# Machine Learning 1, SS24
# Homework 2
# PCA. Neural Networks.

Thomas Wedenig, thomas.wedenig@tugraz.at

| | |
|---|---|
| Tutor: | Sofiane Correa de Sa, correadesa@student.tugraz.at |
| Points to achieve: | 25 pts |
| Bonus points: | 5* pts |
| Deadline: | 31.05.2024 23:59 |
| Hand-in procedure: | Use the **cover sheet** that you can find in the TeachCenter.<br>Submit **all python files and a report (PDF)** to the TeachCenter.<br>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.<br>If this happens twice, we will grade the group with<br>"Ungültig aufgrund von Täuschung" |
| Course info: | TeachCenter, https://tc.tugraz.at/main/course/view.php?id=1648 |

## Contents



## General remarks

- **Do not add any additional `import` statements** anywhere in the code.
- **Do not modify the function signatures** of the skeleton functions (i.e., the function names and inputs).
- **Do not change the file name** of any of the Python files.

Failing to conform to these rules will lead to point deductions. Further, 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 3.9+.

# 1 Neural Networks [15 points]

In this task, we will use an implementation of Multilayer Perceptron (MLP) from `scikit-learn`. The documentation for this is available at the scikit-learn website. The relevant multi-layer perceptron class is `MLPClassifier`, as we will solve a classification task.

## 1.1 PCA and Classification [12 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 model will be smaller and thus, it will take less time to train the model.

**Tasks:**

1. **PCA for dimensionality reduction.** Use `PCA` from `sklearn.decomposition` to reduce the dimensionality of the training data `X_train`. Create an instance of `PCA` class, and set `random_state` to 42. 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.

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` $\in \{2, 10, 100, 200, 500\}$.

   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_hidden` (a value in the set above), 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 `num_hidden` $\in \{2, 10, 100, 200, 500\}$, report the accuracy on the train and validation set, and the final training loss.

3. In general: How do we know if a model does not have enough capacity (the case of underfitting)? How do we know if a model starts to overfit?

   In your particular case: Does overfitting/underfitting happen with some of your architectures/models? (If so, say with what number of neurons that happens). Which of the trained models would you perfer? Explain why.

4. **Overfitting.** 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) - depending on what you think works best. State your choice in your report.

   Then, using a setup as in (a), (b), or (c), report the train and validation accuracy, and the loss for `num_hidden` $\in \{2, 10, 100, 200, 500\}$. Does this improve the results from the previous step? Which model would you choose now?

5. From all models you have trained so far, pick the one you consider best. For this model, plot the loss curve (training loss over iterations). Hint: Check the attributes of the classifier.

## 1.2 Model selection and Evaluation Metrics [3 points]

To find a good configuration of hyperparameters, 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:

   - $\alpha \in \{0.0, 0.1, 1.0\}$
   - `solver` $\in$ { `'lbfgs', 'adam'` }
   - `hidden_layer_sizes` $\in \{(100,), (200,)\}$

   Create a dictionary of these parameters that `GridSearchCV` 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=42` as default parameters of `MLPClassifier`. Create a `GridSearchCV` object with the dictionary you have created. Run cross validation with $k = 5$ folds by setting `cv=5`. If you want a more verbose output during the search procedure, set e.g. `verbose=4`.

3. What was the best parameter set that was found in this grid search? What was the best score obtained (i.e., the mean cross-validation score)? Hint: Check the attributes of the `GridSearchCV` object.

4. Implement `show_confusion_matrix_and_classification_report`. Among all models trained so far (also the ones you have constructed without the help of `GridSearchCV`), pick the model you consider best. Then, pass this final model to the implemented function (together with the PCA-projected test set). Report the final test accuracy in your report. Also include the plot of the confusion matrix and the classification report.

5. Explain in words what *recall* measures and what *precision* measures. Which class in the test dataset was misclassified most often?

6. (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 Neural Networks From Scratch [10 points, 1* bonus point]

In this task, we will implement a neural network from scratch, i.e., without using `MLPClassifier` from `scikit-learn`. We will again solve the same classification task as above.

In order to train the network, we will use a minimal implementation of an *automatic differentiation* framework. This will allow us the compute gradients of the loss w.r.t. the weights and biases of the neural network, which in turn allows us to perform (stochastic) gradient descent.

The central component of this framework is the `Scalar` class in the `autodiff` module. An object of this class can store a single scalar value $x \in \mathbb{R}$ and, as soon as we compute derivatives of the loss $\mathcal{L}$, this object will also contain the partial derivative $\frac{\partial \mathcal{L}}{\partial x}$ at the point $x$ (which is again just a scalar value). The `Scalar` class keeps track of a computational graph by overloading methods like `__add__` or `__mul__` (which are called when you add or multiply two Python objects, respectively). Thus, we need to implement all computations we wish to differentiate using `Scalar` objects (i.e., the neural network forward pass, including all activation functions, and the loss function).

For example, if we wish to add two Scalar objects `s1 = Scalar(1.0)` and `s2 = Scalar(2.0)`, we just compute `s = s1 + s2`, where `s` is again a `Scalar` object which holds the result of the computation, as well as the computational graph that produced this result.

**Tasks:**

1. In `autodiff/neural_net.py`, implement the `__call__` method of the `Neuron` class. This method should compute the output of a single neuron, given a list of inputs. Note that the inputs are given as a list of `Scalar` objects and you should again return a `Scalar` object.

2. In `autodiff/neural_net.py`, implement the `__init__` and `__call__` method of the `FeedForwardLayer` class.

3. In `autodiff/neural_net.py`, implement the `__init__` and `__call__` method of the `MultiLayerPerceptron` class.

4. In `mlp_classifier_own.py`, you can find an implementation of a multi-class[1] classifier that uses your implementation of the MultiLayerPerceptron class under the hood. Carefully read and understand the code in this file. Note that when the MLP in `self.model` performs a forward pass, we get the *logits* at the output layer, i.e., the values that still need to be fed into the output activation function. Implement the `softmax` method and the `multiclass_cross_entropy_loss` method.

5. In `nn_classification_from_scratch.py`, implement the `train_nn_own` function. In there, create an `MLPClassifierOwn` object: We want to train for 5 epochs, set the L2 regularization coefficient $\alpha = 0$, construct a network with a single hidden layer with 16 neurons and set `random_state=42`. As input to this small network we will use the PCA-projected *Sign Language* dataset – however, this time, we will only use 16 principle components for the PCA projection (`n_components=16`). We take these measures to keep the network size small, as our implementation is highly inefficient (no parallelization/vectorization). After training, use the `score` method of the classifier to compute the accuracy on the train, validation and test set. Report the final train, validation and test accuracy in your report.

**Bonus tasks [1* bonus point]:** Right now, we can only train our network without L2 regularization ($\alpha = 0$). Implement the `l2_regularization_term` method in `mlp_classifier_own.py`. The output of this method is added to the loss during training. Specifically, let $\Omega(\boldsymbol{\theta})$ denote the output of this method. Then we have

$$\Omega(\boldsymbol{\theta}) = \frac{\alpha}{2|\mathcal{B}|} \|\boldsymbol{\theta}\|_2^2$$

where $\boldsymbol{\theta}$ is the vector of network parameters and $|\mathcal{B}|$ is the batch size.

Train the same network in Task 2, but now set $\alpha > 0$. Pick two different values for $\alpha$ and see if you can observe any difference to the case where $\alpha = 0$. State which value of $\alpha$ you have tried. Do you think that L2 regularization is useful in this case?

[1] We will extend this to also work with binary classification in Task 3, so you can ignore any code that is run when `self.num_classes == 2` for now.

# 3 Bonus: Binary Classification Extension [4* bonus points]

In this bonus task, we wish to extend `MLPClassifierOwn` to not only support multi-class classification, but also *binary classification* (i.e., `num_classes == 2`).

**Tasks:**

1. In `mlp_classifier_own.py`, implement the methods `sigmoid` and `binary_cross_entropy_loss`.
2. The code skeleton will create a binary classification dataset by slicing out the subset of the Sign Language dataset where the label is 0 or 1 (neglecting all other classes). In `nn_classification_from_scratch.py`, use the `train_nn_own` function from the previous task to train the model on this subset of data. After training, use the `score` method of the classifier to compute the accuracy on the train and test set. Report the final train, validation and test accuracy in your report.
3. (Theory Question) Your friend comes to you and says: "I have changed the labels in the Sign Language dataset in the following way: I leave every image with class 0 untouched, but I change the label of every image with a class id $\in \{1, 2, \ldots, 9\}$ to class 1. Using this new dataset with binary labels, I have trained a binary classifier, which achieves a test accuracy of $\approx 90\%$. That's great, don't you think?" Is accuracy a misleading metric in this case? Explain why/why not. If you think that it is misleading, explain which performance metric is more useful in this case.

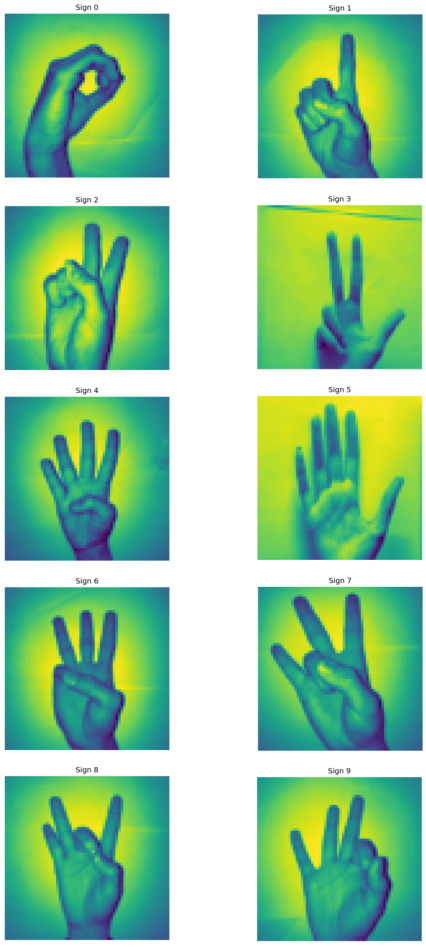


Figure 1: Example image for each class of the Sign Language dataset.