ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

Department of Information Technology and Electrical Engineering

Machine Learning on Microcontrollers

227-0155-00L

Exercise 3

Model training and evaluation: an introduction to scikit-learn & Keras/PyTorch

Michele Magno, PhD
Viviane Potocnik
Marco Giordano
Pietro Bonazzi

Wednesday 2nd October, 2024

1 Introduction

In this lab session you will learn the basics of Machine Learning (ML) and have a practical experience on how to use the most common ML tools to solve classification tasks.

If you don't have any experience with ML or you want to review what you already know, you can read through this section to get the core ideas. Otherwise, you can jump directly to section 5 for the exercises.

1.1 Basics of Machine Learning

ML is defined as a set of methods that can automatically detect patterns in data, and then use the uncovered patterns to predict future data, or to perform other kinds of decision making under uncertainty (ref. K. P. Murphy, *Machine Learning A Probabilistic Perspective*).

There are different kinds of ML problems, such as:

- Classification: the goal is to learn an indicator function. For example, given a dataset containing pictures of cats and dogs, you want your model to learn how to separate cats from dogs, i.e. given new pictures of cats or dogs, the model should be able to correctly tell if it's a picture of a cat or a picture of a dog.
- Regression: the goal is to learn a real-valued function. For example, one wants to predict the height of a person knowing his/her shoe size or predict how the sales of passenger vehicles in India will be next year.
- Dimension reduction: the goal is to learn a linear or non-linear projection of your dataset. One example is the so-called maximally informative dimensions technique, which is a dimensionality reduction technique used in neuroscience to project a neural stimulus onto a low-dimensional subspace so that as much information as possible about the stimulus is preserved in the neural response.
- Data compression: the goal is to learn an encoding for efficient representation, for example image or audio compression. Data compression can be either lossy or lossless (e.g. dimensionality reduction is a form of lossy compression).

There are four types of learning:

- Supervised learning: learn a function that maps an input to an output based on example input-output pairs (labels given). Examples of supervised learning algorithms: Decision Trees, Nearest Neighbor, Naïve Bayes, Linear Regression, SVM, etc.
- Unsupervised learning: learn from test data that has not been labeled, classified or categorized (no labels). Examples: K-means clustering, association rules, density estimation, PCA, etc.
- Semi-supervised learning: make use of unlabeled data for training – typically a small amount of labeled data with a large amount of unlabeled data
- Reinforcement learning: take actions in an environment so as to maximize some notion of cumulative reward

In this course we will mostly be focusing on classification or regression tasks using supervised learning techniques.

1.2 General Workflow

Figure 1 shows the general ML workflow for a supervised learning case.

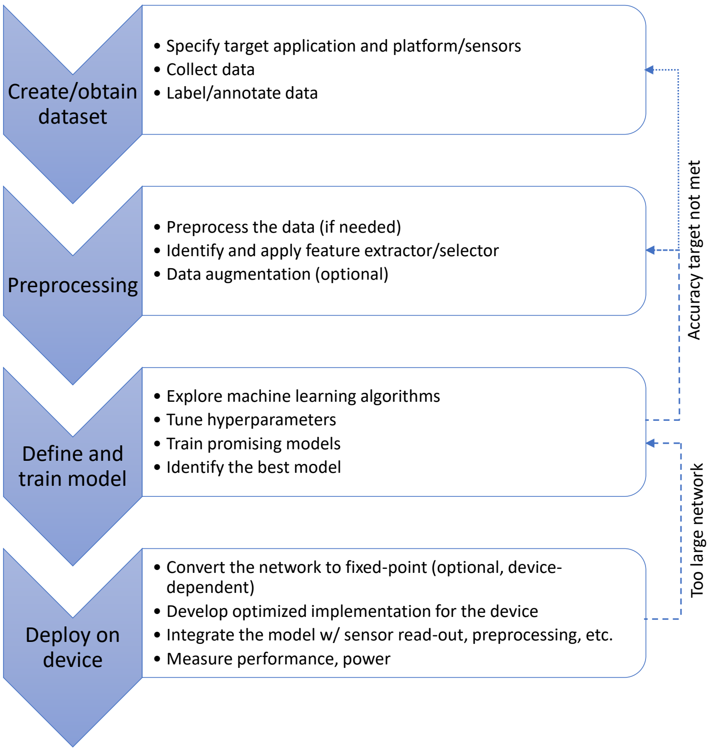

Create/obtain dataset
• Specify target application and platform/sensors
• Collect data
• Label/annotate data
Preprocessing
• Preprocess the data (if needed)
• Identify and apply feature extractor/selector
• Data augmentation (optional)
Define and train model
• Explore machine learning algorithms
• Tune hyperparameters
• Train promising models
• Identify the best model
Deploy on device
• Convert the network to fixed-point (optional, device-dependent)
• Develop optimized implementation for the device
• Integrate the model w/ sensor read-out, preprocessing, etc.
• Measure performance, power
Accuracy target not met
Too large network

Figure 1: General ML workflow.

After identifying your target application, you first collect the dataset and label it if it's not already available, then apply preprocessing steps, if needed, to clear the data and enhance the signal-to-noise ratio followed by feature extract and/or selection. Afterwards, you split the dataset and start the training process. The dataset is divided into:

- Training set: used for learning the function (fit the parameters, e.g. weights, of for example a classifier) which maps the input to the output.
- Validation set: used for validating the model, i.e. identify the model and tune the hyperparameters (i.e. the architecture) of a classifier. It also helps to avoid overfitting.
- Test set: independent from the other sets and used for testing the final trained model. If a model which fits to the training dataset also fits the test dataset well, minimal overfitting has taken place.

It is common to find ML applications where the training set and the validation set are merged into one single set (e.g. using cross-validation when the dataset is small). Sometimes the validation set is called test set, and the final model is directly tested on real-world data during application.

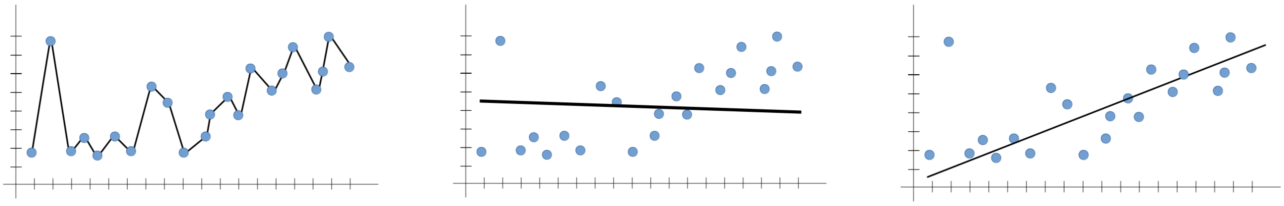
1.3 Underfitting and Overfitting

One of the key points in ML is the problem of underfitting/overfitting. When you train a model, you want your trained model to be able to generalize, i.e. the model should be able to give sensible outputs to sets of input that it has never seen before. Based on this idea, the term underfitting and overfitting refer to deficiencies in the model's performance to predict on unseen data.

Overfitting, or sometimes called overtraining, happens when a statistical model contains more parameters than the amount of data available for training. In other words, the model fits too closely or exactly to a particular set of data, such that some of the residual variation (i.e. the noise) is also extracted as if that variation represented underlying model structure. Overfitted model may therefore fail to fit additional data or predict future observations reliably. An illustrative example of overfitting on linear regression is shown in Figure 2a.

Contrarily, underfitting, or undertraining, occurs when the trained model cannot appropriately capture the underlying structure of the data (see Figure 2b). It happens when the model lacks some parameters or terms that would be necessary for a correctly specified model. For example, it occurs when fitting a linear model to non-linear data. Such a model will not be able to predict the non-linear behaviour of the data.

Finally, Figure 2c shows the desired predictive model in an example of linear regression.



(a) Overfitting.

(b) Underfitting.

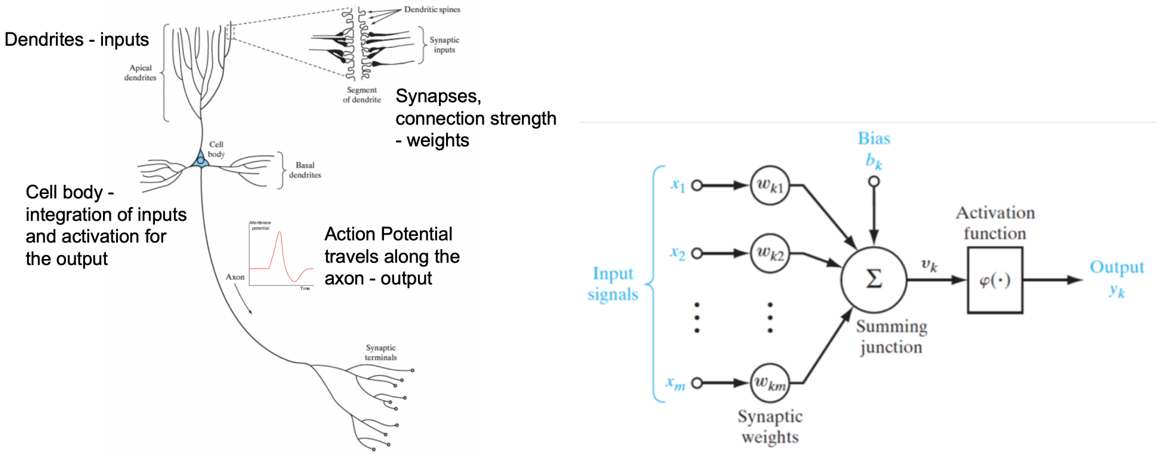
(c) Desired model.

Figure 2: Illustrative example of overfitting and underfitting based on linear regression problem.

2 Introduction to Artificial Neural Networks

Artificial Neural Networks (ANNs) are computing systems inspired by biology, more specifically the brain. The nervous system of most known beings is made up of many neurons, shown in Figure 3a, connected together in different ways. As a reference, a neuron may decide to 'fire', or transmit a signal along the axon to the synaptic terminals, based on the input signals at the dendrites. Connecting many millions of these together allows processing of information from vague inputs to valuable, more refined information.

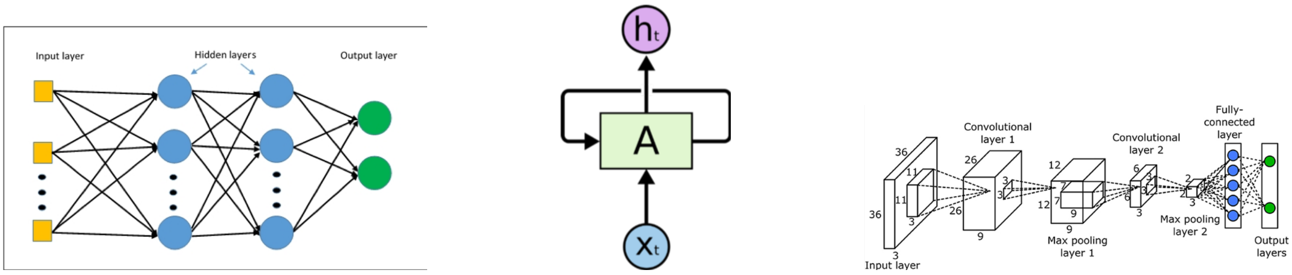
An ANN attempts to emulate this, combining inputs in different ways and deciding whether or not to activate, thereby processing the given information, as shown in Figure 3b. Inputs, which may go to many different neurons, are each multiplied by a weight, computed during training. The sum of all these weighted inputs determines if the neuron 'fires' or not. Connecting these neurons to many layers allows similar processes as in biology, processing signals to valuable information.



(a) (b)

Figure 3: Illustrations of biological neuron (a) and artificial neuron (b).

There are many ways neurons can be connected to each other, however the overall architecture is similar. Initially, the input data is transmitted through an input layer, which acts as an initial processor for the given data, often expanding the data to allow more detailed computations. Layers usually take many inputs, often sharing these with other neurons, and process these to produce one output. Structuring these in layers allows for sequential computation, taking signals from one layer and feeding them into the next. After the initial input layer, the signals are usually forwarded to many hidden layers, where the size and function may vary. Ultimately a final output layer is used to determine the result of the computation.



(a) Feedforward Neural Network (NN)

(b) Recurrent NN

(c) Convolutional NN

Figure 4: Commonly used types of artificial neural networks.

There are different types of NNs, mostly varying in the function of the hidden layers and the type of connections between neurons. Figure 4 demonstrates the most commonly used NNs. Feedforward NNs simply have neuron layers connected in a *feedforward* way, i.e. the connections between the nodes do not form a cycle. Recurrent NNs can use their internal state (memory) to process sequences of inputs and the connections between the nodes form a directed graph along a temporal sequence. Finally, convolutional NNs have convolution layers where convolutional filters are seen as neurons and are applied to the input data (1D signals, 2D images, or 3D volumes).

These different types of networks come with different advantages, e.g. recurrent networks allow for learning with respect to spatial or temporal dependencies of a series of input feature vectors whereas convolutional networks are best at image processing. During this course we will be using mostly feedforward NNs and convolutional NNs.

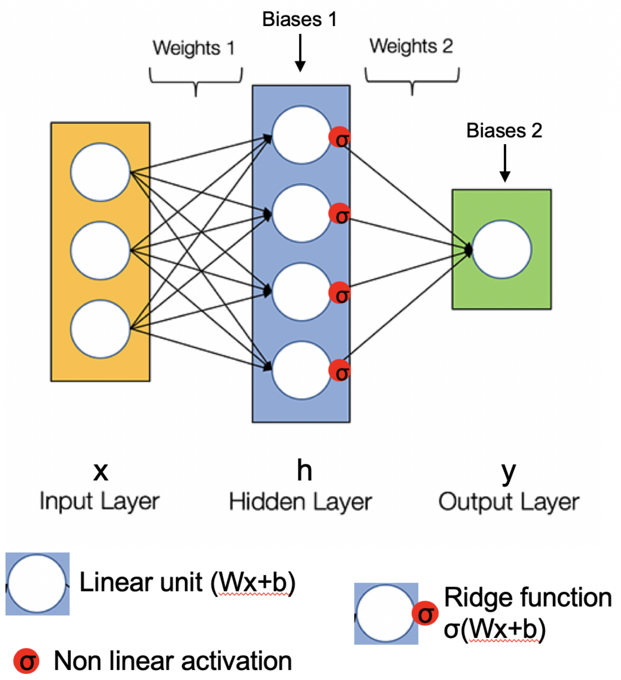


Figure 5: Structure of a feedforward neural network.

3 Feedforward NNs (Multi-layer perceptrons)

3.1 Structure

A feedforward NN, as the name suggest, is a neural network which only processes information in the forward direction, i.e. there are no cycles or feedback paths. Multi-Layer Perceptron (MLP) falls in this category. The neurons, in this case also called perceptrons, are arranged in subsequent layers and connected in a feedforward manner from the input to the output. If every neuron in a precedent layer is connected to all the neurons in the following layer, then the layer is called dense or fully-connected layer, otherwise it is named sparsely-connected layer.

Each perceptron can be modeled with a linear and a non-linear part, the linear part being a matrix multiplication of a learned weight matrix W with the input vector x and the addition of some bias vector b, i.e. $Wx + b$. The non-linear part comes with the activation function σ, turning the output of the neuron into $\sigma(Wx + b)$. Activation functions come in many styles, the most famous ones being ReLU, sigmoid and tanh.

3.2 Training a Neural Network

For supervised learning, we are given large amounts of training data that each have an input x and an output y. To train our network, we want to find W and b that solve following equation:

$$\hat{y} = \sigma(W_2\sigma(W_1x + b_1) + b_2)$$

This however is a numerically hard task, since many matrix inversions are not well-conditioned and the inversion of the non-linear activation function might not even uniquely exist or be numerically unstable as well.

3.3 Optimizers and Loss functions

Due to the above mentioned difficulties, we do not actually ever invert any part of this equation and instead opt for stepwise optimization of our solution by using a method called backpropagation on a loss function. This so called loss function maps the output of our current solution and the correct output to some real number which represents the difference or error in output. Well-known loss functions are the quadratic error function or the cross-entropy. We then update the weights and biases of our network by backpropagating this loss with an optimizer.

The choice of loss function and optimizer is an art in and of itself, so some experience and experimentation can often lead to better results.

To help you better understand this process, let's take a look at the most important optimizer, Stochastic Gradient Descent (SGD) with momentum term. We formulate training as an optimization problem. Let L_i be the loss function and ω the parameters of layer i of the whole network:

$$L(\omega) = \sum_i L_i(\omega)$$

$$\text{minimze } L(\omega)$$

The choice of optimizer dictates how we go about minimizing the loss function. The most important and standard optimizer is called SGD and optimizes by simply updating ω in the direction of steepest descent. The standard version would update ω by the following rule:

$$\omega := \omega - \eta \nabla L(\omega)$$

where we call η the learning rate.

Since this form only takes into account gradient, we might end up in a local minimum, which might not be what we want. To solve this problem, we can add a momentum term, which allows us to ”escape” a local minimum. The rule is then adapted as follows:

$$\Delta\omega := \alpha\Delta\omega' - \eta\nabla L(\omega)$$
$$\omega := \omega + \Delta\omega$$

Where ω' is the previous update of the weights. This leads to two parameters we can tune, learning rate η and momentum term α. Controlling these two parameters can be essential to end up with a well-working network.

There are some more abstractions that are used with the training process, like minibatching, rate decay and many more, the essential part however is understanding the way the loss function and the optimizer work.

In the next example the quadratic error loss function is implemented and used with a SGD optimizer without momentum term.

```
class NeuralNetwork:
    def __init__(self, x, y):
        self.input = x
        self.weights1 = np.random.rand(self.input.shape[1],4)
        self.weights2 = np.random.rand(4,1)
        self.y = y
        self.output = np.zeros(y.shape)

    def feedforward(self):
        self.layer1 = sigmoid(np.dot(self.input, self.weights1))
        self.output = sigmoid(np.dot(self.layer1, self.weights2))

    def backprop(self):
        # application of the chain rule to find derivative of the loss
        #   function with respect to weights2 and weights1
        d_weights2 = np.dot(self.layer1.T, (2*(self.y - self.output) *
            sigmoid_derivative(self.output)))
        d_weights1 = np.dot(self.input.T,  (np.dot(2*(self.y - self.output) *
            sigmoid_derivative(self.output), self.weights2.T) *
            sigmoid_derivative(self.layer1)))

        # update the weights with the derivative (slope) of the loss function
        self.weights1 += d_weights1
        self.weights2 += d_weights2
```

Luckily this is all already implemented in many publicly available libraries, such as `scikit-learn`, `Keras` and `PyTorch`.

4 Notation

Student Task: Parts of the exercise that require you to complete a task will be explained in a shaded box like this.

Note: You find notes and remarks in boxes like this one.

5 Preparation

For this lab you will need Python 3.8 as well as Jupyter Notebook. Make sure these tools are installed before you start. Also download the exercise folder from the course website.

Note 1: The simplest way to install Python 3.8 and Jupyter Notebook is using Anaconda. It can be downloaded on the following website: `https://www.anaconda.com/distribution/`

Alternatively in Linux or macOS you can use your package manager to install `python3`.

Please be aware that some computers come pre-installed with python 2.7. As this is no longer maintained, please ensure you are using a newer python 3 installation.

The packets we require, *jupyter*, *scikit-learn*, and *matplotlib* are included with an Anaconda installation. If they are not on your system, you can either `pip install` or (if using Anaconda) `conda \ install` them. We will also need *pandas* and *tensorflow*, which can be installed as described above.

6 Jupyter Notebook

Jupyter Notebook is an interactive runtime environment for different programming languages. We will use it to run Python code snippets. You can run code snippets by selecting the code fields and pressing SHIFT + ENTER. Since the code is still run on a server which in this case is your machine, you have to make sure that any and all libraries you intend to use are installed and available.

Any variables and objects generated by code snippets you have run in a notebook are maintained over the whole notebook. You can also edit code segments and re-run them without reloading the notebook.

You can start a Jupyter server either from the Anaconda interface or by typing `jupyter notebook` into a shell.

7 First steps in Machine Learning

In this first exercise you will see how we can use *scikit-learn* to run classifiers like Support Vector Machine (SVM) and decision trees on data sets. We use the famous MNIST dataset for this task.

SVM is a supervised learning algorithm which uses hyperplanes to separate classes of features. The basic case is using linear regression, i.e. fitting a straight line between two classes of features. Feature classes which can be separated by a straight line are called linearly separable. To extend the method to non-linear learning problems, we can transform the feature space such that the feature classes become linearly separable. This, however, is no easy task and requires good intuition.

The advantages of SVM in comparison to other methods are the small number of parameters and easy portability training, however, SVM is prone to many pitfalls like duplicate data, overfitting on badly selected features and many more.

Decision trees use a statistical approach to formulate a set of decision rules to classify objects. These decision rules are induced by the data set and a measure function. In each step the feature which offers the best classification according to the measure function is found. This produces a tree of rules, where the leafs are the classes. The advantages of decision trees are again a small number of parameters and fast convergence on small datasets. The disadvantages are usually higher error rate and oftentimes bad generalization for non-linear problems. This can however be combated by using other measure functions.

Although SVM and decision trees are comparatively simple tools for classification, we need to remind ourselves that for microcontroller applications we aim to reduce the computation effort as much as we can. Using a simple model might therefore be beneficial to save energy when compared to more complex methods.

Student Task 1:

1. Download the exercise files from polybox and unpack them to a convenient location.
2. Open *Jupyter Notebook*.
3. Open the first exercise notebook.
4. Solve the tasks in the notebook.

8 Applying SVMs to real life data

In this task, you will use sensor data from a smartphone gyroscope and accelerometer. You will use an SVM to classify the data into twelve classes. In the first part of this task you will use a linear approach and in the second part you will use non-linear transforms to improve your classification results.

Student Task 2:

1. Open the second exercise notebook.
2. Download the dataset from the link provided in the notebook. Ensure the folder *HAPT Data Set* is in the same directory as the .ipynb notebooks (download the folder by clicking *Data Folder* on the website).
3. Solve the tasks in the notebook.

9 Tensorflow and Keras

Tensorflow is a framework for dataflow-oriented programming that provides access to different libraries and data structures which are used to describe operations on tensors rather than on single numbers. Tensorflow has found large popularity in the context of machine learning, where it has become one of the leading frameworks for training models in both academia and industry.

Since version 1.4 Keras is part of the Core API of Tensorflow. Keras has been originally designed to enable an easy-to-use API to describe neural networks. Keras supports most if not all modern deep network architectures, from convolutional approaches to recurrent layers. Models are easy to set up in Keras, you simply have to declare an empty model and add layers in the order you want them to be executed.

The great advantage of Tensorflow and Keras for training is the ubiquitous support of these frameworks by most embedded toolchains.

9.1 Getting started with Keras

When describing any model with any machine learning library, you have to keep few things in mind. Let's take a look at a small network and dissect it.

```
import tensorflow as tf
from tensorflow.keras import models, layers

inputShape = (28,28,1)
outputShape = (10)

batchSize = 100
epochs = 5

model = models.Sequential()

model.add(layers.Flatten(input_shape=inputShape))
model.add(layers.Dense(42, activation="relu"))
model.add(layers.Dense(outputShape, activation='softmax'))

model.compile(loss='categorical_crossentropy',
optimizer=keras.optimizers.Adam(),
metrics=['accuracy'])

model.fit(x_train, y_train, batchSize, epochs)
```

This code describes a small Fully-Connected neural network also named Multi Layer Perceptron (MLP). It is composed of three layers:

- An Input layer to specify the dimensions of the input tensor and format the input.
- A Dense hidden layer of 42 neurons binded with the ReLU activation function.
- A Dense output layer with a Softmax activation to return the normalized probability distribution of the classification.

The important thing to take note of is that you need to tell your model what kind of input it expects - in this case it is a tensor with dimensions 28x28x1, which could be a 28x28 pixel greyscale image. You also have to make sure your labels match with your output shape, i.e. if you have n different labels, you will need n different outputs.

The training parameters are given in the calls to *model.compile* and *model.fit*, let's take a look at the arguments passed to the *compile* method.

For the `loss` function, we use *categorical_crossentropy*. The *categorical* stands for the one-hot encoding property of the labels. This avoids implying that the label number of categories influences training, for more details see this link. The *crossentropy* is a statistical measure function for the quality of an estimated distribution function. What we expect to get with this loss function is a network which returns an estimation of probabilities over our categories, i.e. the numbers we get correspond to likeliness.

We have previously seen the function of the optimizer in training a neural network. Keras provides many `optimizers`, most of which build on the SGD approach and add some terms or modify some weights.

The `metrics` field specifies the output the network generates during training, you will usually want to leave accuracy there, but for some applications you might want to use mean-square error (mse) or different metrics.

In the model.fit call we pass the training set as well as batch size and number of epochs. The `batch_size` dictates how many input vectors you train at the same time - The basic case is using batch size = 1, which means that the complete backpropagation process with updating the model parameters is done for every input. Using bigger batch sizes (usually at most a few hundred) can have a couple of advantageous effects: model updates can become more stable, since outliers are averaged other train values. Also training is faster, since fewer updates have to be computed. Using too large batches can be detrimental however, since averaging over too many inputs does not perform well in practice. The `epoch` field sets how often the training procedure is repeated over the whole dataset.

With this information you should be able to get started with deep learning in Keras - There are of course much more advanced layer structures and parameters you can read up about on Keras.io.

If you feel unsure about your network architecture, you can always talk to one of the assistants.

Student Task 3:

1. First check out the documentation of Keras on `https://keras.io/`
2. Answer the following questions:
   - What is the so-called Sequential model in Keras? ____________________
   - What kind of layers are supported? ____________________
   - Which operation does the Dense layer implement? ____________________
   - What is an activation? ____________________
   - Write the equations of ReLU and Softmax activations. ____________________

     ____

     ____
   - Which layers would you use to implement a sparsely-connected multi-layer perceptron? ____________________
3. Now open the third exercise notebook.
4. Solve the tasks in the notebook.
5. What are your final train, validation, and test accuracies? ____________________

10 Multiply-Accumulate (MAC) Operations and Model Performances

When deploying Machine Learning model on power and compute-constrained platforms it is crucial to have good metrics to drive further optimizations. Two well used platform-agnostic metrics are the number of parameters and the number of MAC for a given model. While the number of parameters of a model is highly correlated to the minimum memory needed, the energy consumption and latency are partially dictated by the number of MAC.

Several tools are available to automatically get the number of MAC of a model. Given a PyTorch model you can use the pthflops package to directly get the number of MAC. Otherwise STM32 Cube IDE and GapFlow can display the number of MAC after loading your model.

Student Task 4:

1. Considering two matrix A and B of dimension $(L \times M)$ and $(M \times N)$ respectively. What is the minimum number of Multiply-Accumulate (MAC) operations to perform to compute C such that $C = A \times B$? ______________________________

2. Using what you've learned during the class and the previous exercises, find the number of MAC needed to run the inference of the Neural Network described in sub-section 9.1 (do not consider the activation functions). ______________________________

   ____

11 Getting started with pytorch

While Keras is one of the most popular frontends for machine learning in industry, PyTorch remains the most popular framework in academia. The main difference between the two framework lies in how a network is described; while Keras is designed around compiling a graph with its `models.Sequential` and `models.compile` functions, PyTorch is designed around allowing users to interconnect individual layers, which is done in a `forward` function. The code below implements the same network as task 3 using PyTorch.

```
import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

# Declare the network as a nn.Module
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=(3,3), padding=(1,1))
        self.pool1 = nn.MaxPool2d((2, 2))
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(4732, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = self.pool1(x)
        x = self.flatten(x)
        x = self.fc1(x)
        return x

network = Net()
optimizer = optim.Adam()
criterion = nn.CrossEntropyLoss()

# convert from numpy to torch tensors
xx_train = torch.from_numpy(x_train)
yy_train = torch.from_numpy(y_train)

torch_x_train = xx_train.permute(0, 3, 1, 2)

# combining x_train and y_train to make it usuable with DataLoader
train_data = list(zip(torch_x_train.float(), yy_train.long()))
trainloader = torch.utils.data.DataLoader(train_data,
shuffle=True, batch_size=100)

total_step = len(trainloader)
for epoch in range(epochs):  # loop over the dataset multiple times
    running_loss = 0.0
    for i, (inputs, labels) in enumerate(trainloader):

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = network(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

Student Task 5: Analyze the PyTorch implementation and consult the documentation if needed: `https://pytorch.org/docs/stable/index.html`

Answer the following questions:

- Where is the Softmax activation in the PyTorch code? ______________________
- What is the data format in PyTorch? ______________________
- What is the equivalent of `model.fit` in PyTorch? ______________________

  ____

  ____
- How do you implement `'same'` padding in PyTorch? ______________________

  ____

  ____

ℰ **Congratulations! You have reached the end of the exercise.**
If you are unsure of your results, discuss with an assistant. ℰ