

Data structures and preprocessing

For feeding into a neural network built with dense layers, the input data should be arranged to a NumPy array with two dimensions and shape (samples, features). This can be viewed as a table of numbers: each row corresponds to a particular sample, and the columns correspond to a specific feature of the sample.

For regression or binary classification problems, the labels are stored in one-dimensional NumPy arrays of shape (samples,). However, for multiclass classification problems, it is advisable to use one-hot labeling (to_categorical in Keras).

Featurewise normalization to mean zero and unit standard deviation.

Data preprocessing can be necessary if

- there are missing values in the data or
- the numerical values of some feature vary over a broad range

```
mean = train_data.mean(axis=0)
train_data -= mean
std = train_data.std(axis=0)
train_data /= std
test_data -= mean
test_data /= std
```

Model architecture

Selecting the optimal architecture for a particular project is a difficult task. However, there are some guidelines that help in making the decision.

- A bigger model (more layers and more neurons in each layer) is better in the sense that it can produce a broader range of possible representations. However, a bigger model is also more difficult and slow to train. Also, it is more susceptible to overfitting, reproducing irrelevant features specific to the training set (with no general predictive value).
- The optimal model depends on the amount of available training data. The less data you have, the worse overfitting will be.
- ReLU is a popular activation function to try first for the hidden layers. The activation function for the final output layer depends on the problem type:
 - regression: no activation
 - binary classification: sigmoid
 - multiclass classification: softmax

Compiling the model

The choice of the loss function is made according to the problem type. Good choices are

- regression: mse (mean squared error)
- binary classification: binary_crossentropy
- multiclass classification: categorical_crossentropy

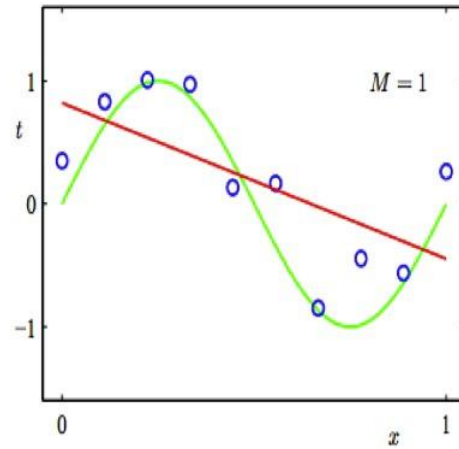
The optimizer takes care of updating the model weights as dictated by the computed gradient of the loss function. The 'rmsprop' optimizer is a good choice for many problems.

In compiling the model, the choice of 'metrics' dictates what is presented on the computer screen while the training is in progress. This also depends on the problem type; use e.g.

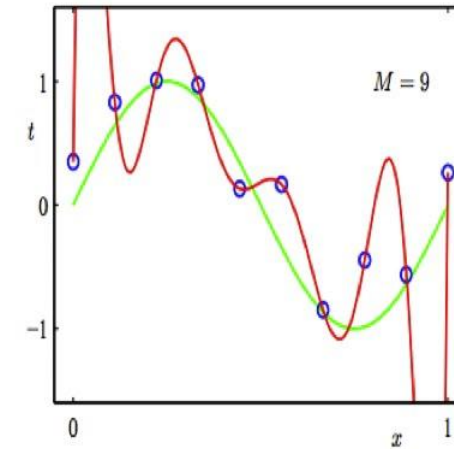
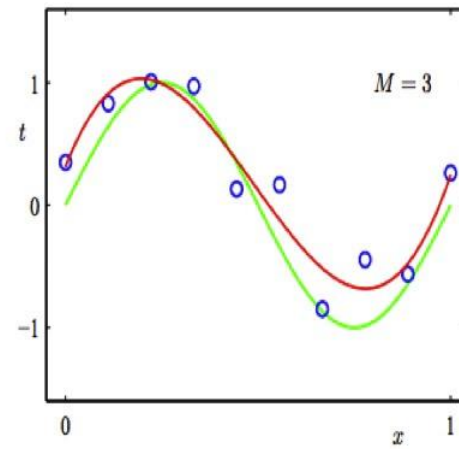
- regression: ['mae'] (mean absolute error)
- classification: ['accuracy']

Examples of under- and overfitting

Regression:

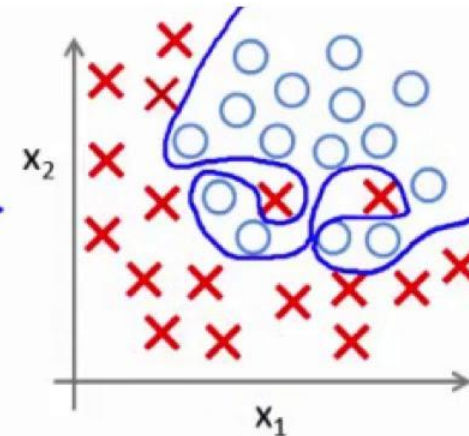
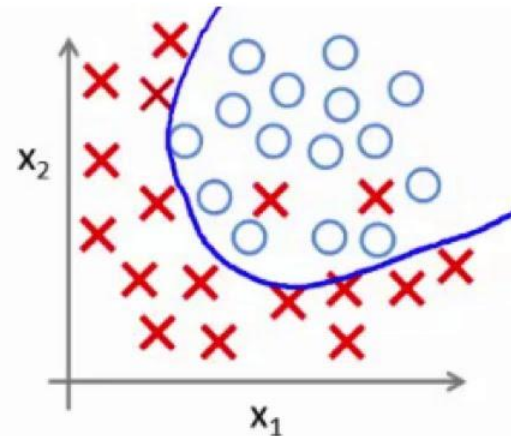
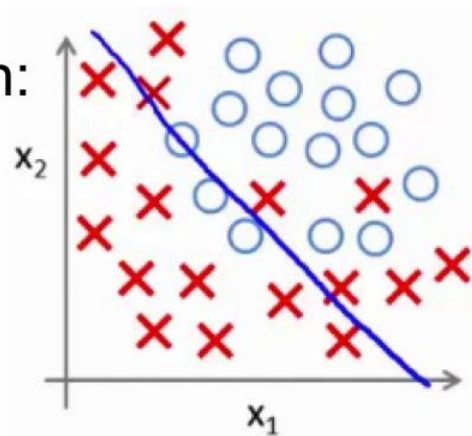


predictor too inflexible:
cannot capture pattern



predictor too flexible:
fits noise in the data

Classification:



Fighting overfitting

Overfitting is a central issue in every single machine learning project. It can be fought against by

- Adding more data
 - Always the best option, but usually not feasible. With more data, the possibility to pick up patterns specific to the training set decreases.
- Simplifying the model
 - Reduces overfitting, but might also prevent finding all the relevant patterns by restricting the hypothesis space.
- Adding regularization
 - Overfitting usually involves parameters acquiring values varying over a broad range. This can be prevented by adding a suitable penalty term to the loss function.
- Adding dropout
 - A form of regularization, where a small number of randomly selected units in the model are dropped out of training.

Implementing regularization

Overfitting often manifests itself so that the trainable parameters (weights) of the iterated network end up having values that vary on a wide range. Regularization is a technique for preventing this to happen. It is achieved by introducing an additional "penalty" term to the loss function that depends on the magnitudes of the weight parameters. Two common choices are terms proportional to the sum of the absolute values of the weights (L1 regularization) or the sum of the squares of the weights (L2 regularization).

Regularization discourages large variations in the weight values, making the value distributions more regular. This reduces overfitting.

This shows an example of how (L2) regularization can be implemented with a layer in Keras. The parameter in the parenthesis controls the overall magnitude of the penalty term.

```
from tensorflow.keras import regularizers
```


```
model.add(Dense(units=16, kernel_regularizer=regularizers.l2(0.01), activation='relu'))
```



Introducing dropout

Besides regularization, the dropout technique provides another useful method for preventing a model to overfit. In this method, some selected percentage of the neuron output values of a particular network layer is set to zero. The outputs set to zero are selected in random, and only for a single computation through the network (for the next run, a new random selection takes place). This strategy reduces overfitting in the sense that it makes learning irrelevant details more difficult, which improves the robustness of the final model.

```
from tensorflow.keras.layers import Dropout  
  
model.add(Dense(units=16, activation='relu'))  
model.add(Dropout(0.3))
```



Dropout is implemented in Keras by adding a separate layer immediately after the relevant layer. The parameter in parenthesis is the proportion of randomly selected neuron outputs in the previous layer to be set to zero (here 30%).