

Computational Intelligence SEW SS18

Homework 2

Neural Networks

Anand Subramoney, Guillaume Bellec

Tutor: Christian Kopf, christian.kopf@student.tugraz.at
Points to achieve: 16 pts
Extra points: 4* pts
Info hour: 27.04.2018 15:00 - 16:00, i11
Deadline: 03.05.2018 23:55
Hand-in procedure: Submit your **python files and report** at teachcenter
Course info: <https://www.spsc.tugraz.at/courses/computational-intelligence-sew/>
Newsgroup: tu-graz.lv.ew

Contents

1 Regression with Neural Networks [7 points]	2
1.1 Simple Regression with Neural Networks [3 points]	2
1.2 Regularized Neural Networks [4 points]	3
2 Face Recognition with Neural Networks [9 points]	5
2.1 Pose Recognition	5
2.2 Face Recognition	5
3 Optional: Implementation of a perceptron [4* points]	6

General remarks

Your submission will be graded based on...

- The correctness of your results (Is your code doing what it should be doing? Are your plots consistent with what algorithm XY should produce for the given task? Is your derivation of formula XY correct?)
- The depth and correctness of your interpretations (Keep your interpretations as short as possible, but as long as necessary to convey your ideas)
- The quality of your plots (Is everything important clearly visible in the report, are axes labelled, ...?)
- **Every** result that should be graded must be included in your report.
- INOTE is an implementation-related note

For this assignment, we will be using an implementation of Multilayer Perceptron from scikit-learn. The documentation for this is available at the [scikit website](http://scikit-learn.org). The two relevant multi-layer perceptron classes are – `MLPRegressor` for regression and `MLPClassifier` for classification.

For both classes (and all scikit-learn model implementations), calling the `fit` method trains the model, and calling the `predict` method with the testing or training data set gives the predictions for that data set, which you can use to calculate the testing and training errors.

1 Regression with Neural Networks [7 points]

Throughout this task, use the `MLPRegressor` class and use the 'logistic' activation function for the hidden layer (using the `activation` parameter). The output layer uses an identity activation function by default and the loss function used is mean squared error (MSE).

1.1 Simple Regression with Neural Networks [3 points]

We first train a feed-forward neural network to learn a simple 1-dimensional function. We explore the effect of the number of neurons on network performance, look at the variation of error as the network learns, and visualize the function the network learns.

The dataset for this task is in the file `data.json`. The file `nn_regression_main.py` contains code for loading the data and running the functions corresponding to each section of this task. This file doesn't need to be modified. The file `nn_regression.py` contains one function for each section of this task (and one function to calculate error). This is where you add your code to implement required functionality. The file `nn_regression_main.py` contains various functions for plotting.

INOTE In this exercise we use the scikit class `MLPRegressor`, if not specified the regressor has to be used with the solver 'lbfgs', for 200 iterations with the regularization `alpha=0`, the logistic function as activation function, and $n_h = 8$ hidden neurons on a single hidden layer. Use the `hidden_layer_sizes` parameter to set the hidden layer size, `solver` to set the training solver, `alpha` to set the regularization, `activation` with value 'logistic' to set the activation function, and `max_iter` to set the number of iterations. The `hidden_layer_sizes` is a tuple of length equal to the number of hidden layers, and each element contains the number of hidden neurons in that layer. So for example, for a network with 1 hidden layer containing 8 neurons, you would pass in `hidden_layer_sizes=(8,)`. You can use the `random_state` argument to set the random seed for the initialization of weights.

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.

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

- 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?

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. Initialize 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 ?
- Include plots of how the MSE varies with the number of hidden neurons.
- Interpret and discuss your results in the context of over/under fitting.

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:

- Include the plot of the variations of the MSE with three different number of hidden neurons for each solver.
- Is the risk of overfitting increasing or decreasing 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 in real-life problems, according to your analysis which solver will be more appropriate when the number of neurons increases ?

1.2 Regularized Neural Networks [4 points]

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:

$$\text{msereg} = \text{mse} + \frac{\alpha}{2n} \sum_i w_i^2$$

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 .
- What is the best value of α ?
- Is regularization used to overcome overfitting or underfitting ? Why ?

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 is 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 `plotBars_early_stopping_mse_comparison` in `nn_regression_plot.py` to plot bar plots 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.
- 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?
- 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?

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).
- 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.

2 Face Recognition with Neural Networks [9 points]

For this task, use the `MLPClassifier` class and the '`tanh`' activation function for the hidden layer(s). Leave all the other parameters to their default values. The output layer uses an identity activation function by default and the loss function used is cross-entropy, which, for the binary classification case, is formulated as:

$$Loss(\hat{y}, y, W) = -y \ln \hat{y} - (1 - y) \ln(1 - \hat{y}) + \frac{\alpha}{2n} \|W\|_2^2$$

where \hat{y} is the predicted value, y is the actual value, α is the regularization parameter and W is the weight matrix. `MLPClassifier` uses soft-max across the output neurons to do multi-class classification..

The data file `faces.json` contains face images. The dataset contains images of different persons, with different poses (straight/left/right/up), with/without sunglasses and showing different emotions. It contains 2 datasets: `dataset1` (`input1`, `target1`) with 60 data points and `dataset2` (`input2`, `target2`) with 564 data points. The `input` matrices contain $32\text{px} \times 30\text{px}$ images and the `target` matrices contain the class information – the first column codes the person, the second column the pose, the third column the emotion and the last column indicates whether the person is wearing sunglasses.

Before training the network, the data has to be normalized so that each sample has unit norm. This is done so that even if there are changes in illumination in the photos, only the relative values matter. The data is normalized using scikit-learn's `normalize` function ([documentation](#)).

The file `nn_classification_main.py` contains code for loading and normalizing the data and running the functions corresponding to each section of this task. This file doesn't need to be modified. The file `nn_classification.py` contains one function for each section of this task. This is where you add your code to implement required functionality. The file `nn_classification_main.py` contains various functions for plotting.

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.

INOTE Use scikit-learn's `confusion_matrix` function to calculate the confusion matrix. Documentation for this can be found [here](#)

INOTE You can use the `coefs_` attribute of the model to read the weights. It is a list of length $n_{layers} - 1$ where the i th element in the list represents the weight matrix corresponding to layer i .

INOTE Use the `plot_hidden_layer_weights` in `nn_classification_plot.py` to plot the hidden weights.

In your report:

- Include the confusion matrix you obtain and discuss. Are there any poses which can be better separated than others?
- Can you find particular regions of the images which get more weights than others?
- Include all plots in your report.

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.

INOTE Use the `random_state` parameter of `MLPClassifier` to pass in different random seeds to get different initial weights.

INOTE Use the `plot_histogram_of_acc` in `nn_classification_plot.py` to plot the histogram of accuracies.

INOTE Use the `plot_image` in `nn_classification_plot.py` to plot the misclassified images.

In your report:

- Why do different networks have different accuracies? Explain the variance in the results.
- Do the misclassified images have anything in common?
- Include all plots in your report.

3 Optional: Implementation of a perceptron [4* points]

In this task, we'll implement a perceptron and use the perceptron training rule to train the perceptron to do classification. A schematic of a perceptron is shown in figure 1.

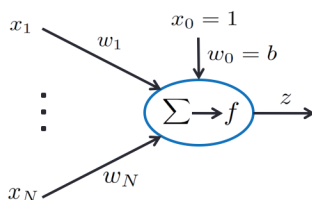


Figure 1: Schematic of a perceptron

Where $a = w^T x$ and $z = f(a)$. We'll use the heaviside step function for f : $f(a) = 0$ if $a < 0$ and $f(a) = 1$ if $a \geq 0$.

The perceptron training rule is as follows:

1. Initialize the weights to zero or random values
2. For each iteration:
 - For each sample
 - If sample is classified correctly, don't change the weights. i.e. if $z = 0$ and $y^{(i)} = 0$ or if $z = 1$ and $y^{(i)} = 1$.
 - Otherwise update the weights according to

$$\mathbf{w} := \mathbf{w} + \eta(y^{(i)} - z)x^{(i)}$$

- Stop when either max iterations reached, or all samples are classified correctly.

Now, implement the perceptron and perceptron training rule:

- In the method `_fit` of class `Perceptron` in file `perceptron.py` write code to train a perceptron according to the above algorithm given a training set of samples and classes.

- In the method `_predict` of class `Perceptron` in file `perceptron.py` write code to predict the classes of the given samples
- In the function `main` in file `perceptron.py` train and test your implementation of the perceptron using the data loaded with function `load_data`. Calculate the MSE and classification error. Also plot the decision boundary learnt. Try this with different learning rates and number of iterations.
- Repeat the above for the data that's not linearly separable loaded using the `load_non_linearly_separable_data` function.
- Repeat for both data sets, but using scikit-learn's implementation of the perceptron learning ([documentation](#)).

INOTE You can plot the dataset, and decision boundary using functions `plot_data` and `plot_decision_boundary` respectively.

INOTE To calculate MSE and classification errors you can either use scikit learn provided functions or the ones you implemented for the previous sections.

INOTE To be able to use both your implementation of the `Perceptron` class, and scikit-learn's `Perceptron`, you can import the scikit-learn version under a different name with `from sklearn.linear_model import Perceptron as SkPerceptron` and using the `SkPerceptron` class whenever you want to use scikit-learn's implementation.

- In your report:
 - Include plots of the decision boundaries learnt for the two datasets, two implementations, and different values of learning rates and number of iterations and briefly explain/discuss the graphs.
 - How do the results of scikit-learn's implementation of Perceptron differ from your implementation?
 - How many training iterations does it take for the perceptron to learn to classify all training samples perfectly?
 - What is the misclassification rate for both datasets?
 - How would you learn a non-linear decision boundary using a single perceptron? (recall the solutions in HW1)