shape (making it a heavily connected graph) than the `mandist`.

Figure 11 shows some results of different experiment in **example\_SOM\_iris.m** script. The model used different topologies and grid size of 10x10. *Link distance* (*linkdist*) and *Manhattan distance* (*mandist*) were tested with equal amount of epochs (100).

A remark conclusion is that hexagonal topology shows a smoother topology than the grid topology. Additionally *linkdist* gives harder limits on the decision surface than *mandist* as shown in figure 11.

Figure 10: Top: Trajectory of the neurons using Manhattan distance. Bottom: Trajectory of the neurons using Link distance.

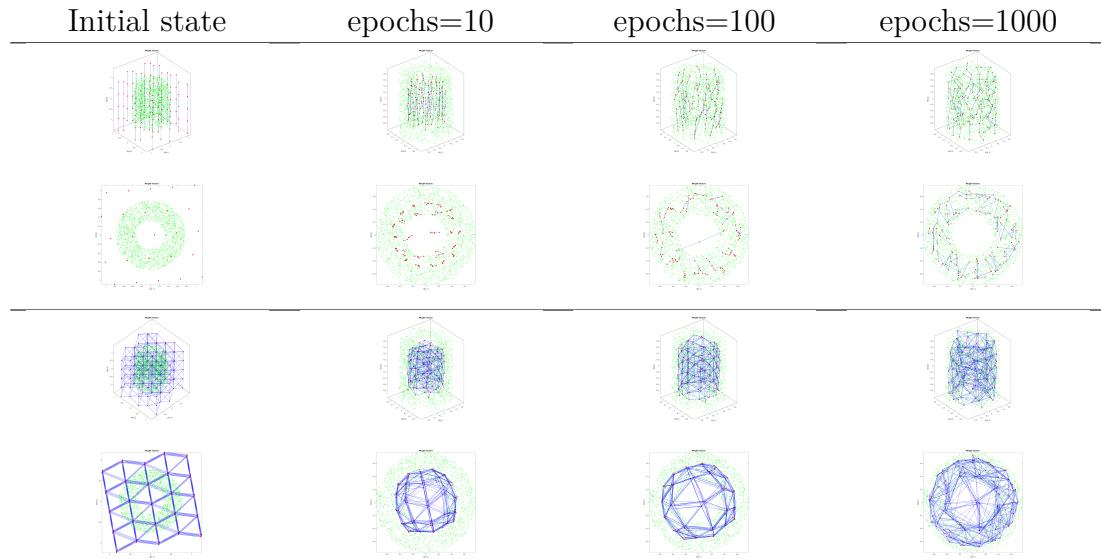
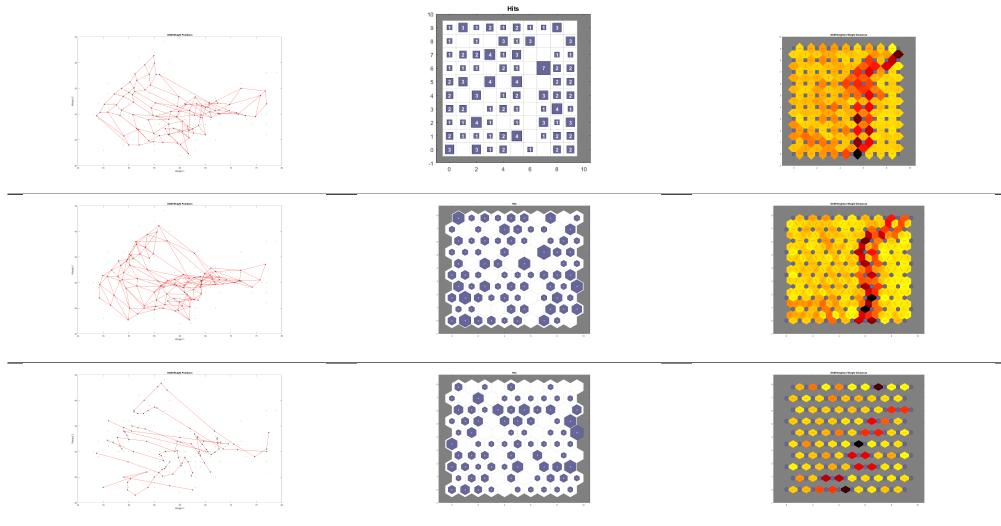


Figure 11: Top: Plots of SOM using grid topology and *linkdist*. Middle: Plots of SOM using hexagonal topology and *linkdist*. Bottom: Plots of SOM using hexagonal topology and *mandist*.



## 4 S4: Deep Learning: stacked autoencoders and convolutional Neural networks

### 4.1 Stacked autoencoders

After analyzed **DigitClassification.m** script, several experiments were run. Tables 2 and 3 show the results of different models with different hyperparameters.

All the models could beat the original one (second column in table 2) but the model with third hidden layers (seventh column in table 2) before the fined-tuning. However after the fined-tuning, the DeppNets could not overcome the original setting. Even the best model (fifth column in table 2) was not able to perform better.

Three interesting remark are noticed. The model with best results after fined-tuning took around 7 minutes per run. That is a significant amount of time compared with the other models, even the second best model before fined-tuning (sixth column in table 2) took around 3 minutes per run. This is a extensive impact in the time complexity of the overall model where the gain was relatively small (1.03). Is really it a good tradeoff between time and accuracy?. It precisely depends on the application, for instance in applications where the accuracy is highly important such as the health-care sector.

However, this tradeoff between time and accuracy can be easily solved using the fined-tuning phase. Clearly the results shown that the fined-tuning phase improve all models, even the worst model that used 3 hidden layers. It was completely not expected that the accuracy of the models improved to beat even the FFNN. Fined-tuning phase has shown that is an important phase in order to improve, even in the worst cases, the accuracy of the model.

Finally, after fined-tuning all the models beat the FFNN with 1 and 2 layers. However the overall performance are not bad whatsoever. Reaching an average of 96.94 and 96.45, FFNN has shown that it is a very powerful as well as a simple model. As aforementioned, FFNN can be implemented in many applications that do not require heavy critical decision.

Table 2: Results of DeepNet before fined-tuning. (# layer, max epochs, hidden units)

	1, 400, 100 2, 100, 50	1, 400, 100 2, 400, 50	1, 100, 100 2, 400, 50	1, 400, 400 2, 100, 200	1, 400, 200 2, 100, 100	1, 400, 100 2, 100, 50 3, 50, 25
1	86.72	94.16	91.28	99.18	98.24	28.84
2	81.18	94.70	93.12	99.02	97.76	29.80
3	89.16	95.36	93.70	98.32	97.84	13.68
4	78.66	93.88	92.26	98.82	97.92	17.18
5	82.48	92.62	91.94	98.90	97.34	17.18
Avg	<b>83.64</b>	<b>94.14</b>	<b>92.46</b>	<b>98.85</b>	<b>97.82</b>	<b>21.34</b>

Table 3: Results of DeepNet after fined-tuning. (# layer, max epochs, hidden units)

	1, 400, 100 2, 100, 50	1, 400, 100 2, 400, 50	1, 100, 100 2, 400, 50	1, 400, 400 2, 100, 200	1, 400, 200 2, 100, 100	1, 400, 100 2, 100, 50 3, 50, 25
1	99.68	98.96	99.52	99.48	98.86	99.24
2	99.76	98.92	99.52	99.56	98.88	99.30
3	99.80	99.06	99.52	99.10	98.36	99.32
4	99.78	98.72	99.54	99.20	98.04	99.32
5	99.72	98.86	99.72	99.32	99.00	99.32
Avg	<b>99.75</b>	<b>98.90</b>	<b>99.56</b>	<b>99.33</b>	<b>98.64</b>	<b>99.30</b>

Table 4: Results using a FFNN.

1 hidden layer	2 hidden layers	
1	97.02	95.84
2	96.64	97.52
3	96.28	97.26
4	96.28	97.26
5	97.80	94.42
Avg	<b>96.94</b>	<b>96.45</b>

## 4.2 Convolutional neural networks

### What do these weights represent?

Hence those weights represent the stack of filtered images.

### What is the dimension of the input at the start of layer 6 and why?

The previous layer 5 is a MaxPooling layer. It was found that the *PoolSize* was set to 3x3 pixels and *Stride* to 2x2. For each filtered image in the stack, it reduces the input size from 11x11x3 to 5x5x3. According with the MatLab Documentation [1], the attribute *NumChannels* represents the feature maps. In this documentation they states that this input value corresponds to the number of filters in the previous convolutional layer. In this case the number of filter of layer 2 are 96. Hence, the input must take into account the shrunk filtered image by the numbers of filters. The input for layer 6 is of 4 dimensions [5,5,3,96], the same dimensionality than the output of layer 2. An interesting observation is that actually the *NumChannels* parameter of layer 6 is a vector of [48,48] and not an integer value of 96. According with [1] this cannot be set manually as a vector, it must be a integer value. I assumed the filters are split due to the CrossChannelNormalizationLayer and Matlab internally accepts that type of input for Layer 6.

### What is the final dimension of the problem? How does this compare with the initial dimension?

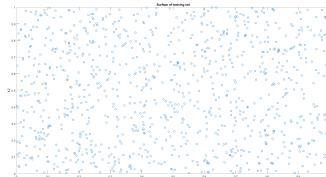
According with the ClassificationOutputLayer, it contains a single vector (1 dimension) that contains 1000 of neurons. The initial dimension was [227,227,3] which gives us a total of 154,587 elements. The reduction clearly is significant, the initial input was reduced roughly 155 times.

## 5 Final project

### 5.1 P1.1: Nonlinear regression

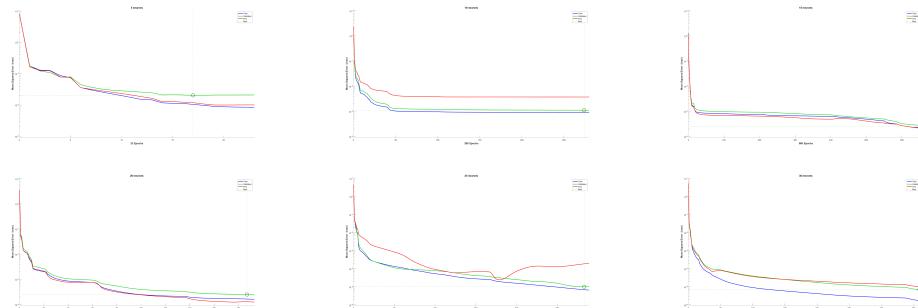
Appendix A shows the code used for this assignment. The different set to be used in as training, validation and test set were obtained using the built-in command *randperm*. The command was used to generate a random set of numbers from 1 to the size of the dataset. Those become the indices of the numbers to create the final data set. Figure 12 shows the surface of the training set.

Figure 12: Surface of training set with 1000 elements



Based on results obtained in sections **S1** and **S4**. It was chosen to use a FFNN with one hidden layer, *tansig* and *purelin* as transfer functions and *trainlm* as a learning algorithm. Default parameters according with [2]. In order to chose a suitable number of neurons in the hidden layer, different networks were tested. According with the results in figure 13, the best model was the FFNN with 30 neurons in the hidden layer.

Figure 13: Plots of FFNN performances with different number of neurons.



Finally the performance of the chosen model was assessed using the test set. A total of 100 iterations using different random seed were run in order to avoid a naive results. The MSE average of the model was 1.1736e-05. Figure 14 shows the result of one of the iterations, in this case the MSW was 1.6656e-06.

### 5.2 P1.2: Classification

The same architecture as previous section was used. Different number of neurons were tested in order to set best. Figure 19 shows the results of the test. According with the results and in contrast with the previous section, best number of neurons for this specific tasks is 45.

Figure 14: Plot of the ground truth and prediction surface.

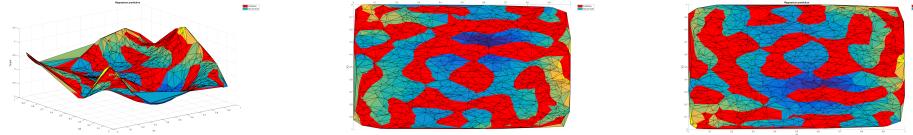
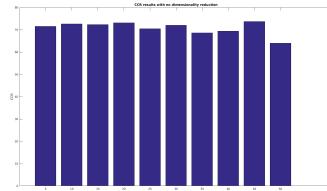
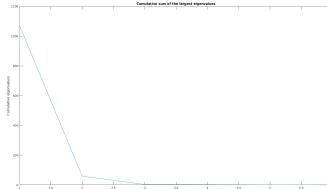


Figure 15: Results of the classification tasks using different number of neurons



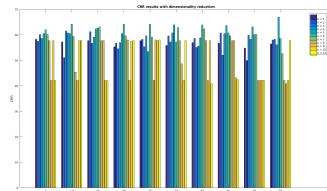
The PCA algorithm was implemented using the code of section **S3**. Different k components were tested. Using the knowledge obtained from the experiments of section **S3** and the results of figure 16 the best k components might be 3. In figure 16, one can see that there is a decay of the cumulative largest values after the third component, which state that after it, the error of the reconstruction will not be reduced dramatically.

Figure 16: Results of the classification tasks using different number of neurons



A final experiment was set, knowing using the lower dimension data set. In order to have a big picture of the different k values and different FFNN models, all the combinations were run. Figure 17 shows the results. It was noticed that k=3, although it was shown as a good value for PCA, the best k values lie between 5 and 6, which depends on the number of neurons used in the model.

Figure 17: Results of the classification tasks using different number of neurons



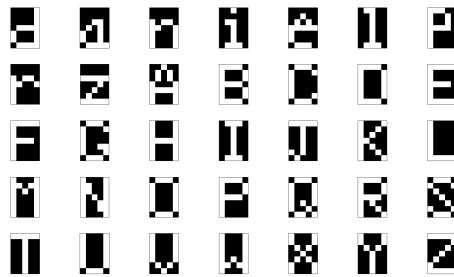
Additionally, the overall performance is lower, which indicates that the dataset is sensible to dimensionality reduction. It is reasonable since the dimension is not as big as the

dimension of the data set used in section **S3**. Besides, the the dataset might be high correlated which decreased the performance of the PCA algorithm which can imply a reduction of the performance of the FFNN. It is concluded that inputs are as important as the model of the neural networks; the better the quality of the inputs (features), the better the performance of the neural network.

### 5.3 P2: Character recognition with Hopfield networks

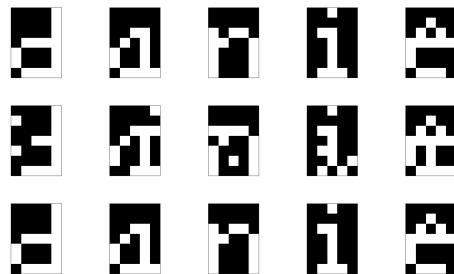
As warm-up task, the required letters were created and added to the vocabulary. Figure 18 shows the final results.

Figure 18: Set of letter in the dataset (attractors)



In order to tested the accuracy of the Hopfield network, 10,000 iterations were tested using different distorted patterns. Figure 19 shows the results of one of the iterations. This Hopfield network could recall perfectly the numbers, a MSE average of 0 was obtained. After an exhaustive set of tests, no spurious patterns were found.

Figure 19: Top: original letter (attractor state). Middle: distorted letter by 3 pixeles. Bottom: reconstruction using the Hopfield network.



After the initial experiment, different values of  $P$  were tested with 100 each. The results of the experiments are shown in figure 20. This experiments shown that the maximum capacity of attractors to generated a perfect recall is roughly 5. This empirical results shown that the theoretical loading capacity using the Hebb-rule, shown in figure 21, can be a little off with the actual capacity. However it is a good estimation in order to prevent unexpected states (spurious patterns).

Figure 20: Average of absolute error and number of spurious patterns found with different P values.

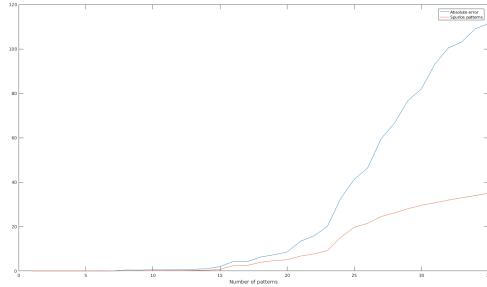
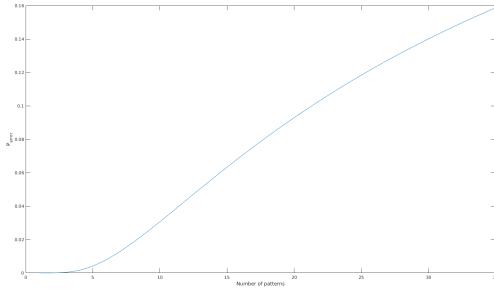
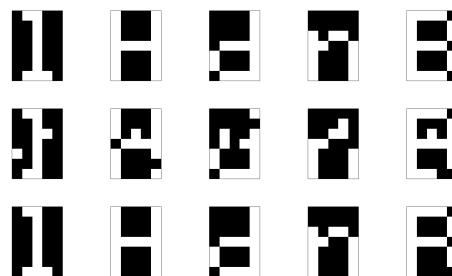


Figure 21: Theoretical loading capacity using the Hebb-rule.



Finally an alternative to overcome this downside of the Hopfield network, one can rely on the powerful FFNN. In fact, a FFNN was implemented with one hidden layer and using the default settings and 10 neurons in the hidden layer and 35 neurons in the output layer. A dataset of 1,000 was created. 80% was used as training test and validation test and the rest was used as test set. The overall CCR of the FFNN was 96.71%. Figure 22 shows the results of the first 5 predictions.

Figure 22: Top: Original letter. Middle: distorted letter by 3 pixels. Bottom: prediction of the FFNN.



## A Code: Nonlinear regression

```
1 clear ;
2 clc ;
3 close all ;
4
5 load( 'Data_Problem1_regression.mat' )
6
7 %student number r0605947
8
9 % ***** Preprocessing *****
10
11 Tnew = (9*T1 + 7*T2 + 6*T3 + 5*T4 + 4*T5) / (9+7+6+5+4);
12 Inputs = [X1 X2];
13
14 % ***** Task 1 *****
15 % Get independent samples
16 rng(97654); %Use to have the same indices to compare results
17 Indices = randperm( size(Tnew,1) );
18
19 Itraning = Indices(1:1000);
20 Ivalidation = Indices(1001:2000);
21 Itest = Indices(2001:3000);
22
23 Xtraning = Inputs(Indices(1:1000),:);
24 Tvalidation = Tnew(Indices(1:1000),:);
25
26 Xtest = Inputs(Indices(1001:2000),:);
27 Ttest = Tnew(Indices(1001:2000),:);
28
29 % Plot the surface of training set
30 figure;
31 plot(Xtraning(:,1),Xtraning(:,2), 'o');
32 title('Training set surface');
33 xlabel('X1');
34 ylabel('X2');
35
36 % ***** Task 2 *****
37
38 max_epochs = 1000;
39
40 % creation of networks - 5 neurons
41 net5=feedforwardnet(5, 'trainlm');
42 % training
43 net5.trainParam.epochs=max_epochs;
44 [ net5 ,tr5]=train( net5 ,Xtraning , Tvalidation );
```

```

45 %creation of networks - 10 neurons
46 net10=feedforwardnet(10, 'trainlm');
47 %training
48 net10.trainParam.epochs=max_epochs;
49 [ net10 , tr10]=train( net10 , Xtraning' , Tvalidation' );
50
51 %creation of networks - 15 neurons
52 net15=feedforwardnet(15, 'trainlm');
53 %training
54 net15.trainParam.epochs=max_epochs;
55 [ net15 , tr15]=train( net15 , Xtraning' , Tvalidation' );
56
57 %creation of networks - 20 neurons
58 net20=feedforwardnet(20, 'trainlm');
59 %training
60 net20.trainParam.epochs=max_epochs;
61 [ net20 , tr20]=train( net20 , Xtraning' , Tvalidation' );
62
63 %creation of networks - 25 neurons
64 net25=feedforwardnet(25, 'trainlm');
65 %training
66 net25.trainParam.epochs=max_epochs;
67 [ net25 , tr25]=train( net25 , Xtraning' , Tvalidation' );
68
69 %creation of networks - 30 neurons
70 net30=feedforwardnet(30, 'trainlm');
71 %training
72 net30.trainParam.epochs=max_epochs;
73 [ net30 , tr30]=train( net30 , Xtraning' , Tvalidation' );
74
75 figure;
76 plotperform(tr5);
77 title('5 neurons');
78 figure;
79 plotperform(tr10);
80 title('10 neurons');
81 figure;
82 plotperform(tr15);
83 title('15 neurons');
84 figure;
85 plotperform(tr20);
86 title('20 neurons');
87 figure;
88 plotperform(tr25);
89 title('25 neurons');
90

```

```

91 figure;
92 plotperform(tr30);
93 title('30 neurons');
94
95
96 % ***** Task 3 *****
97
98 net=feedforwardnet(30,'trainlm');
99 %training
100 net.trainParam.epochs=350;
101 net=train(net,Xtraning',Tvalidation');
102
103 Y_eval_tr = net(Xtest');
104 Y_eval = Y_eval_tr';
105
106 figure;
107 tri = delaunay(Xtest(:,1),Xtest(:,2));
108 trisurf(tri,Xtest(:,1),Xtest(:,2),Y_eval)
109 hold on
110 tri = delaunay(Xtest(:,1),Xtest(:,2));
111 trisurf(tri,Xtest(:,1),Xtest(:,2),Ttest)
112 title('Regression prediction');
113 xlabel('X1');
114 ylabel('X2');
115 zlabel('Target');
116 hold off
117
118 err = immse(Ttest,Y_eval);
119 fprintf('\n The mean-squared error is %0.4f\n',err);
120
121 iterations = 100;
122 err_m = (iterations);
123
124 for i = 1:iterations
125 rng(i); % change seed for each iteration
126 Indices = randperm(size(Tnew,1));
127
128 Itraning = Indices(1:1000);
129 Ivalidation = Indices(1001:2000);
130 Itest = Indices(2001:3000);
131
132 Xtraning = Inputs(Indices(1:1000),:);
133 Tvalidation = Tnew(Indices(1:1000),:);
134
135 Xtest = Inputs(Indices(1001:2000),:);
136 Ttest = Tnew(Indices(1001:2000),:);

```

```

137
138
139 net=feedforwardnet(30,'trainlm');
140 %training
141 net.trainParam.epochs=350;
142 net=train(net,Xtrain,'Tvalidation');
143
144 Y_eval_tr = net(Xtest);
145 Y_eval = Y_eval_tr';
146
147 err_m(i) = immse(Ttest, Y_eval);
148 end
149
150 avg_error = mean(err_m);
151 fprintf('\n The average mean-squared error is %0.4f using %d
iterations\n', avg_error, iterations);
152
153 figure;
154 plot(err_m);

```

## B Code: Classification

```

1 clear;
2 clc;
3 close all;
4
5 % ***** Preprocessing data *****
6 rng(97654); %Use to have the same indices to compare results
7 % Load data
8 % Positive 4 and 5
9 % Negative 6
10
11 all_raw_set = importdata('winequality-red.csv',';');
12 all_set = all_raw_set.data;
13 positive_samples_4 = all_set(all_set(:,end) == 4, :);
14 positive_samples_5 = all_set(all_set(:,end) == 5, :);
15 negative_samples = all_set(all_set(:,end) == 6, :);
16
17 % Concatenate set as input set
18 X = [positive_samples_4(:,1:end-1);
19     positive_samples_5(:,1:end-1);
20     negative_samples(:,1:end-1)];
21
22 % Convert set into suitable format for output, 1 for
23 % positive and -1 for negative class and concatenate them as

```

```

24 % target set
25 n_positive = length(positive_samples_4) + length(
26     positive_samples_5);
26 n_negative = length(negative_samples);
27 Y = [ones(n_positive,1);
28       -ones(n_negative,1)];
29
30 % Split data into training set, validation set and test set.
31 % I used randperm to have an evenly distribution od the
32 % samples.
33 % Total oof samples are 1372 I roughly divided 80% for
34 % training set (1000 samples) and 20% for test set
35 % (372 samples).
36
37 n_samples = n_positive + n_negative;
38 randIdx = randperm(n_samples);
39
40 training_set = X(randIdx(1:1000),:);
41 validation_set = Y(randIdx(1:1000),:);
42 test_set = X(randIdx(1001:end),:);
43 ground_truth = Y(randIdx(1001:end),:);
44
45 % ***** task 1 *****
46
47
48 % Create network without reduced dimensionality
49 result_matrix = zeros(50);
50
51 for neurons=5:5:50
52     net1=feedforwardnet(neurons,'trainlm');
53
54     % Training and simulation
55     net1.trainParam.epochs=1000;
56     net1=train(net1,training_set',validation_set');
57
58     % Performance on test set
59     predictions = net1(test_set');
60     ccr1 = (sum(sign(predictions)==ground_truth'))*100)/
61         length(ground_truth);
62     result_matrix(neurons) = ccr1;
63 end
64 result_matrix(~any(result_matrix,2),:) = [];
65 result_matrix(:,~any(result_matrix,1)) = [];
66
67 figure;

```

```

68 x = 5:5:50;
69 bar(x,result_matrix);
70
71 % ***** task 2 *****
72
73 % Find the k optimal components
74 conv_x = cov(training_set);
75 [E,s] = eigs(conv_x);
76 cumsum_values = diag(cumsum(s,100));
77
78 figure;
79 plot(cumsum_values);
80 title('Comulative sum of the largest eigenvalues');
81 xlabel('k component');
82 ylabel('Comulative eigenvalues');
83
84
85 % Process training and test sets to apply PCA
86 k = 11;
87
88 result_matrix = zeros(50,11);
89 for neurons=5:5:50
90     for k=1:11
91         % Apply PCA to training set
92         conv_x = cov(training_set);
93         [E,s] = eig(conv_x);
94         [~, indx] = sort(diag(s), 'descend');
95         E = E(:,indx);
96         projection_training = E(:,1:k)';
97         training_set_reduction = projection_training *
98             training_set';
99
100
101     % Apply PCA to test set
102     conv_x = cov(test_set);
103     [E,s] = eig(conv_x);
104     [s, indx] = sort(diag(s), 'descend');
105     E = E(:,indx);
106     projection_test = E(:,1:k)';
107     test_set_reduction = projection_test * test_set';
108
109 % ***** task 3 *****
110 % Create network with reduced dimensionality
111 net2=feedforwardnet(neurons,'trainlm');
112

```

```

113 % Training and simulation
114 net2.trainParam.epochs=1000;
115 net2=train(net2,training_set_reduction,validation_set');
116
117 % Performance on test set
118 predictions = net2(test_set_reduction);
119 ccr2 = (sum(sign(predictions)==ground_truth'))*100)/
    length(ground_truth);
120 fprintf('neurons=%f k=%f CRR=%f \n',neurons,k,ccr2);
121 result_matrix(neurons,k) = ccr2;
122 end
123 end
124
125 result_matrix(~any(result_matrix,2),:) = [];
126
127 figure;
128 x = 5:5:50;
129 bar(x, result_matrix);

```

## C Code: Character recognition with Hopfield networks

```

1 clear;
2 clc;
3 close all;
4
5 % daniel perez -> danielprz
6
7 % ***** Preprocessing data *****
8 capitals_letters = prprob;
9 lowercase_letters = lowercase;
10
11 X=[lowercase_letters,capitals_letters];
12 X(X==0)=-1;
13
14 % Print all letters
15 figure;
16 for i=1:35
17 subplot(5,7,i);
18 colormap(gray);
19 imagesc(reshape(X(:,i),5,7)',[0,1]);
20 axis image;
21 set(gca,'xtick',[],'ytick',[]);
22 end

```

```

23
24 % **** task 1 ****
25 % Create Hopfield Network
26 num_characters=5;
27 T = X(:,1:num_characters);
28 net = newhop(T);

29
30 % Distort characters
31 err = [];

32
33 for lo=1:10000
34     new_T = zeros(35, num_characters);
35     num_pix=3;

36
37     for i=1:num_characters
38         indx = randperm(35);
39         new_T(:, i) = T(:, i);
40         for j=1:num_pix
41             new_T(indx(j), i) = T(indx(j), i) * -1;
42         end
43     end

44
45 [y, Pf, Af] = sim(net, {num_characters 50}, {}, new_T);
46
47 prediction = sign(y{end});

48
49 err = [err immse(prediction, T)];
50 end

51
52 mean_error = mean(err);
53
54 figure;
55 for i=1:5
56     subplot(3,5,i);
57     colormap(gray);
58     imagesc(reshape(T(:, i), 5, 7)', [0,1]);
59     axis image;
60     set(gca, 'xtick', [], 'ytick', []);

61
62     subplot(3,5,i+5);
63     colormap(gray);
64     imagesc(reshape(new_T(:, i), 5, 7)', [0,1]);
65     axis image;
66     set(gca, 'xtick', [], 'ytick', []);

67
68     subplot(3,5,i+10);

```

```

69 colormap( gray );
70 imagesc( reshape( prediction (:, i) ,5 ,7) ', [0 ,1]) ;
71 axis image;
72 set(gca , 'xtick' ,[] , 'ytick' ,[] );
73 end
74
75 % ***** task 2 *****
76 erro_by_p = [];
77 spurious_by_p = [];
78 spurious_state = [];
79 for P=1:35
80 % Create Hopfield Network
81 T = X(:,1:P);
82 net = newhop(T);
83 err=[];
84 inner_counter = 0;
85 for iteration=1:100
86 new_T = zeros(35, P);
87 num_pix=3;
88
89 for i=1:P
90 indx = randperm(35);
91 new_T(:, i) = T(:, i);
92 for j=1:num_pix
93 new_T(indx(j), i) = T(indx(j), i) * -1;
94 end
95 end
96
97 [y,Pf,Af] = sim(net,{P 50},{},new_T);
98
99 if (y{end} ~= y{end-1})
100 fprintf('Hopfield network does not converge \n');
101 break;
102 end
103
104 prediction = sign(y{end});
105
106 err = [err sum(sum(prediction ~= T))];
107
108 for psub = 1:P
109 ynext = repmat(prediction(:, psub), 1, P);
110 if (sum(sum(ynext ~= T))==0)
111 fprintf('Found spurious state \n');
112 spurious_state = [spurious_state prediction(:, psub)];
113 inner_counter = inner_counter + 1;

```

```

114         end
115     end
116 end
117
118 spurious_by_p = [spurious_by_p (inner_counter/100)];
119 erro_by_p = [erro_by_p mean(err)];
120 end
121
122 figure;
123 plot(erro_by_p);
124 hold on;
125 plot(spurious_by_p);
126
127 figure;
128 for i=1:10
129     subplot(1,10,i);
130     colormap(gray);
131     imagesc(reshape(spurious_state(:,i),5,7)',[0,1]);
132     axis image;
133     set(gca,'xtick',[],'ytick',[]);
134 end
135
136 % Theoretical Heb-rule
137 figure;
138 Xs = 1:size(X,2);
139 sig = sqrt(Xs/size(X,1));
140 Z = 1./sig;
141 P_error = normcdf(ones(size(Xs))*Inf) - normcdf(Z);
142 plot(Xs,P_error);
143
144 % ***** task 3 *****
145 num_characters=25;
146 T = X(:,1:num_characters);
147 Target =
148     [T T T T T T T T T T T T T T T
149      T T T T T T T T T T T T T T T T T T T];
150
151 t_length = length(Target);
152
153 % Training set
154 training_set = zeros(35, t_length);
155 num_pix=3;
156
157 for i=1:t_length
158     indx = randperm(35);
159     training_set(:, i) = Target(:, i);

```

```

160 for j=1:num_pix
161     training_set(indx(j),i) = Target(indx(j),i) * -1;
162 end
163
164
165
166 randIdx = randperm(1000);
167 t_set = training_set(:,randIdx(1:800));
168 validation_set = Target(:,randIdx(1:800));
169 test_set = training_set(:,randIdx(801:end));
170 ground_truth = Target(:,randIdx(801:end));
171
172
173 net1=feedforwardnet(10,'trainlm');
174
175 % Training and simulation
176 net1.trainParam.epochs=500;
177 net1=train(net1,t_set,validation_set);
178
179 % Performance on test set
180 predictions = net1(test_set);
181 predictions = sign(predictions);
182
183 ccr1 = (sum(sum(predictions == ground_truth)))/(200*35);
184
185 figure;
186 for i=1:5
187     subplot(3,5,i+5);
188     colormap(gray);
189     imagesc(reshape(test_set(:,i),5,7)',[0,1]);
190     axis image;
191     set(gca,'xtick',[],'ytick',[]);
192
193     subplot(3,5,i+10);
194     colormap(gray);
195     imagesc(reshape(predictions(:,i),5,7)',[0,1]);
196     axis image;
197     set(gca,'xtick',[],'ytick',[]);
198
199
200     subplot(3,5,i);
201     colormap(gray);
202     imagesc(reshape(ground_truth(:,i),5,7)',[0,1]);
203     axis image;
204     set(gca,'xtick',[],'ytick',[]);
205 end

```

## References

- [1] *convolution2dlayer.* <https://nl.mathworks.com/help/nnet/ref/convolution2dlayer.html>. Accessed: 2017-05-27.
- [2] *Create, configure, and initialize multilayer neural networks.* <https://nl.mathworks.com/help/nnet/ug/create-configure-and-initialize-multilayer-neural-networks.html>. Accessed: 2017-05-27.