Computational Intelligence SS19
Homework 2
Neural Networks

Anand Subramoney, Guillaume Bellec

Tutor:	Sofiane Correa de Sa, correadesa@student.tugraz.at
Points to achieve:	16 pts
Extra points:	4* pts
Info hour:	12.04.2019 09:00 - 10:00, IGI Seminar Room
	30.04.2019 13:00 - 14:00, i11
Deadline:	02.05.2019 23:55
Hand-in procedure:	Use the **cover sheet** available in TeachCenter.
	Submit your **python files and pdf report** in TeachCenter.
Course info:	https://tc.tugraz.at/main/course/view.php?id=642
Newsgroup:	tu-graz.lv.ci

Contents

General remarks

Your submission will be graded based on:

- Correctness (Is your code doing what it should be doing? Is your derivation correct?)
- The depth of your interpretations (Usually, only a couple of lines are needed)
- The quality of your plots (Is everything clearly visible in the print-out? Are axes labeled? . . .)
- Your submission should run with Python 3.3+
- 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. 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 [9 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

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_plot.py` contains various functions for plotting.

INOTE In this exercise we use the scikit class `MPLRegressor`, if not specified the regressor has to be used with the solver 'lbfgs', for 5000 iterations with the regularization `alpha=0`, the logistic function as activation function. 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 `fit`, and compute the output predicted on the testing set using `predict`.
- Plot the learned functions for $n_h = 2, 5, 50$ using the test dataset. Use the function `plot_learned_function` in `nn_regression_plot.py` for the plot.
- Repeat this a few times, since the results will vary every run when the random seed is not fixed. Include results in your report that you think are demonstrative of a "typical" case.

In your report:

- Include demonstrative 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 function `calculate_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 for $n_h = 5$. 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:

- What is the minimum, maximum, mean and standard deviation of the mean square error obtained on the training set?
- Is the minimum 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 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 $n_h \in \{1, 2, 3, 4, 6, 8, 12, 20, 40\}$ hidden neurons on one layer.
- 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 a model that uses the best value of n_h. 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 independent of the choice of the random seed (use errors that are averaged over runs with different random seeds)?
- 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) **Change of MSE during the course of training**:

In the function `ex_1_1_d` in file `nn_regression.py`:

- Write code to train a neural network with $n_h \in \{2, 5, 50\}$ 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'. 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 how the MSE varies during the course of training with the three different number of hidden neurons for each solver.
- Does the risk of overfitting increase or decrease with the number of hidden neurons?
- 'adam' is a variant of 'sgd', and both are first order methods (the parameter updates are based only on the first-order derivatives), whereas 'lbfgs' is a second order method (the updates are also based on the second-order derivatives). Which methods 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?

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 and set `max_iter` to 5000 unless specified otherwise.

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 $n_h = 50$ hidden neurons with values of alpha $\alpha \in \{10^{-8}, 10^{-7}, 10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1, 10, 100\}$. Stack your results in an array where the first axis correspond to the regularization parameter and the second to the number of random seeds. Run each training with 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/are the best value(s) 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.

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 two-thirds of your old training set become your new training set and the rest is your validation set. 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 $n_h = 50$ and $\alpha = 0$ on each selection of the training set. Train using the 'lbfgs' solver for 10 different random seeds and monitor the error on each set every 20 iterations. Run the training for a total of $100,000$ 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.
- 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 [7 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 32px × 30px 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 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 ith 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: Back-propagation with weight sharing [4* points]

In question 2, all faces were centered within the image. For another dataset shown in Figure 1 we also want to perform face recogntion but the faces are nor scaled nor centered. This raises two problems: (a) the image is much larger. So with one weight connecting each pixel to each hidden units the number of parameters gets too large; (b) if we learn to recognize a shape/feature in a location, we want to be able to detect it anywhere else in the image. To solve these two problems, we structure the neural network in a particular manner using restricted "receptive fields" and "weight sharing".

We define our network and notations as in Figure 1. For simplicity we consider only one output neuron. The conventions we use are: i is the index over the hidden neurons and j is a single index pointing to the pixel number. W of size (N_{pixels}, K_{hidden}) are the weights to the hidden layer and w of size (K_{hidden}) are the weights to the output, f_h and f_{out} are the activation functions of respectively the hidden and the output layers.

The equations that define the network model are therefore:

$$a_h^i = \sum_j W_{ij} z_{in}^j \text{ activation of hidden neuron } i \quad (1)$$

$$z_h^i = f_h^i(a_h^i) \quad (2)$$

$$a_{out} = \sum_i w_i z_h^i \text{ activation output neuron} \quad (3)$$

$$z_{out} = f_{out}(a_{out}) \quad (4)$$

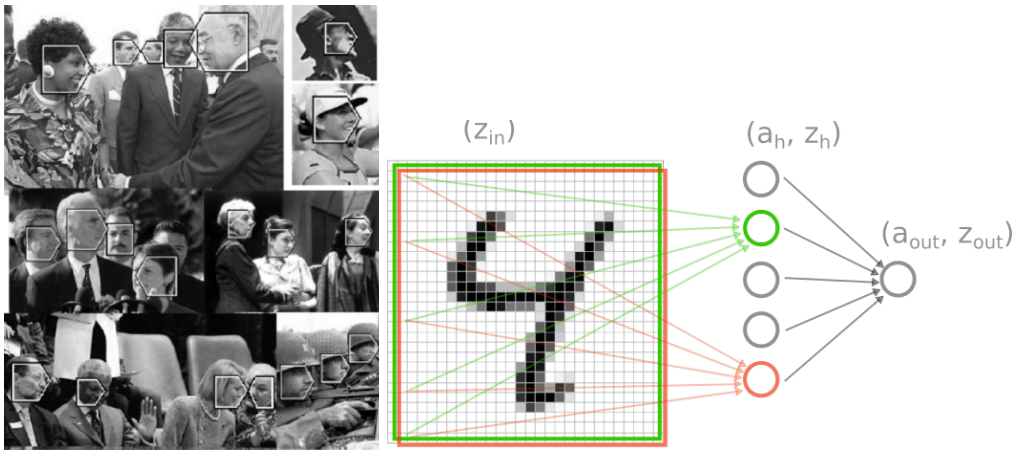

Figure 1: **Recognition of non-centered faces with a two layer neural network. Left** Examples of dataset for face recognition where faces are not centered. Compared to question 2, this problem is much harder as one need first to detect where the faces are, and then to recognize them. **Right** Neural network architecture without restricted receptive fields and weight sharing. The grey level of the input pixels directly encode z_{in}, the hidden layers apply the non-linear activation function f_h to the weighted inputs $a_h = W z_{in}$ to transmit information to the output layer: $z_h = f_h(a_h)$. The cost function finally compares z_{out} to some labels using an activation function f_{out} to process the weighted outputs of the hidden layer $z_{out} = f_{out}(a_{out})$ with $a_{out} = w z_h$.

The cost function J is not specified but we know it is a function of only z_{out}. For instance the mean square error could be used to perform regression or the cost of logistic regression for doing classification. We will compute, as in back-propagation, the gradient of the error function with respect to weight W_{ij} or w_i: $\frac{\partial J}{\partial W_{ij}}$ and $\frac{\partial J}{\partial w_i}$. Because of the chain rule we have $\frac{\partial J}{\partial W_{ij}} = \frac{\partial J}{\partial z_{out}} \frac{\partial z_{out}}{\partial W_{ij}}$. We assume that we know $\frac{\partial J}{\partial z_{out}}$ and we focus, in the rest of the exercise, on the derivation of $\frac{\partial z_{out}}{\partial W_{ij}}$ with and without weight sharing.

a) As in back-propagation, use the chain-rule to derive the following two equalities: (Include a scan of your derivations on paper in the online report, or write the formula with latex).

$$\frac{\partial z_{out}}{\partial w_i} = f'(a_{out}) z_h^i \tag{5}$$

$$\frac{\partial z_{out}}{\partial W_{ij}} = f'(a_{out}) w_i f'(a_h^i) z_{in}^j \tag{6}$$

As explained in Figure 2, in comparison to a fully connected neural network, each hidden unit is specialised for a specific location in the image. The specific location of each neuron is called its "receptive field": basically the sub-window that it is looking at. Formally it means that only the pixels within the receptive fields of neuron i have non-zero weights going toward i. To cover the full space each neuron i specializes for a different location. Using this strategy the number of parameters is greatly reduced.

b) If the initial image is of size $256px \times 256px$ and each receptive field is of size $32px \times 32px$, for an arbitrary number of hidden neurons K_{hidden}, what is the number of potentially non-zero connections to the hidden units? Compare this to a densely connected network where all pixels project to all the hidden neurons. How many hidden neurons do we need to cover the full image if the neurons' receptive fields are not allowed to overlap?

Now, we add an additional feature that all hidden units share the same structure of input weights (see Figure 2). The weight template shared by all hidden units is denoted ϕ. If the connection from the top-left corner of the receptive field of neuron hidden neuron i is modified, the connection from the top-left corner

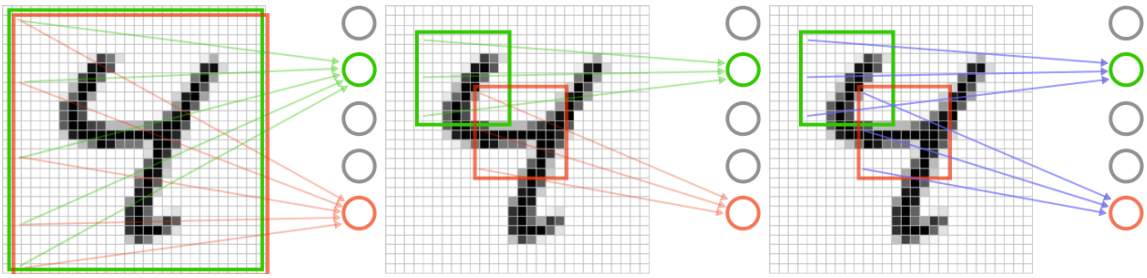

Figure 2: **Limiting the number of connections to hidden units using receptive fields and weight sharing. Left** Dummy neural network where all pixels are connected to all hidden units. **Center** Partial connection, each hidden unit has a specific receptive field (sub-window that the neuron is looking at). It is specialised for a location. The number of non-zero weights connecting pixels and hidden units is much smaller. **Right** To reduce even more the number of parameters, one can use the weight sharing strategy where all hidden unit share the same input weight but corresponding to a different location.

of another hidden neuron i' is also modified. We now use the additional index k to denote the pixel number within a receptive field. The actual pixel j to which it corresponds to in the image is given by the mapping $j = p(i, k)$. p maps a given hidden neuron number i and position k within its receptive field to the absolute pixel number j is corresponds to. We are interested in the gradient of the output neuron z_{output} with respect to a weight change in the template ϕ. Formally the use of the template means that for all hidden unit i and position k of the receptive field we have:

$$W_{i,p(i,k)} = \phi_k \tag{7}$$

c) Explain how weight sharing reduces the number of parameters further. Explain how, with weight sharing, the network can still detect faces appearing in locations different from where it was in the training set.

d.1) In two steps we will now compute the gradient $\frac{\partial z_{out}}{\partial \phi_k}$ needed to perform back-propagation with weight sharing. For simplicity we first compute the gradient of z_h^i – the output of the hidden unit. Write the gradient $\frac{\partial z_h^i}{\partial \phi_k}$ as a function of the derivative of the activation of the hidden neuron $f_h'(a_h^i)$ and the value of the input unit $z_{in}^{p(i,k)}$ connected to the hidden neuron i through the weight ϕ_k.

d.2) Using the chain rule it is now possible to access the gradient of z_{out}. Write the gradient $\frac{\partial z_{out}}{\partial \phi_k}$ as a function of $f_h'(a_h^i)$, $z_{in}^{p(i,k)}$, the derivative of the output activation $f_{out}'(a_{out})$ and the weights to the output neuron w_i. Compare, in one sentence, this gradient $\frac{\partial z_{out}}{\partial W_{ij}}$ from part a).

d.3) Given that J is now the mean square error between z_{out} and a target value z_{target}, we want to implement gradient descent to optimize the template ϕ_k, with learning rate η. Using the previous questions demonstrate that the parameter update has the form:

$$\phi_k \leftarrow \phi_k + \eta(z_{target} - z_{out}) f_{out}'(a_{out}) \sum_i w_i f_h'(a_h^i) z_{in}^{p(i,k)}$$