Assignment 2

Computational Intelligence, SS2018

Team Members		
STRUGER	Patrick	01530664
BÖCK	Manfred	01530598
HAUPT	Anna	01432018

Contents

1 Regression with Neural Networks

1.1 Simple Regression with Neural Networks

(a) **Learned function** In the function ex_1_1_a in file nn_regression.py:

- Write code to train a neural network on the training set using the regressor method fit, and compute the output predicted on the testing set using the method predict.
- Plot the learned functions for $n_h = 2, n_h = 8$ *and* $n_h = 40$ using the test dataset. Use the function plot_learned_function in nn_regression_plot.py for the plot.

In your report:

- Include plots of the learned function and the actual function for all values of n_h.
- Interpret your results in the context of under/over fitting.

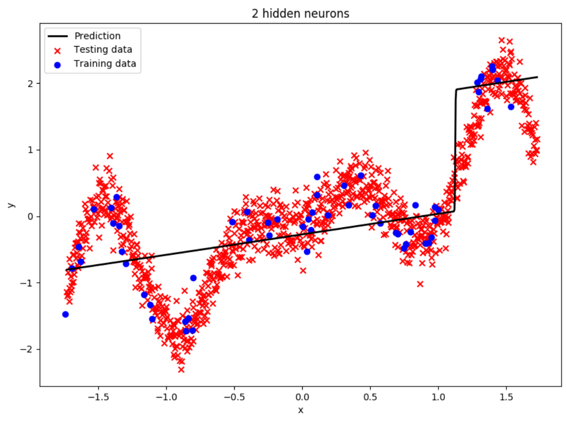

Figure 1: Simple regression with 2 neurons.

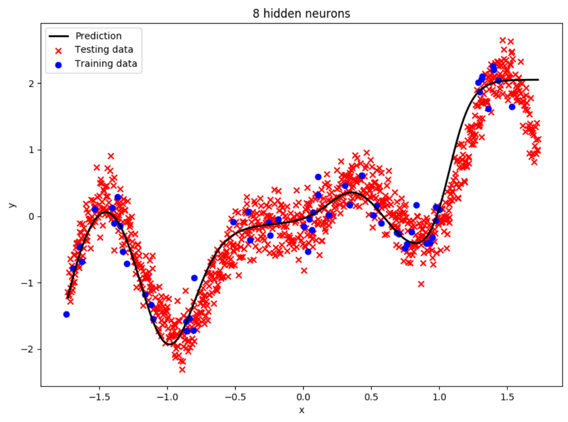

Figure 2: Simple regression with 8 neurons.

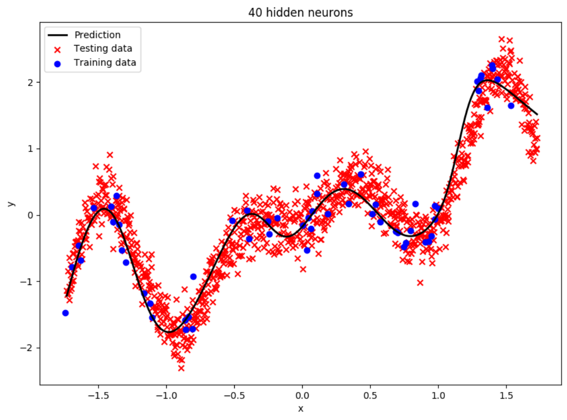

Figure 3: Simple regression with 40 neurons.

The more hidden neurons are used, the more accuracy in the prediction of the training data occurs. When using fewer neurons it is more likely that the effect of *underfitting* occurs while when using too much neurons the effect of *overfitting* may occur.

(b) **Variability of the performance of deep neural networks**
In the function calculate_mse in file nn_regression.py:

- Implement the calculation of MSE.

In the function ex_1_1_b in file nn_regression.py:

- Wrap the training together with the MSE evaluations in a for loop, and compute the MSE across 10 different random seeds. Change the random seed by passing a different value to the random_state argument of the neural network constructor.

In your report answer the following questions (one sentence is sufficient for each question):

- **What is the minimum, maximum, mean and standard deviation of the mean square error obtained on the training set? Is the min MSE obtained for the same seed on the training and on the testing set? Explain why you would need a validation set to choose the best seed?**

Random Seed	Training Error	Testing Error
1	0.05680241	0.13079428
2	0.0572133	0.19897999
3	0.0432381	0.20902372
4	0.04436419	0.11277367
5	0.05516039	0.18864168
6	0.05033942	0.18287762
7	0.06278832	0.19871679
8	0.0627082	0.18217017
9	0.05904944	0.28337206
10	0.0466659	0.19281981

The minimum mean squared error is *0.0432381* using a random seed of 4, while the maximum mean squared error of *0.06278832* occurs when using a random seed of 9. The mean is *0.05383296660111668* and the standard derivation is *0.00688234499225213*.

As shown in the table above, the minimum MSE for the testing does not indicate the minimum MSE for the training set. It could be very useful to introduce a validation set because it seems to be similar to the testing set. Therefore the validation set indicates the testing set and can be used to find the best matching seed.

- **Unlike with linear-regression and logistic regression, even if the algorithm converged the variability of the MSE across seeds is expected. Why?**

That is expected because of the *train-test-split* function, which does not split always the same way (if random state is not set). So the initial weights for calculating the MSE across seeds is not constant.

- **What is the source of randomness introduced by Stochastic Gradient Descent (SGD)? What source of randomness will persist if SGD is replaced by standard Gradient Descent?**

In Stochastic Gradient Descent the parameters for every sample observation are estimated (not just the initial one) which gives it a lot of randomness - each sample is evaluated iteratively. The initial random configuration will persist when replacing SGD by standard Gradient Descent.

(c) **Varying the number of hidden neurons:**
In the function ex_1_1_c in file nn_regression.py:

- Write code to train a neural network with $n = [1, 2, 3, 4, 6, 8, 12, 20, 40]$ hidden neurons on one layer. Intialize the regressor with $max_iter = 10000, tol = 1e - 8$.
- Compute the MSE over 10 random seeds. Stack the results in an array where the first dimension corresponds to the hidden neuron number and the second dimension indexes the random seed number.
- Plot the mean and standard deviation as a function of n_h for both the training and test data using the function plot_mse_vs_neurons in nn_regression_plot.py.
- Plot the learned functions for one of the models trained with $n_h = 40$ (make sure you use $max_iter = 10000, tol = 1e - 8$). Use the function plot_learned_function in nn_regression_plot.py for the plot.

In your report:

- **What is the best value of n_h independently of the choice of the random seed?**
  The best amount of neural networks we obtained is 6.

- **Include plots of how the MSE varies with the number of hidden neurons.**

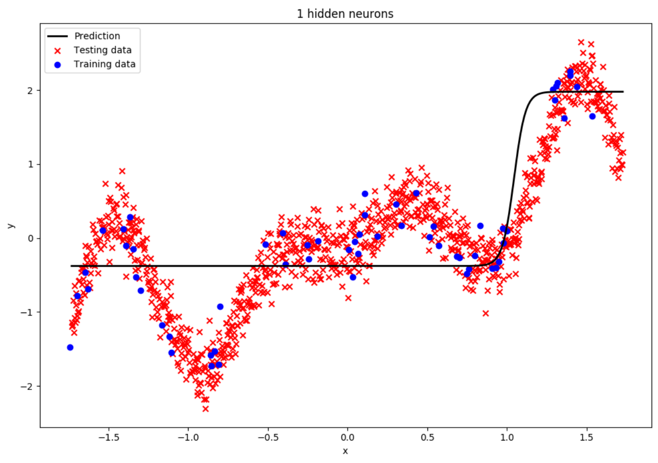

Figure 4: Regression with 1 neuron
$\rightarrow$ *underfitting.*

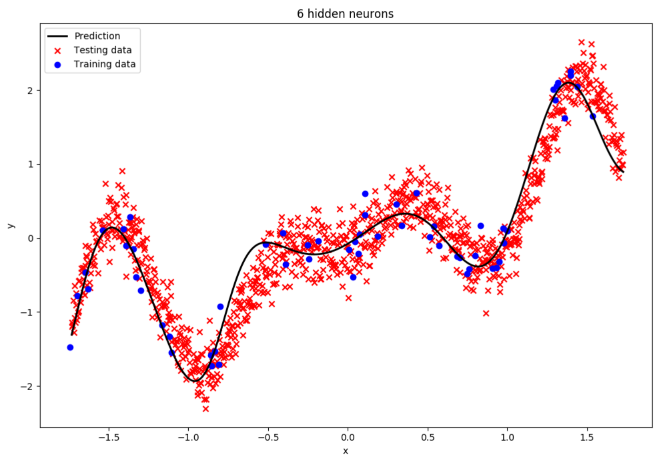

Figure 5: Regression with 6 neuron
$\rightarrow$ *mostly accurate.*

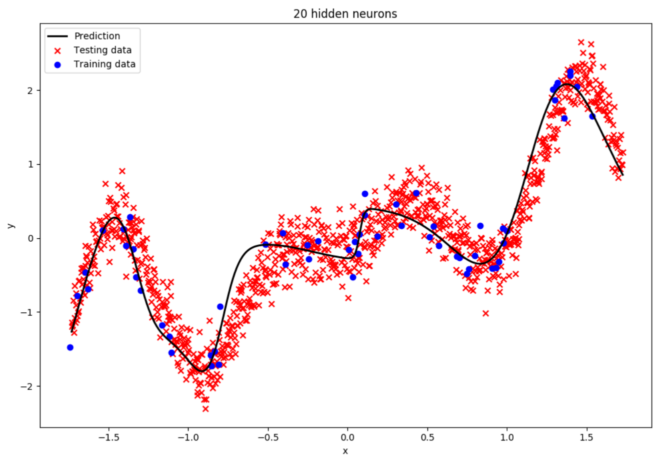

Figure 6: Regression with 20 neuron
$\rightarrow$ *overfitting*

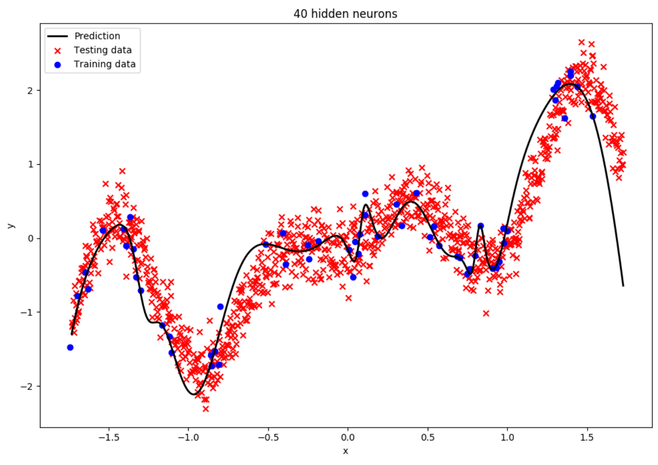

Figure 7: Regression with 40 neuron
$\rightarrow$ *overfitting*

- **Interpret and discuss your results in the context of over/under fitting**

  When using neurons under 6, we got the problem of underfitting, while when using more than 40 neurons the effect of overfitting may occur.

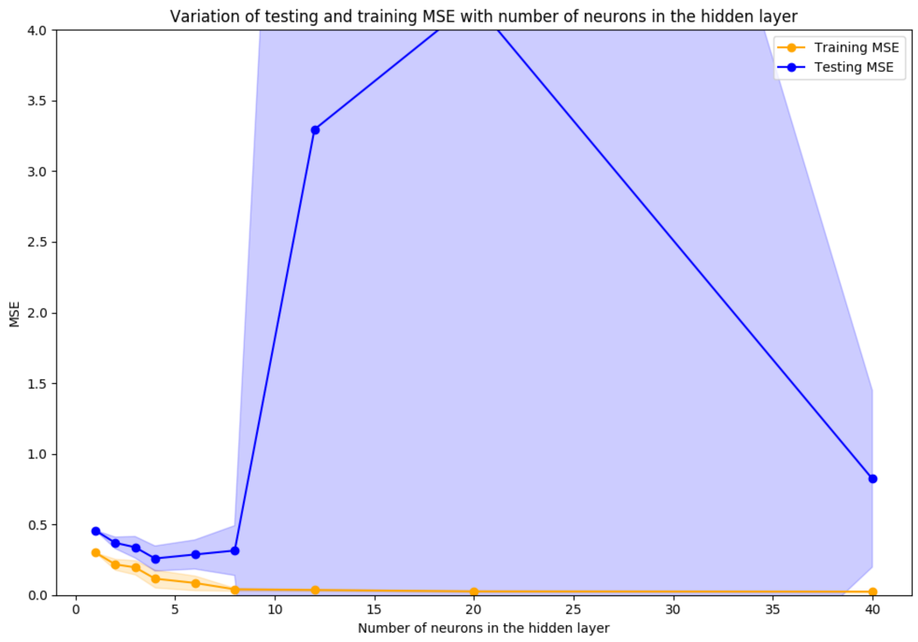

Figure 8: The mean and standard deviation as a function of n_h for both the training and test data.

(d) **Variations of MSE during training:**
In the function ex_1_1_d in file nn_regression.py:

- Write code to train a neural network with $n_h \in 2, 8, 40$ hidden neurons on one layer and calculate the MSE for the testing and training set at each training iteration for a single seed, say 0. To be able to calculate the MSEs at each iteration, set warm_start to True and max_iter to 1 when initializing the network. The usage of warm_start always keeps the previously learnt parameters instead of reinitializing them randomly when fit is called. Then, loop over iterations and successively call the fit function and calculate the MSE on both datasets. Use the training solver 'lbfgs', for 10000 iterations. Stack the results in an array with where the first dimension correspond to the number of hidden neurons and the second correspond to the number of iterations Use the function plot_mse_vs_iterations in nn_regression_plot.py to plot the variation of MSE with iterations.
- Replace the solver by 'sgd' or 'adam' and compute the MSE across iterations for the same values of n_h.

In your report, answer the following questions:

- **Is the risk of overfitting increasing or decreasing with the number of hidden neurons?**

  In our case lbfgs seems to be perform best.
  Overfitting increases with the number of hidden neurons.

- **'adam' is a variant of 'sgd' and both are first order methods (the parameter updates are based on the gradient only), whereas 'lbfgs' is a second order method (the updates are also based on the Hessian). Which methods seem to perform best in this problem ? What feature of stochastic gradient descent helps to overcome overfitting ? The neural network is rather small as compared to what is used is real-life problems, according to your analysis which solver will be more appropriate when the number of neurons increases?**

  To overcome overfitting are the regularization parameter alpha, early stopping or mini-batch learning is used. If the number of neurons increases first order methods are more appropriate (e.g adam solver).

- **Include the plot of the variations of the MSE with three different number of hidden neurons for each solver.**

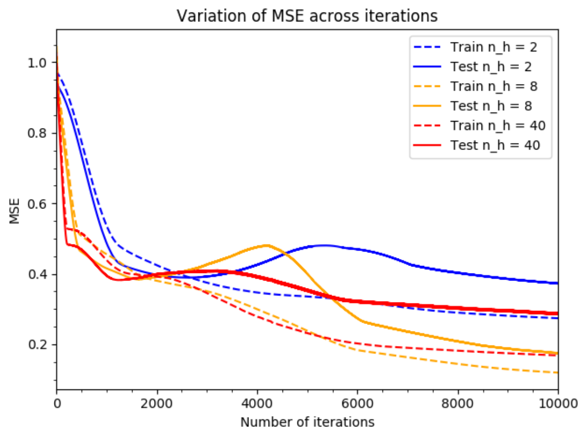

Figure 9: Variation with Adam Solver.

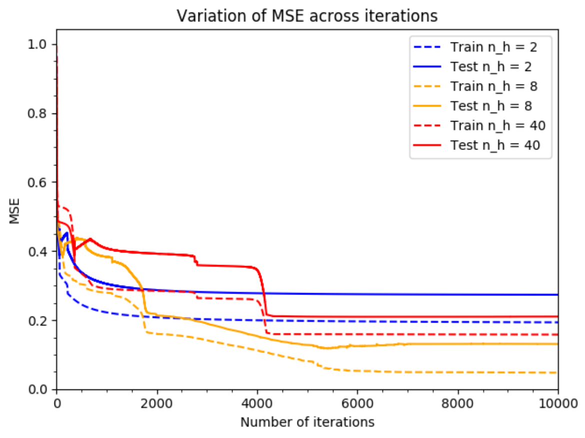

Figure 10: Variation with Lbfgs Solver.

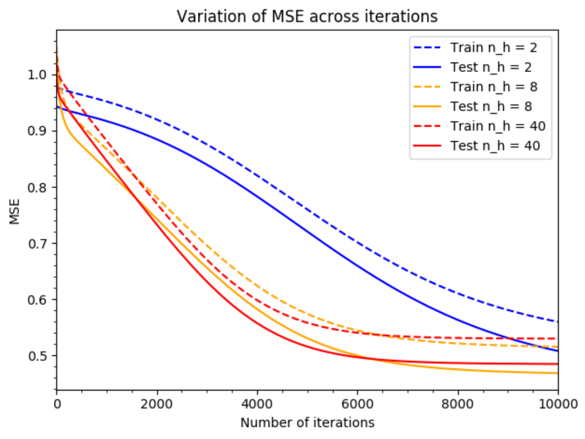

Figure 11: Variation with Sgd Solver.

1.2 Regularized Neural Networks

Now we want to investigate different regularization methods for neural networks, i.e. weight decay and early stopping. Use the same dataset as before.

(a) **Weight Decay:**
Here, we train the network with different values of the regularization parameter α. The loss function in this case looks like this:

$$mser eg = mse + \frac{\alpha}{2n} \sum_i w_i^2$$

In the function ex_1_2_a in file nn_regression.py:

- Write code to train a neural network with $n = 40$ hidden neurons with values of alpha $\alpha = [10-8, 10-7, 10-6, 10-5, 10-4, 10-3, 10-2, 10-1, 1, 10, 100]$. Stack your results in an array where the first axis correspond to the regularization parameter and the second to the number of random seeds. Use the training solver 'lbfgs', for 200 iterations and 10 different random seeds.
- Plot the variation of MSE of the training and test set with the value of α. Use the function plot_mse_vs_alpha in nn_regression_plot.py to plot the MSE variation with α.

In your report:

- **Include plots of the variation of MSE of the training and test set with the value of α.**

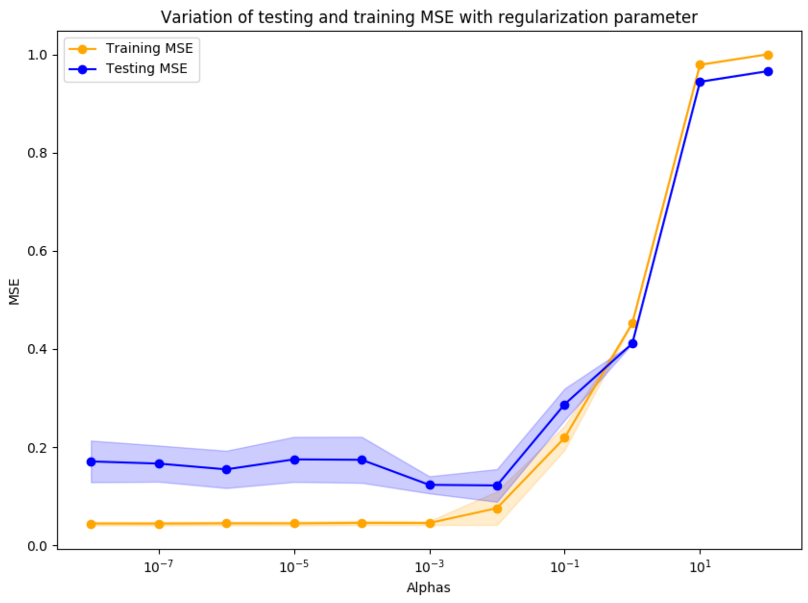

Figure 12: Variation of MSE of the training and test set with the value of α.

- **What is the best value of α?**
The best value for α seems to be 10^{-2}.

- **Is regularization used to overcome overfitting or underfitting? Why?**
Regularization is used to keep the value of the regularization parameters as small as possible. Therefore it is used to overcome overfitting because it prefers simple models.

(b) **Early Stopping:**
This question demonstrates how early stopping is very efficient at reducing overfitting. To put ourself in extreme overfitting condition, we add some noise to the training data. This is already done in nn_regression_main.py.
In the function ex_1_2_b in file nn_regression.py:

- Early stopping requires the definition of a validation set. Split your training set so that half of your old training set become your new training set and the rest is your validation set. Watch out, it is crucial to permute the order of the training set before splitting because the data in given in increasing order of x.
- Write code to train a neural network with $n = 40$ and $\alpha = 10 - 3$ on each selection of the training set. Train for 2000 iterations using the 'lbfgs' solver for 10 different random seeds and monitor the error on each set every 20 iterations. For each individual seed, generate the list of (1) the test errors after the last iteration, (2) the test errors when the error is minimal on the validation set, (3) the ideal test error when it was minimizing the error on the test set.
- Use the function plot_bars_early_stopping_mse_comparison in nn_regression_plot.py to plot bar plats comparing MSE for early stopping with last iteration and the ideal case.

In your report:

- **Include the bar plots to compare the errors on the test sets at the last training iterations, at early stopping and when it is minimal.**

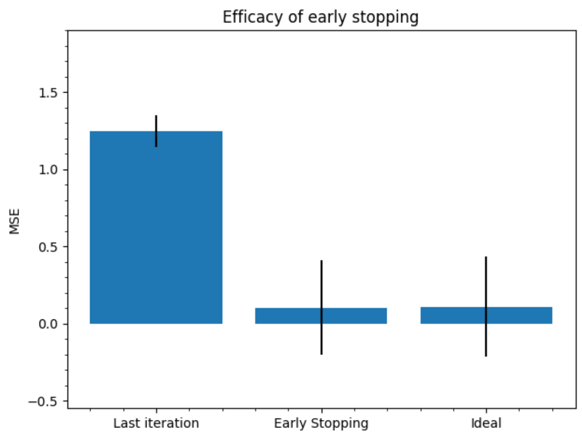

Figure 13: Early stopping.

- **In the light of question 1.1.b) is it expected that early stopping happens (validation error is minimized) at the same iteration number for all random seeds? Is it coherent with your results?**

  It is not expected that early stopping happens at the same iteration number for all random seeds because of the different initial weights set by the random state parameter. This is also shown in our results.

- **Early stopping in its standard form is a little different, instead of stopping when the validation error is minimized, one stops training as soon as the validation error increases. What are the pros and cons of those standard form of early stopping and the one you implemented?**

  The standard method of early stopping stops the training when the validation error increases, this is why this method can be faster but the validation error can be higher.

(c) **Combining the tricks:**
In the function ex_1_2_c in file nn_regression.py:

- Combining the results from all the previous questions, train a network with the ideal number of hidden neurons, regularization parameter and solver choice. Use 10 seeds, a validation set and early stopping to identify one particular network (a single seed) that performs optimally.

In your report:

- **Explain your choice of number of hidden neurons, regularization parameter and solver. Then describe in a short paragraph but rigorously the protocol followed to identify the optimal random seed (mention all the parameter you chose such as).**

  We decided to use 40 neurons for the calculation for reliable measurements because we used this amount of neurons for the previous calculations, so we are able to get valid and comparable data. For the regularization parameter alpha we used 10^{-2} regarding the best value for alpha from exercise 1.2a. By experimenting with different values for the solver and regularization parameter, we got to the conclusion that the solver *lbfgs* delivered the best results for our purpose.

  During the iteration over the random seeds we calculate the MSE for the training, testing and validation sets 100 times. We use early stopping to identify one particular network as mentioned in the assignment sheet. We then calculate the early stopping index in our MSE lists by using the *argmin* function to evaluate the best matches.

- **Report the mean and standard deviation of your training, validation and testing error. Report the training, validation and testing error of your optimal random seed.**

  Mean Square Error Test:
  - Mean: *1.1399337348359726*
  - Standard deviation: *0.0023326551020294577*

  Mean Square Error Training:
  - Mean: *1.0163030942948361*
  - Standard deviation: *7.195428704528305e-05*

  Mean Square Error Validation:
  - Mean: *0.9796973640162612*
  - Standard deviation: *0.00038479446712075645*

  Optimal seed: 1
  - Training MSE with optimal seed: *2*
  - Testing MSE with optimal seed: *2*
  - Validation with optimal seed: *1*

2 Face Recognition with Neural Networks

2.1 Pose Recognition

In the function ex_2_1 in file nn_classification.py:

- Write code to train a feed-forward neural network with 1 hidden layers containing 6 hidden units for pose recognition. Use dataset2 for training after normalization, 'adam' as the training solver and train for 200 iterations.
- Calculate the confusion matrix
- Plot the weights between each input neuron and the hidden neurons to visualize what the network has learnt in the first layer.

In your report:

- **Include the confusion matrix you obtain and discuss. Are there any poses which can be better separated than others?**

  The Confusion Matrix we obtained:

	Straight	**Left**	**Right**	**Up**
Straight	126	3	5	7
Left	1	138	0	2
Right	1	0	136	1
Up	11	4	1	128

  Table 1: Confusion Matrix

  The higher the weight of the confusion matrix, the better the accuracy of the pose recognition. In our matrix we obtain high weights for *left* and *right* values. They have more accuracy than *straight* or *up*. But the accuracy for up and straight poses is still very accurate.

- **Can you find particular regions of the images which get more weights than others?**

  In our confusion matrix, particular the *left*, *right*, *straight* and the *up* regions got more weights than others. The higher the diagonal values of the matrix, the more accuracy we have in our pose recognition.

- **Include all plots in your report.**

Figure 14: Recognized Poses

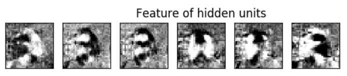

Figure 15: Weighted Regions

2.2 Face Recognition

In the function ex_2_2 in file nn_classification.py:

- Write code to train a feed-forward neural network with 1 hidden layer containing 20 hidden units for recognising the individuals. Use dataset1 for training, 'adam' as the training solver and train for 1000 iterations. Use dataset2 as the test set.
- Repeat the process 10 times starting from a different initial weight vector and plot the histogram for the resulting accuracy on the training and on the test set (the accuracy is proportion of correctly classified samples and it is computed with the method score of the classifier).
- Use the best network (with maximal accuracy on the test set) to calculate the confusion matrix for the test set.
- Plot a few misclassified images.

In your report:

- **Include all plots in your report.**

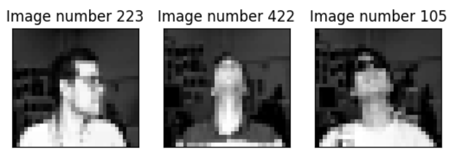

Figure 16: Face recognition with different poses.

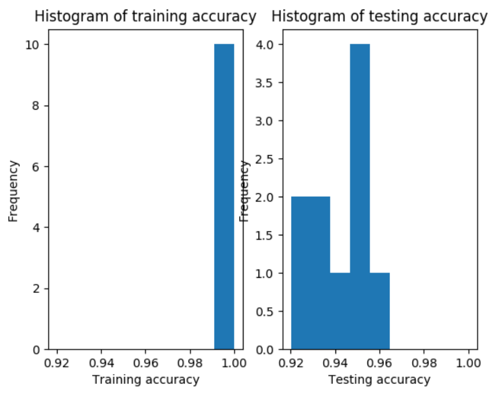

Figure 17: Accuracy histogram of training and testing set.

- **Why do different networks have different accuracies? Explain the variance in the results.**

  We computed networks with different initial weight vectors and so they differ in values and accuracy. The accuracy of the training set is very close to 1.0 while we got some fluctuations in the testing set, where the accuracy is still very high with mostly 0.95.

The Confusion Matrix we obtained:

27	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0
0	29	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	28	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	24	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	28	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	8	19	0	0	1	0	0	0	0	0	0	0	0	0	0
0	6	0	0	0	0	0	23	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	29	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	28	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	28	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	23	0	0	0	2	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	29	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	27	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	29	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	29	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	27	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	29	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	29	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	28

Table 2: Confusion matrix generated by the network with the most accuracy on training set.

- **Do the misclassified images have anything in common?**

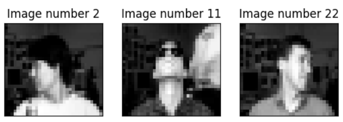

Figure 18: Misclassified image set 1.

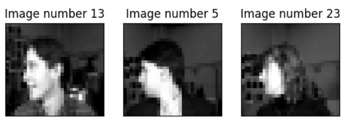

Figure 19: Misclassified image set 2.

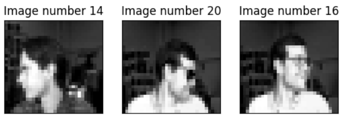

Figure 20: Misclassified image set 3.

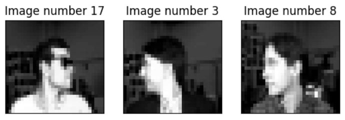

Figure 21: Misclassified image set 4.

Some of the images are very equal to each other. Also the poses, especially *left* and *right* commonly appear in all misclassified image sets.