Machine Learning 1, SS23
Homework 2
PCA. Neural Networks.

Ceca Kraišniković, ceca@igi.tugraz.at

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:	19.05.2023 23:55
Hand-in procedure:	Use the **cover sheet** that you can find in the Teach Center. Submit your **python files and a report** to the Teach Center. Do not zip them. Do not upload the data folder.
Submissions after the deadline:	Each missed day brings a (-5) points penalty.
Plagiarism:	If detected, 0 points for all parties involved. If this happens twice, the course graded as failed.
Course info:	TeachCenter, https://tc.tugraz.at/main/course/view.php?id=1648

Contents

1 Neural Networks [17 points] **2**
1.1 PCA and Classification [14 points] 2
1.2 Model selection using *GridSearch* [3 points] 3

2 Regression with Neural Networks [8 points] **3**

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

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 readable/interpretable? Are axes labeled? . . .)
- Your submission should run with Python 3.5+.

For this assignment, we will use 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 [17 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, i.e., the input dimension to the neural network that we want to train to classify the images would be quite large ($64 \cdot 64 = 4096$), 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.** Use *PCA* from *sklearn.decomposition* to reduce the dimensionality of the data. Create an instance of *PCA* class, and set *random_state* to 1. Choose *n_components* (number of principal components) such that about 95% of variance is explained. In the report, state *n_components* that you used, and the exact percentage of variance explained that you got.

   Hints: You will need to fit the model, and apply the dimensionality reduction of the original *features*, in order to obtain the data with a reduced dimension: (*n_samples*, *n_components*). Check the Attributes of this class, and find out which one to use to get the percentage of variance explained.

2. **Varying the number of hidden neurons.** We will use the data with the reduced dimension (from the previous step) to train a neural network to perform classification. We will vary the number of neurons in one hidden layer of the neural network: $n\_hidden \in \{2, 10, 100, 200\}$.

   For this task, we will use *MLPClassifier* from *sklearn.neural_network*. Create an instance of *MLPClassifier*. Set *max_iter* to 500, *solver* to *adam*, *random_state* to 1, *hidden_layer_sizes* to be *n_hid* (one of the values in *n_hidden*), and the other parameters should have their default values. (If the warning about the optimization not converging appears, you can ignore it. No need to do anything about it.)

   For each *n_hid*, report the accuracy on the train and test set, and the loss.

   Answer the questions: How do we know (in general) if the model does not have enough capacity (the case of underfitting)? How do we know (in general) 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?

   (Update on May 8, 2023 is marked in blue) Hint: To answer these questions properly, you may need to implement 1.1.3 first, i.e., the task that follows.

3. To prevent overfitting, we could use a few approaches, for example, introducing regularization and/or early stopping. Copy the code from the previous task. Try out (a) $alpha = 0.1$ (b) $early\_stopping = True$, (c) $alpha = 0.1$ and $early\_stopping = True$. Choose (a), (b), or (c) that you think works best (write a sentence saying what your choice was).

   Then (using a setup as in (a), (b), or (c)), report the train and test accuracy, and the loss for $n\_hidden \in \{2, 10, 100, 200\}$. Does this improve the results from the previous step? Which model would you choose now?

4. **Variability of the performance.** Choose the best-performing parameters 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 the minimum and maximum accuracy, and mean accuracy over 5 different runs and standard deviation i.e., (mean $\pm$ std).

5. *(Update on May 8, 2023 is marked in blue)* Using a model with one fixed seed that performs well (one of those used in 1.1.4), plot the loss curve (loss over iteration). Hint: Check the Attributes of the classifier.

6. *(Update on May 8, 2023 is marked in blue)* Using a model with one fixed seed that performs well (one of those used in 1.1.4), calculate predictions on the test set. In the code, *print* the *classification_report*

and *confusion_matrix*, and include either a screenshot of both of them, or copy the values to the report. How could you calculate yourself *recall* from *support* and *confusion_matrix* entries? Explain in words what is *recall*. What is the most misclassified image? State which class (digit) it was, and how you concluded that.

1.2 Model selection using *GridSearch* [3 points]

To find the best architecture, we can use, for example, *GridSearchCV*, by trying out all the different combinations of the parameters by an automated procedure.

Tasks:

1. We want to check all possible combinations of the parameters:
   - $\alpha \in \{0.0, 0.1, 1.0, 10.0\}$
   - *solver* $\in$ { *'lbfgs', 'adam'* }
   - *activation* $\in$ { *'logistic', 'relu'* }
   - *hidden_layer_sizes* $\in \{(100,), (200,)\}$

   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 and how you calculated it.)
2. Set $max\_iter = 100, random\_state = 1, learning\_rate\_init = 0.01$ as default parameters of *MLPClassifier*.
3. What was the best parameter set? What was the best score obtained (i.e., the mean cross-validation score)? Hint: Check the Attributes of the classifier.
4. Now use the best estimator to calculate the test accuracy and report it.
5. (Theoretical question, general) What is the difference between hyperparameters and parameters of a model (in general)? Explain then the difference using the example of neural networks (i.e., name a few hyperparameters and parameters of neural networks).

2 Regression with Neural Networks [8 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 (*x-datapoints.npy* are the features, *y-datapoints.npy* are the targets).
2. Implement the function *calculate_mse*. In the report, include the code snippet (or a screenshot).
3. Train the network to solve the task (use *MLPRegressor* from *sklearn*). You can perform either a manual search or a random/grid search to find a good model. Vary at least 3 different numbers of neurons in the hidden layer.

   If you use manual search: Describe how you chose the final model – e.g., 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 you used for neurons in the hidden layer(s).

   If you use random/grid search (i.e., *GridSearchCV* from *sklearn*): Report which dictionary of parameters you used, and what was the best model. You have to try out **at least** different numbers of neurons, different optimizers, and some form of regularization. (In total there should be **at least** 10 different combinations that were checked by *GridSearch*.)
4. For the final choice of the model, report the train and test loss.

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.)

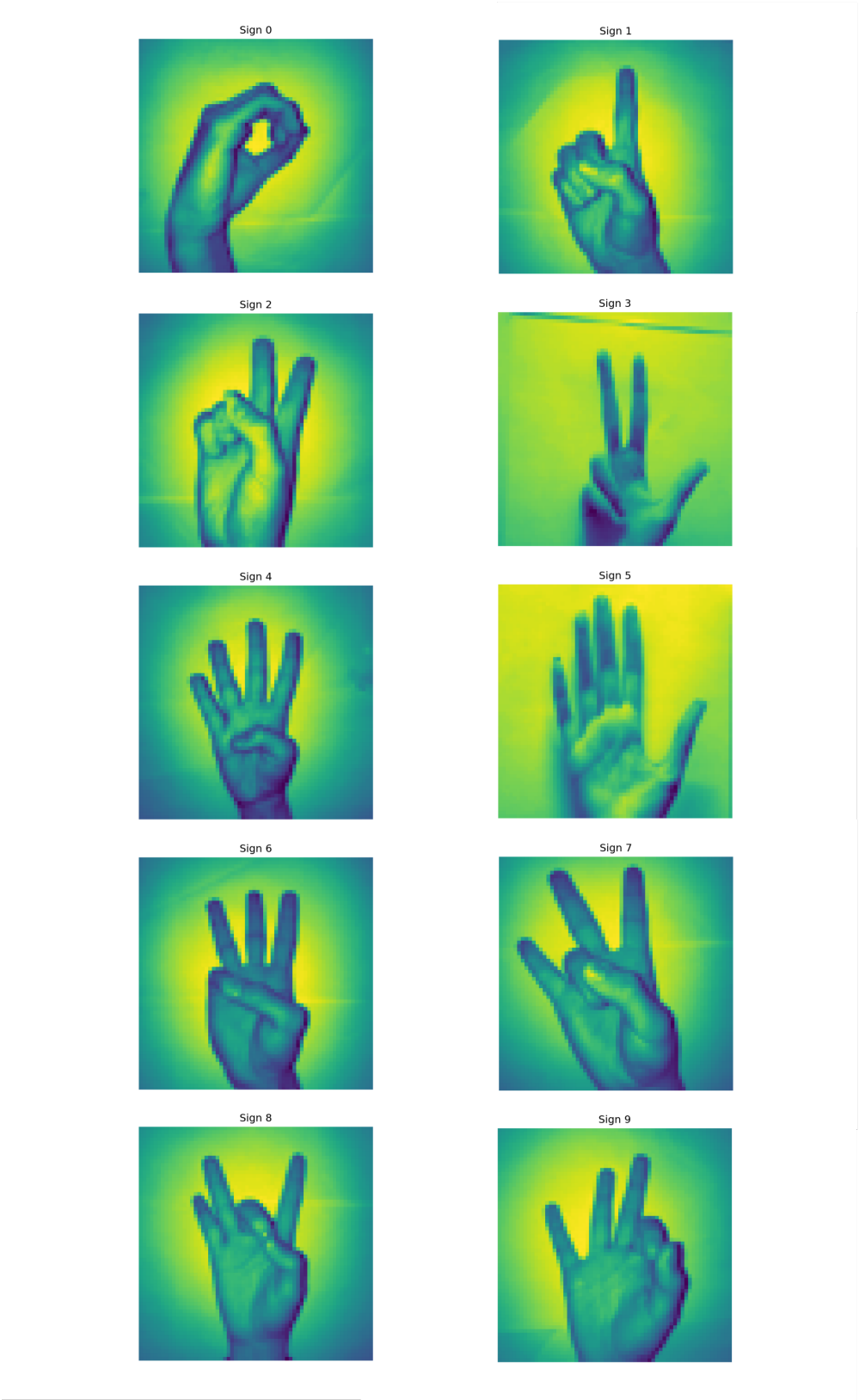

Figure 1: **A few examples for images from Sign Language dataset.**

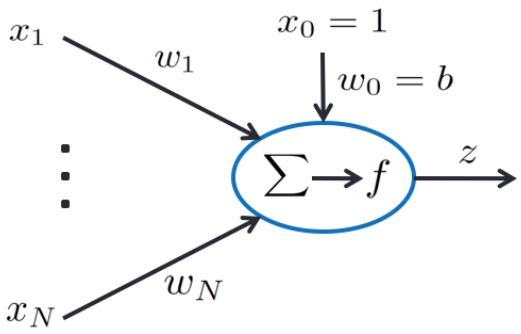

Figure 2: **Schematic of a perceptron**