Machine Learning 1, SS21
Homework 2
PCA. Neural Networks.

Ceca Kraisnikovic

Tutor:	Sofiane Correa de Sa, correadesa@student.tugraz.at
Points to achieve:	25 pts
Bonus points:	4* pts
Info hour:	will be announced via TeachCenter
Deadline:	15.05.2021 23:55
Hand-in procedure:	Use the **cover sheet** that you can find in Teach Center. Submit your **python files and a report** in Teach Center. Do not zip them. Do not upload the data folder.
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+.

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 Neural Networks [16 points]

1.1 PCA and Classification [14 points]

PCA can be used as a data preprocessing technique, to reduce the dimensionality of data. In this task, we will use the Sign Language dataset. It consists of images of hands that should be classified into 10 different classes, as shown in Fig. 1. The images are of size $(64, 64)$ pixels. That means, the input dimension to the neural network that we want to train to classify those images would be quite large ($64 * 64 = 4096$), and hence we want to reduce their dimension by means of PCA. By doing this, the benefit will be that the training time of the network will be shorter.

Tasks:

1. **PCA for dimensionality reduction.** Load the dataset (features and targets). Choose the *n_components* (number of principal components), and use *PCA* from *sklearn.decomposition* to reduce the dimensionality of the data. When creating an instance of *PCA* class, set the $svd\_solver = 'randomized'$ and $whiten = True$. In the documentation, check the methods available that you need to use. You will need to fit the model with *features* in order to obtain the model with *n_components* principal components, and apply the dimensionality reduction on *features* in order to obtain the dataset of dimension $(n\_samples, n\_components)$. In the documentation, check the Attributes, and report the percentage of variance explained. Hint: Choose *n_components* such that about 73% of variance is explained, i.e., *n_components* will be less than 100.

2. **Varying the number of hidden neurons.** We will use the data with reduced dimension (from the previous step) to train a neural network to perform classification. For this task, we will use *MLPClassifier* from *sklearn.neural_network*. Create an instance of *MLPClassifier*. Vary the number of neurons in one hidden layer: $n\_hidden \in \{10, 100, 200\}$. Set *max_iter* to 500, *solver* to *adam*, *random_state* to 1, *hidden_layer_sizes* to be *n_hid* (one of values *n_hidden*), and the other parameters should have their default values. For each *n_hid*, report accuracy on the train and test set, and the loss. How can we see if the model does not have enough capacity (the case of underfitting)? How can we see if the model starts to overfit? Does that happen with some architectures/models? (If so, say with what number of neurons that happens). Which model would you choose here and why?

3. To prevent overfitting, we could use a few approaches, for example, introduce regularization and/or early stopping. Copy the code from the previous task. Try out (a) $alpha = 1.0$ (b) $early\_stopping = True$, (c) $alpha = 1.0$ and $early\_stopping = True$. Choose (a), (b), or (c) that you think works best (write a sentence saying what was your choice). Report the train and test accuracy, and the loss for $n\_hidden \in \{10, 100, 200\}$ (for your choice). Does this reduce the overfitting? Which model would you choose now?

4. **Variability of the performance.** Choose the best performing parameters that you found in the previous task, and vary the seed (parameter *random_state*) with 5 different values of your choice. (**Note:** When the seed is fixed, we can reproduce our results, and we avoid getting different results for every run.) When changing the seed, what exactly changes? Report minimum and maximum accuracy, and mean accuracy over 5 different runs and standard deviation i.e., (mean $\pm$ std).

5. Plot the loss curve (loss over iteration). Hint: Check the Attributes of the classifier.

6. Calculate predictions on the test set (using the model with any seed of your choice). In the code, *print* the *classification_report* and *confusion_matrix*, and include either a screenshot of those two, or copy the values to the report. How could you get *recall* from *support* and $confusion\_matrix$? Explain in words what is recall. What is the most misclassified image? (Just state which class (digit) it was.)

1.2 Model selection using GridSearch [2 points]

Finding the best performing model can be cumbersome. We can use, for example, *GridSearchCV* to find the best architecture, by trying out all different combinations.

Tasks:

1. We want to check all possible combinations of the parameters:
   - $\alpha \in \{0.0, 0.001, 1.0, 10.0\}$
   - $learning\_rate\_init \in \{0.001, 0.02\}$
   - $solver \in \{lbfgs, adam\}$
   - $hidden\_layer\_sizes \in \{(50,), (100,)\}$

   Create a dictionary of these parameters that *GridSearch* from *sklearn.model_selection* requires. How many different architectures will be checked? (State the number of architectures that will be checked.)
2. Set $max\_iter = 500, activation = 'logistic', random\_state = 1, early\_stopping = True$ as default parameters of *MLPClassifier*.
3. What was the best score obtained? (Hint: Check the Attributes of the classifier.)
4. What was the best parameter set? (Hint: Check the Attributes of the classifier.)

2 Regression with Neural Networks [9 points]

Neural Networks can be used for regression problems as well. In this task, we will train a neural network to approximate a function.

Tasks:

1. Load the dataset. Since we have 3-dimensional data and it is possible to visualize it, visualize the data points. Include the figure in your report.
2. Implement the function *calculate_mse*.
3. Train the network to solve the task (use *MLPRegressor* from *sklearn*). You can use either random search or grid search to find a good model. Vary at least 3 different numbers of neurons.

   If you use random search: Describe how you chose the final model - how many neurons you tried out, one or two layers, which optimizer, was it necessary to use early stopping (and if so, what was the percentage of validation set used), which activation function did you use for neurons (*hidden_layer_sizes*).

   If you use grid search: Report which dictionary of parameters you used, and what was the best model. You have to try out AT LEAST different number of neurons, different optimizers, and some for of regularization. (In total there should be AT LEAST 8 different combinations that were checked by GridSearch.)
4. (For the final choice of the model) What was the final loss? Report the final loss achieved. Here we are minimizing the MSE. What would be "the best" loss to achieve in this case? (Hint: When the accuracy, sometimes also called success rate, is reported, it is usually given in percentages. There, the higher the better. When the error rate, or misclassification rate is reported, also given in percentages, the lower the better. However, when the loss is reported, we have to think about the mathematical form of the loss function and conclude what is the minimum of such a function. What is the minimum value of the MSE cost function?)

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 Fig. 2. From there, $a = w^T x$ 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 *sklearn* implementation of the perceptron learning (documentation).

  NOTES:

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

  To calculate MSE and classification errors you can either use *sklearn* functions or the ones you implemented yourself.

  To be able to use both your implementation of the `Perceptron` class, and *sklearn* `Perceptron`, you can import the *sklearn* version under a different name e.g.,
  `from sklearn.linear_model import Perceptron as SkPerceptron` and use the `SkPerceptron` class whenever you want to use *sklearn* 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 *sklearn* 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 what we had for linear regression.)

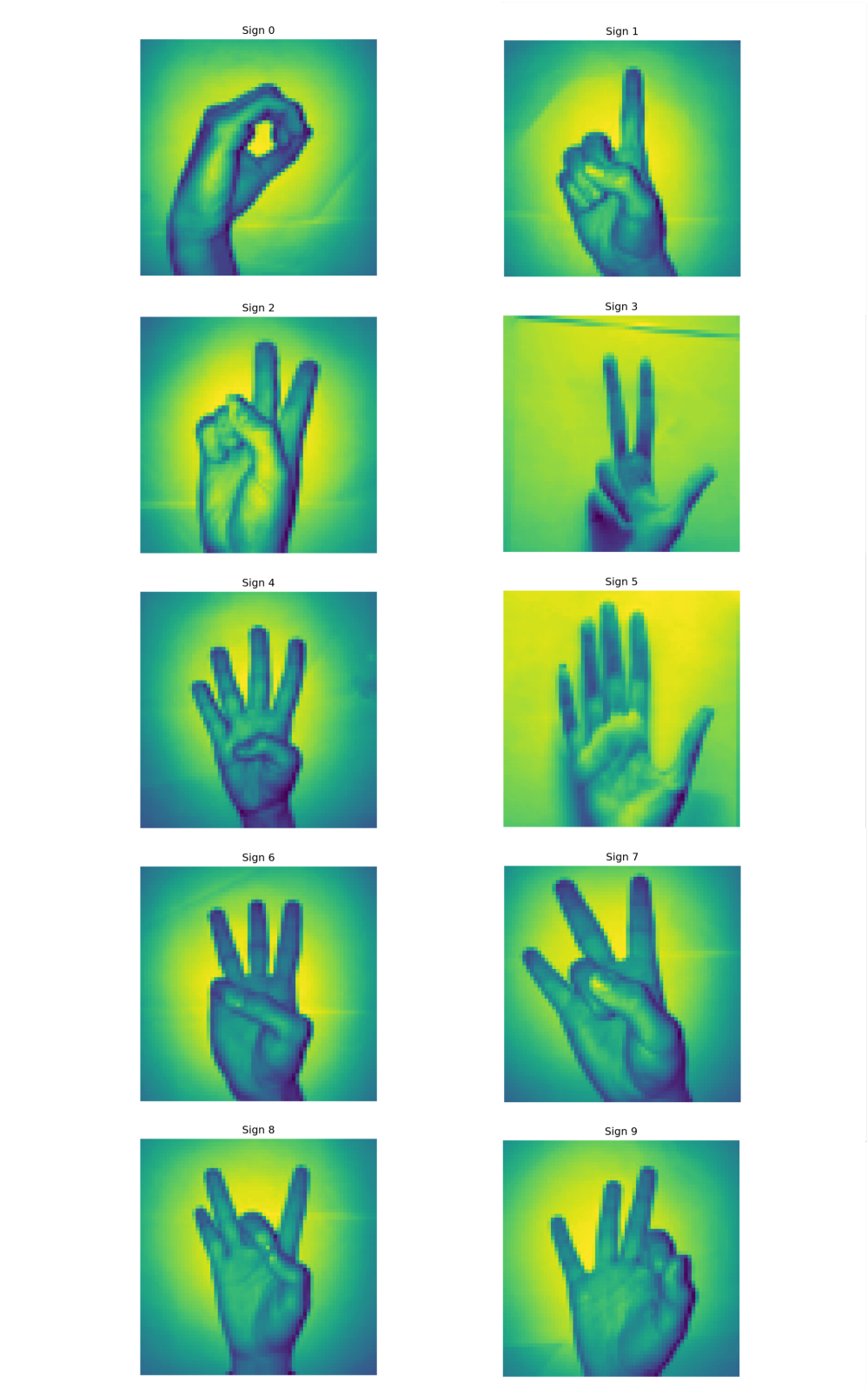

Figure 1: **A few examples for images from Sign Language dataset.**

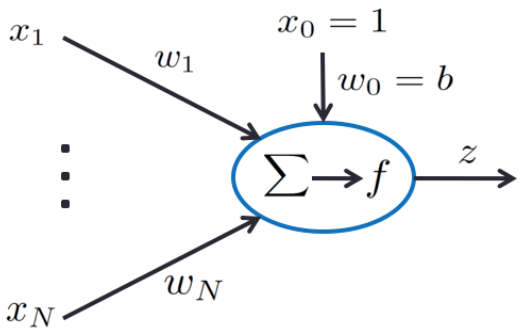

Figure 2: **Schematic of a perceptron**