Laboratory 10

In this laboratory we will focus on neural networks for classification.

We have seen that neural networks can be interpreted as a method to perform non-linear feature expansion through a parametric function that is able to approximate complex smooth transformations. We can then pair the neural network feature expansion with a simple classification model to estimate simple class separation rules in a transformed feature space.

Although there are several advanced libraries for neural networks, in this laboratory we will employ the simple implementation provided by the `sklearn` library. The library provides the tools required to train a simple Multi-Layer Perceptron for classification. The first step consists in importing the `sklearn` neural network library

```
import sklearn.neural_network
```

We can access the MLP implementation through the class `sklearn.neural_network.MLPClassifier`. The first step consists in creating an `MLPClassifier` instance by calling the class constructor. The class constructor accepts several parameters (which we can pass by keyword), all of which have default values. For this laboratory, the parameters of interest are

- `hidden_layer_sizes`: a tuple or list of integers. The length of the list represents the number of *hidden* layers (i.e., excluding input and output layer), whereas the i-th element of the list is the size of the corresponding hidden layer
- `activation`: the non-linear activation function of hidden nodes — `'logistic'` for sigmoid activation, `'tanh'` for hyperbolic tangent, `'relu'` for ReLU, `'identity'` for linear activation ($y = x$)
- `max_iter`: the maximum number of iterations — we will set this value to `2000` in the following
- `learning_rate`: controls the learning rate schedule for stochastic gradient descent methods (we will use the default value)
- `learning_rate_init`: initial value for the learning rate (we will use the default value)
- `batch_size`: the size of a batch for SGD-based methods (we will use the default value)
- `alpha`: L2 regularization coefficient — to preserve consistencies with the logistic regression laboratory, we should set this to $\lambda \times$ (# of samples) (see below)
- `solver`: The solver to employ — possible values are `'lbfgs', 'adam', 'sgd'`; for the laboratory part we will use `'lbfgs'`, for the project part we will use `'adam'` (see below)
- `random_state`: defines the initial seed for weights initialization. For reproducibility, we will set `random_state = 0`

The `MLPClassifier` class creates a MLP with the provided number of hidden layers. It also employs a softmax-activated output layer with a number of nodes equal to the number of classes for multiclass classification, or a single-node sigmoid-activated output layer for binary classification. The output layer is automatically chosen by the implementation based on the values of the label array we provide at training time. At this stage we therefore need only to provide the size of the hidden layers `hidden_layer_sizes`

To reduce the effects of over-fitting, the `MLPClassifier` allows for L2 regularization of the model weights, thus minimizing the regularized loss

$$\mathcal{L}(\boldsymbol{W}_1, \boldsymbol{b}_1, \dots \boldsymbol{W}_l, \boldsymbol{b}_l) = \sum_{j=1}^{l} \frac{\lambda}{2} \left\| \boldsymbol{W}_j \right\|^2 + \ell(\boldsymbol{W}_1, \boldsymbol{b}_1, \dots \boldsymbol{W}_n, \boldsymbol{b}_n)$$

where ℓ is the classification *loss*, i.e., the binary cross-entropy

$$\ell(\boldsymbol{W}_1, \boldsymbol{b}_1, \dots \boldsymbol{W}_n, \boldsymbol{b}_n) = -\sum_{i=1}^{n} \left[c_i \log y_i + (1 - c_i) \log(1 - y_i) \right] \ , \quad y_i = f_{net}(\boldsymbol{x}_i, \boldsymbol{W}_1, \boldsymbol{b}_1, \dots \boldsymbol{W}_n, \boldsymbol{b}_n)$$

with c_i the label of sample $\boldsymbol{x}_i$, or the multi-class cross-entropy

$$\ell(\boldsymbol{W}_1, \boldsymbol{b}_1, \dots \boldsymbol{W}_n, \boldsymbol{b}_n) = -\sum_{i=1}^{n}\sum_{k=1}^{K} z_{ik} \log y_{ik} \ , \quad y_i = \boldsymbol{f}_{net}(\boldsymbol{x}_i, \boldsymbol{W}_1, \boldsymbol{b}_1, \dots \boldsymbol{W}_n, \boldsymbol{b}_n)$$

for multi-class problems, with $\boldsymbol{z}_i$ the label vector for sample $\boldsymbol{x}_i$.

Rather than λ, the `MLPClassifier` employs the term of $\alpha = n\lambda$, where n is the number of samples, to parametrize the loss. To be consistent with our logistic regression implementation, we should therefore pass $\lambda * n$ as value for `alpha` .

The `MLPClassifier` allows for different optimizers. For small datasets, the LBFGS provides a reliable and effective approach. We will therefore use `solver = 'lbfgs'` for the Iris dataset. For larger datasets, the algorithm is not particularly efficient, since each weight update requires a full iteration over the whole training set. For this reason, for the project the suggestion is to switch to the Adam solver `solver = 'adam'`. The Adam solver is a solver based on stochastic gradient descent that improves on standard SGD by dynamically adapting the learning rates. Due to its stochastic nature, it optimizes the loss function in batches, whose size can be controlled through the `batch_size` parameter. We will use the default value (200) for the project. The effectiveness of the algorithm also depends on the initial learning rate. Also in this case, the default value is good enough for our data, but in general it may be worth trying different initial values.

In both cases (LBFGS or Adam), it may be necessary to increase the maximum number of iterations to avoid stopping the training too early. We will use `max_iter = 2000` in the following.

The non-linear activation function can be controlled through the `activation` parameter. For this laboratory we will employ the hyperbolic tangent function, i.e. we set `activation = 'tanh'` .

As an example, to train a 2-hidden-layer MLP with 10 nodes per hidden layer with hyperbolic tangent activation functions using L-BFGS we would call

```
clf = sklearn.neural_network.MLPClassifier(random_state=0, activation='tanh',
    solver='lbfgs', max_iter=2000, alpha=lamb*DTR.shape[1], hidden_layer_size =
     [10, 10])
```

where `lamb` is the regularization coefficient λ .

Once we have instantiated the `MLPClassifier` class, we can train our model by calling the `fit` method. The method receives the training data matrix and the label array. Since the library assumes samples to be rows of the data matrix, we need to remember to transpose our data before calling the `fit` method:

```
clf.fit(DTR.T, LTR)
```

Finally, the `MLPClassifier` object allows to compute class posterior probabilities for a test data matrix with

```
logProb = clf.predict_log_proba(D.T).T
```

Also in this case, we need to transpose the input data, and transpose the function output so that each *row* of `logProb` corresponds to a class, and each *column* of `logProb` corresponds to a sample.

For binary problems, we can convert the output to class posterior ratios and obtain an array of LLRs by removing the prior log-odds as we did for Logistic Regression. Note that `logProb` contains two rows also for binary problem: the first row constains values $\log P(C = 0|\boldsymbol{x}_i)$, whereas the second row contains values $\log P(C = 1|\boldsymbol{x}_i)$. Log-posterior ratios can then be computed as `logProb[1] - logProb[0]` .

Implement a binary neural network classifier for the binary version of the Iris task. The classifier should be trained on the model training part of the dataset, and compute binary log-posterior ratios (as for Logistic Regression) and LLRs (by removing empirical training prior log-odds) for the validation part of the dataset (refer to previous laboratories for the training / validation splits).

Below you can find the loss, error rate (*computed comparing log-posterior scores with 0*), minDCF and actDCF with $\pi = 0.5$ (*these should be computed with LLR scores, not with log-posterior scores*) for the Iris dataset with different values of λ with different architectures. The first architecture does not contain any hidden nodes, and is thus equivalent to a linear logistic regression model (results may slightly differ since the L-BFGS implementation we employed is slightly different from the one used by sklearn, resulting in slightly different convergence criteria). We then consider a network with 1 hidden layer with 10 nodes and a network with 2 hidden layers with 10 nodes. The loss value can be computed from the trained instance `clf` as `clf.loss_` .
NOTE: due to numerical errors and different library implementation, for more complex models (larger number of hidden layers) the results may slight differ, but should be close to those reported below. Note that the first and third rows of the table correspond to the results of the logistic regression model we got in the previous laboratory.

Hidden layers	Regularization Coefficient	Loss	Error rate	minDCF ($\pi_T = 0.5$)	actDCF ($\pi_T = 0.5$)
None	$\lambda = 10^{-3}$	`1.100009e-01`	8.8%	0.0625	0.1181
None	$\lambda = 10^{-2}$	`2.429623e-01`	8.8%	0.0556	0.1181
None	$\lambda = 10^{-1}$	`4.539407e-01`	11.8%	0.0556	0.1111
[10]	$\lambda = 10^{-3}$	`4.631468e-02`	11.8%	0.0625	0.2292
[10]	$\lambda = 10^{-2}$	`1.619642e-01`	8.8%	0.0625	0.1181
[10, 10]	$\lambda = 10^{-3}$	`2.541412e-02`	11.8%	0.1181	0.2292
[10, 10]	$\lambda = 10^{-2}$	`1.356446e-01`	8.8%	0.0625	0.1736

Project

Implement a neural network classifier for the project data. It's suggested to switch the optimizer from `'lbfgs'` to `'adam'` . As for the logistic regression laboratory, models should be evaluated in terms of minimum and actual DCF for and application with $\pi_T = 0.1$.

Try a network with 1 hidden layer. Try different sizes for the hidden layer (for example, 10, 20 and 40). For each layer size, select the regularization coefficient λ by cross-validation (suggestion: try values of lambda in the range `numpy.logspace(-6, -3, 7)`. Consider minimum DCF. What do you observe? Is there a model that performs better than logistic regression or quadratic logistic regression? How does the regularization affect the different models?

Now try a network with 2 hidden layers. Try again different sizes for the layers (for example, 10, 20 and 40). As before, analyze also the effects of the regularization term λ. What do you observe? Can you identify a model that performs better, in terms of minDCF, than the 1-hidden layer best model? How does the regularization affect the different models? Is it more relevant than for the 1-hidden layer case? Explain the different results in terms of the model characteristics (e.g. under-fitting, over-fitting, regularization effects, model complexity, ...)

Finally, analyze the actual DCF of the different models. Are the models well-calibrated (actDCF within few percent points of minDCF) for the target application?