Exercise 2-1  The ADALINE learning rule
The adaptive linear element (ADALINE) model uses the least mean square cost function

\[
\text{cost} = \frac{1}{2} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 ,
\]

for \( N \) training set elements, where \( y_i \) is the actual and \( \hat{y}_i \) the computed class label of pattern \( i \). In contrast to the simple perceptron, classification is not realized by the signum-function. Instead, it is done directly: \( \hat{y} = h \).

(As a reminder: \( M \) is the number of input features of patterns \( x_i \in \mathbb{R}^M \) and the dimensionality of the weight vector \( w \in \mathbb{R}^M \), where \( x_0 = 1 \) is constant and corresponds to the bias or offset.)

a) Deduce the gradient descent-based learning rule (or: adaption rule) for the ADALINE process (analogously to the perceptron learning rule).

b) Specify the corresponding sample-based learning rule.

c) What advantages do sample-based learning rules have?

d) Name the most distinctive characteristics between the ADALINE model and the perceptron model.
Possible Solution:

a) First we take the derivative of the cost function with respect to each \( w_j \). Therefore, we first define the cost function:

\[
\text{cost} = \frac{1}{2} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 = \frac{1}{2} \sum_{i=1}^{N} \left( y_i - \sum_{j=0}^{M-1} w_{j,i} \right)^2
\]

Next, we can take the derivative of each \( w_j \). For the sake of simplicity, we denote them as \( w_{j,*} \).

\[
\frac{\partial \text{cost}}{\partial w_{j,*}} = \frac{1}{2} \sum_{i=1}^{N} 2 \cdot \left( y_i - \sum_{j=0}^{M-1} w_{j,i} \right) \cdot \frac{\partial}{\partial w_{j,*}} \left( y_i - \sum_{j=0}^{M-1} w_{j,i} \right) =
\]

\[
= \sum_{i=1}^{N} \left( y_i - \sum_{j=0}^{M-1} w_{j,i} \right) \cdot (-x_{i,j,*}) =
\]

\[
= \sum_{i=1}^{N} x_{i,j,*} \cdot \left( \sum_{j=0}^{M-1} w_{j,i} \right) - y_i = \sum_{i=1}^{N} x_{i,j,*} (\hat{y}_i - y_i)
\]

Now, we can define the learning rule for the gradient descent as:

\[
w_j \leftarrow w_j - \eta \frac{\partial \text{cost}}{\partial w_j} = w_j + \eta \sum_{i=1}^{N} x_{i,j} (y_i - \hat{y}_i) , \quad (1)
\]

or vectorized:

\[
w \leftarrow w - \eta \frac{\partial \text{cost}}{\partial w} \quad (2)
\]

and is therefore pretty similar to the gradients of a perceptron

\[
w_j \leftarrow w_j + \eta \sum_{i \in M} x_{i,j} y_i = w_j + \eta \frac{1}{2} \sum_{i=1}^{N} x_{i,j} (y_i - \hat{y}_i) . \quad (3)
\]

b) A sample-based learning rule for a sample \( t \) can be defined as:

\[
w_j \leftarrow w_j + \eta x_{j}(t) (y(t) - \hat{y}(t)) = w_j + \eta x_{t,j} (y_t - \hat{y}_t) , \quad (4)
\]

which is also called Delta rule.

c) Sample-based learning rules can be learned on-the-fly: If an existing model is extended by a new sample, it is not necessary to recompute the whole model. This is essential for large datasets which cannot be fit into memory.

d) Striking difference: objective function. Consequences:

- Perceptron generates binary output while ADALINE outputs and optimizes vector of reals.
- If the data is separable, Perceptron converges faster than ADALINE.
- ADALINE is used to approximate the separating hyperplane more than it is used to classify (this could more easily be achieved using the Perceptron).
- the Perceptron uses class labels to learn model coefficients.
- ADALINE uses continuous predicted values to learn model coefficients which is 'more' powerful since it tells us by 'how much' we were right or wrong.
Exercise 2-2  Regularization / Overfitting

a) What is overfitting and how does it occur?

b) How can a model be identified as “overfitted”?

c) How can overfitting be avoided?
Possible Solution:

a) In general ‘Overfitting’: Over-adaptation of a model to a given dataset. We will look at the overfitting problem from three different point of views:

   (a) **Model’s complexity.** Complex models - such as DNNs - can detect subtle patterns in the training data. If the dataset is noisy, or if it is too small (introducing sampling noise) then the model is likely to detect patterns in the noise itself. Trivially, these patterns will not generalize to new instances.

   (b) **# of Training epochs.** With increasing number of epochs, the algorithm learns and is its prediction error on the training set naturally goes down, and so does the prediction error on the validation set. However, we can observe that the validation error stops decreasing and start to go back up again. This indicates that the model has started to overfit the training data. *(Hint: early stopping)*

   (c) **Bias-Variance Trade-off.** One can also approach the problem via the bias-variance trade-off (not discussed in detail in the lecture, therefore one can omit this): A model’s generalization error can be expressed as the sum of three different errors: **Bias.** This error is due to wrong assumptions, such as assuming that the data is linear when it is actually quadratic, respectively, constraining the complexity of the model too much. A high-bias model is most likely to underfit the training data. **Variance.** This error is due to the model’s excessive sensitivity to small variations in the training data, i.e, a model with many degrees of freedom (such sa a high-degree polynomial model) is likely to have a high variance and thus is most likely to overfit the training data. **Irreducible error.** This error is due to the noisiness of the data itself. The only way to tackle this problem is to do some pre-processing and clean up the data (remove null values, etc.) In summary: Increasing a model’s complexity will typically increase its variance and reduce its bias. Conversely, reducing the model’s complexity increases its bias and reduces its variance. This trade-off is widely known as the **Bias-Variance trade-off**)

b)

- \( M \): Number of model variables, \( N \): Number of observations (samples); \( M \approx N \) instead of \( M \ll N \), i.e. too many explaining variables/regressors are included into the model.
- Unstable estimator \( \hat{w}_{LS} \), i.e. small perturbations in the data result in relatively large changes in the components of the estimator. Changes could be different or additional observations.

c)

- On the error-function, a penalty/regularization-term is added, which penalizes the number of model parameters. 
  \[
  J^K_{PE}N(w) = \sum_{i=1}^{N} (y_i - f(x_i, w))^2 \Rightarrow 
  \sum_{i=1}^{N} (y_i - f(x_i, w))^2 + \lambda \sum_{i=0}^{M-1} w_i^2
  \]
  Choice of \( \lambda \) depends on domain-knowledge/experience
- \( N >> M \)
Exercise 2-3  Basis Functions of Neural Networks

Given a test vector $x_i$, the output of a neural network is defined as

$$f(x_i) = \sum_{h=0}^{M-1} w_h \phi_h(x_i, v_h).$$

The weights of the neurons can be learned by employing the back-propagation rule with sample-based gradient descent. In the lecture neural networks with sigmoid neurons have been introduced, but it is possible to employ different basis functions:

a) Which properties do these basis functions have to fulfill?

b) Can a linear combination $\phi(x_i, v_h) = z_h = \sum_{j=0}^{M} v_{h,j} x_{i,j}$ be suitable for this?

c) Is the number of parameters for $\phi(x_i, v_h)$ limited? Could several different basis functions be used for the same neural network?
Possible Solution:

a) The basis function of a neural network should be **continuous** and **partial differentiable** (side note: it is sufficient to be almost everywhere partial differentiable). Furthermore, it comes in handy if the amount of inflection points as well as maxima and minima are minimized (for the case that one does not want to learn a periodic function). The approach by applying a logistic function resembles the functionality of cerebral interconnections. There we have some stable regions which change their value only on a given range rapidly (and not for the entire domain of a function).

b) The linear combination is only suitable for single-layered NNs. If the setup of the NN contains linear neurons in serial, then the neurons neutralize one each other:

\[
\begin{align*}
    f(x_i) &= \sum_{h=0}^{M_g-1} \sum_{j=0}^{M} w_{h} \phi(x_i, v_h) = \sum_{h=0}^{M_g-1} w_{h} \sum_{j=0}^{M} v_{h,j} x_{i,j} = \\
    &= \sum_{h=0}^{M_g-1} \sum_{j=0}^{M} w_{h} v_{h,j} x_{i,j} = \sum_{j=0}^{M} \left( \sum_{h=0}^{M_g-1} w_{h} v_{h,j} \right) x_{i,j} = \sum_{j=0}^{M} u_j x_{i,j}
\end{align*}
\]

The effect is the same as the one of a simple perceptron.

c) The amount of parameters for \( \phi(x_i, v_h) \) are not limited - one can see that if we take the radial basis function (RBF) into account:

\[
\phi(x_{i,j}, \mu_j, \sigma_j) = \exp\left(-\frac{\|x_{i,j} - \mu_j\|^2}{2\sigma_j^2}\right),
\]

which are typically used as a first layer in a 2-layer RBF network in order to propagate the input signal. In the second layer, one can use a linear combination of the signals coming from the first layer. In total this architecture is similar to a gaussian kernel density estimator. Of course it is possible to combine basis functions in a more complex way.

**Exercise 2-4**  Tensorflow Introduction

With the provided Jupyter Notebook, we want to give you a short introduction in Tensorflow and some insights in the basic usage of some common function calls.

**Exercise 2-5**  Train-Test Split and Regularization in Python

The provided Jupyter Notebook will guide you through the steps of implementing and applying Train-Test Splits, Cross Validation and Regularization in Python. Complete the specified tasks by filling in the missing code. You can further play around with varying parameters or trying out other models provided by sklearn.