Computational Intelligence SS20
Homework 3
Neural Networks

Ceca Kraisnikovic
Horst Petschenig

Tutor:	Sofiane Correa de Sa, correadesa@student.tugraz.at
Points to achieve:	20 pts
Bonus points:	4* pts
Info hour:	19.05.2020 Cisco WebEx Meeting, see TeachCenter
Deadline:	26.05.2020 23:55
Hand-in procedure:	Use the **cover sheet** available in TeachCenter. Submit your **python files and pdf report** in TeachCenter.
Course info:	TeachCenter

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 3.5+.
- 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 training or testing data set gives the predictions for that data set, which you can use to calculate the training and testing errors.

1 Regression with Neural Networks [14 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 that 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` for 5000 iterations with the regularization `alpha=0` and the logistic function as activation function. Use the `hidden_layer_sizes` parameter to set the hidden layer size, `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 seed for the random number generator that samples the initial 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 seeds for $n_h = 5$. Change the random seed by passing a different value to the `random_state` argument of the neural network constructor.
  NOTE: When the seed is fixed, executing the same function again will give you the same results. Here we want to use the same 10 different seeds every time the function is called. One way to achieve this is to use the index of a for loop as the `random_state` argument.

In your report answer the following questions:

- What is the minimum, maximum, mean and standard deviation of the mean squared 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 $n_h \in \{1, 2, 4, 6, 8, 12, 20, 40\}$ hidden neurons in one layer.
- Compute the MSE over 10 different 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.
  NOTE: For every n_h use the same seeds.
- 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 in 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 learned 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. 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.

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.
- Does the risk of overfitting increase or decrease with the number of hidden neurons?
- What feature of stochastic gradient descent helps to overcome overfitting?

1.2 Regularized Neural Networks: Weight Decay

Different regularization methods for neural networks exist, such as weight decay and early stopping (covered in the lectures). Through this task we will investigate weight decay. Use the same dataset as before and set `max_iter` to 5000 unless specified otherwise.

Here, we train the network with different values of the regularization parameter α. The loss function in this case looks like this:

$$\text{MSE}_{\text{reg}} = \frac{1}{2n}\sum_{i=1}^{n}\left((\hat{y}^{(i)} - y^{(i)})^2 + \alpha\|W\|_2^2\right)$$

where α is the regularization term, and n is the number of examples.

In the function `ex_1_2` in file `nn_regression.py`:

- Write code to train a neural network with $n_h = 50$ hidden neurons with values of alpha $\alpha \in \{10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1, 10, 100\}$. Run each training with 10 different seeds. Stack your results in an array where the first axis corresponds to the regularization parameter and the second to the number of 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.

NOTE: Use the same function `calculate_mse` for calculating and plotting MSE (not MSE_{reg}). Important is, when α is not zero, the regularization term will be added to the error (cost) function, and as such optimized internally.

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?

2 Classification with Neural Networks: Fashion MNIST [6 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. To do multi-class classification, `MLPClassifier` uses soft-max across the output neurons and the loss function used is cross-entropy:

$$\text{CE}_{\text{reg}} = -\frac{1}{n}\sum_{i}^{n}\sum_{c}^{C} y_c^{(i)} \ln \hat{y}_c^{(i)} + \frac{\alpha}{2}||W||_2^2$$

where C is the number of classes, n is the number of examples, $\hat{y}_c^{(i)}$ is the predicted probability per class, $[y_1^{(i)}, \ldots, y_C^{(i)}]$ is the one-hot[1] encoded ground truth (target), α is the regularization parameter and W is the weight matrix.

In this task, we will use the `Fashion MNIST` dataset. The dataset contains images of different items of clothing. Each example is a 28×28 grayscale image, associated with a label from 10 classes (T-shirt/top, trousers/pants, pullover shirt, dress, coat, sandal, shirt, sneaker, bag, ankle boot). The train set is of shape $(60000, 784)$, and the test set of shape $(10000, 784)$. The first dimension represents the number of train/test examples, whereas the second dimension represents all the pixel values of the image (as a vector, $28 \times 28 = 784$). Each of these 784 values represents a feature. Labels (targets) contain the class information, i.e., the integer number from 0 to 9, representing different classes, in the order listed above.

The file `nn_classification_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_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.

[1] One-hot encoding assigns each element in a set a unique bit-string where only one bit is set. The bit is set at the position that corresponds to the index of the element in the set. Example: We can assign the value $[0, 1, 0]$ to the element "b" when we deal with the set $[a, b, c]$.

2.1 Fashion MNIST

In the function `ex_2_1` in file `nn_classification.py`:

- Write code to train a feed-forward neural network with 1 hidden layer containing *n_hidd* hidden units for `Fashion MNIST`. Choose *n_hidd* yourself. Set the maximum number of training iterations to 50.
- After choosing the number of hidden units *n_hidd*, repeat the process 5 times starting from a different initial weight vector (i.e., use 5 different seeds) and plot the boxplot for the resulting accuracy on the training and on the test set.
  NOTE: The accuracy is proportion of correctly classified examples 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 the weights between each input neuron and the hidden neurons to visualize what the network has learned in the first layer. Use again the best network (with maximal accuracy on 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 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 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.

INOTE Use the `plot_boxplot` in `nn_classification_plot.py` to plot the boxplot of accuracies.

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

In your report:

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

3 Bonus: Implementation of a Perceptron [4* points]

In this task, we will 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,

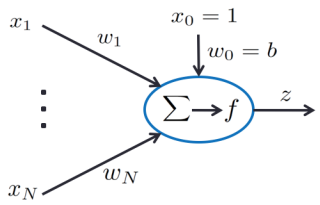

Figure 1: Schematic of a perceptron

where $a = w^T x$ and $z = f(a)$. We will 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 learned. 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 learned 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 (the percentage of incorrectly classified examples) for both datasets?
  - How would you learn a non-linear decision boundary using a single perceptron? (Hint: recall the solutions in HW2.)