Multi-Layer Perceptron

Sven Fritsch

Overview

- MLP Architecture
- Forward Pass
- Task 1 and 2
- Backward Pass
- Task 3
- Keras Example
- Convolutional Neural Networks

Motivation

As we have already seen in the first lecture, a perceptron is only able to create a linear decision boundary. For more complex classification tasks we want to be able to create a non-linear decision boundary. We can do this by extending the perceptron to multiple subsequent layers.


We need to go deeper

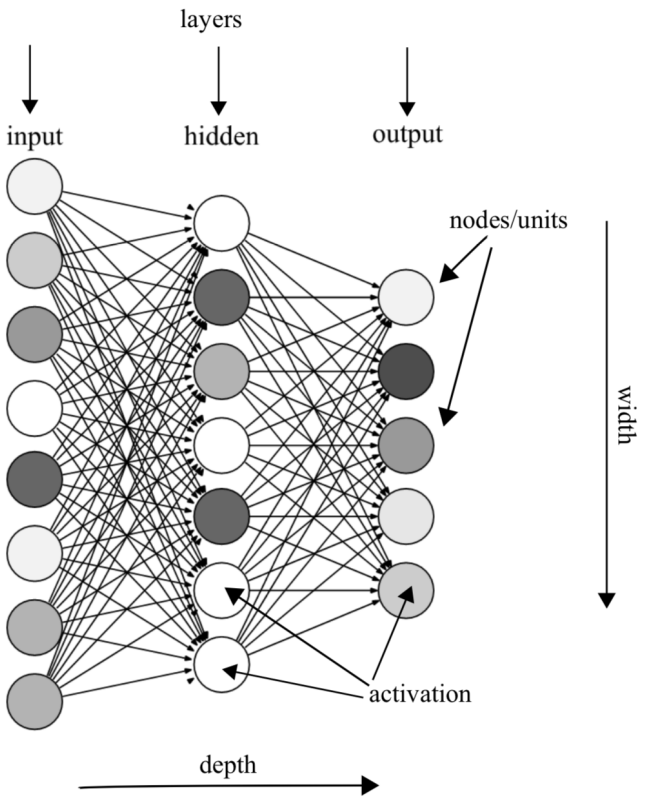
MLP Architecture

Multi-Layer Perceptrons (MLP) comprise 3 different layer types:

- one input layer (has no activation function)
- at least one hidden layer (has activation function)
- one output layer (has activation function)

As every node in a layer is connected with every node in the subsequent layer, hidden layers are also called fully connected layers. (In high-level APIs like Keras they are called Dense-layers.)

MLP Architecture

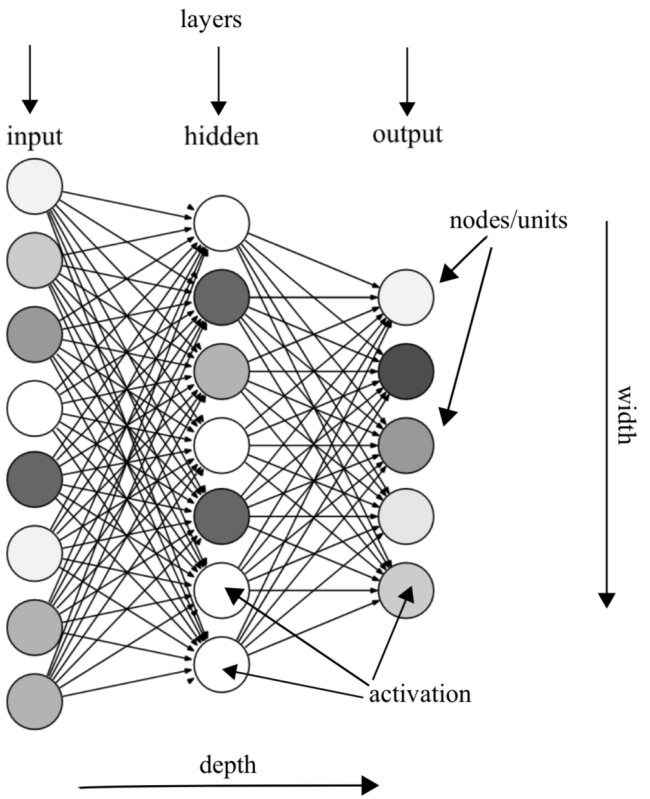


In the **input layer**, each node represents one input feature of a data sample.

Hidden layer(s) are needed to learn non-linear representations.

The **output layer** can consist of one (e.g. regression, binary classification) or more (e.g. multiple classes) nodes.

MLP Architecture



Width describes the number of nodes in a layer.

Depth describes the total number of layers in a model.

Weights and **biases** are learned parameters.

Hyperparameters

While model parameters like weights and biases can be learned (iteratively updated), hyperparameters **cannot be learned** by the model: They are selected by the programmer and refer to the model structure, thus having a potentially huge impact on model accuracy.

Hyperparameters

- **Number of hidden layers and nodes**: rather go deeper than too wide (8 layers with a total of 62,378,344 parameters in AlexNet)
- **Activation functions** for hidden and output units: linear, sigmoid, tanh, ReLU, softmax, ...
- **Learning rate** η: 0.1, 0.001, adaptive, ...
- **Training algorithm**: full batch mode, mini batch mode, stochastic gradient descent
- **Dropout**: percentage of neurons that are randomly deactivated during each epoch to prevent overfitting.

Hyperparameters

Generally, it is unknown which hyperparameters yield the best result for a certain model and application. Therefore, the task of choosing/ tuning the optimal set of hyperparemeters can be performed

- manually (trial and error)
- Random Search
- Grid Search
- Bayesian Optimization
- Reinforcement Learning (see Reference [1])

Samples

Samples: A sample is a single row of data. A training dataset is comprised of many rows of data, e.g. many samples. A sample may also be called an instance, an observation, an input vector, or a feature vector.

Batches

Batches: The batch size is a hyperparameter that defines the number of samples to work through before updating the internal model parameters. A training dataset can be divided into one or more batches.

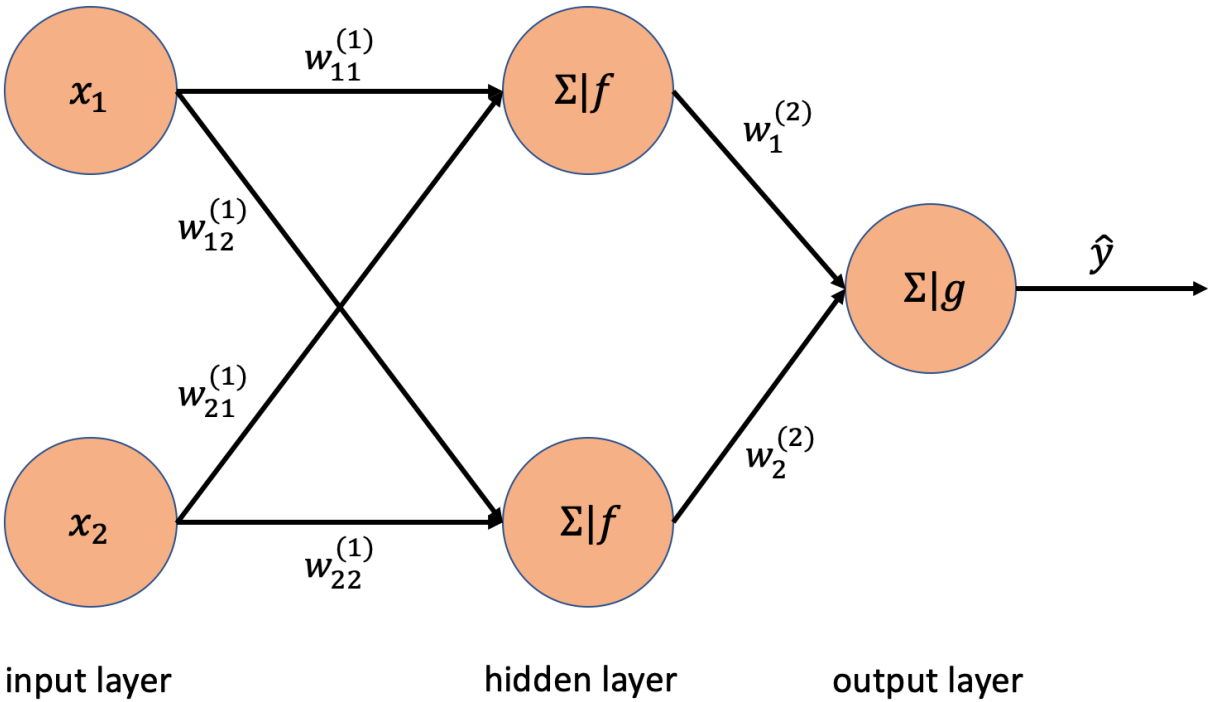
- Batch Gradient Descent: Batch Size = Size of Training Set
- Stochastic Gradient Descent: Batch Size = 1
- Mini-Batch Gradient Descent: 1 < Batch Size < Size of Training Set

Epochs

Epochs: The number of epochs is a hyperparameter that defines the number times that the learning algorithm will work through the entire training dataset. One epoch means that each sample in the training dataset has had an opportunity to update the internal model parameters.

MLP as a Function

How do we get from input x to output $\hat{y}$, when we assume learned weights $W^{(1)}$ and $w^{(2)}$?



A network with one hidden layer and one output unit can then be described by the function:

$$\hat{y} = g\left(w^{(2)^T} f\left(W^{(1)^T} X\right)\right)$$

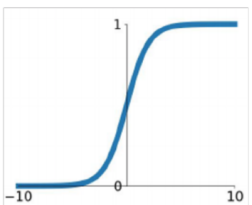
Activation Functions

Why? To enable our model to represent non-linear relationships

Activation functions for hidden units:

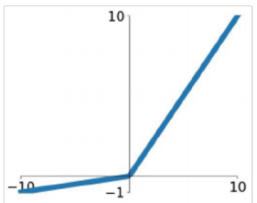
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



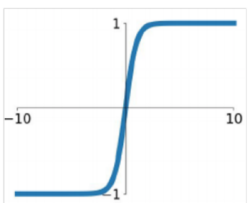
Leaky ReLU

$$\max(0.1x, x)$$



tanh

$$\tanh(x)$$

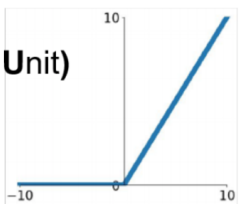


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

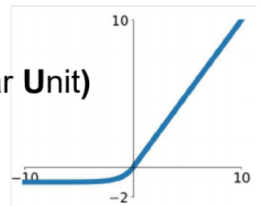
ReLU
(Rectified Linear Unit)

$$\max(0, x)$$



ELU
(Exponential Linear Unit)

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Activation Functions for Hidden Layers

The three most popular activation functions used for hidden layers are:

- Sigmoid
- tanh
- ReLU

As there is a significant drawback for both Sigmoid and tanh known as vanishing gradient problem most state-of-the-art models use ReLU activation for hidden layers.

Activation Functions for Output Layers

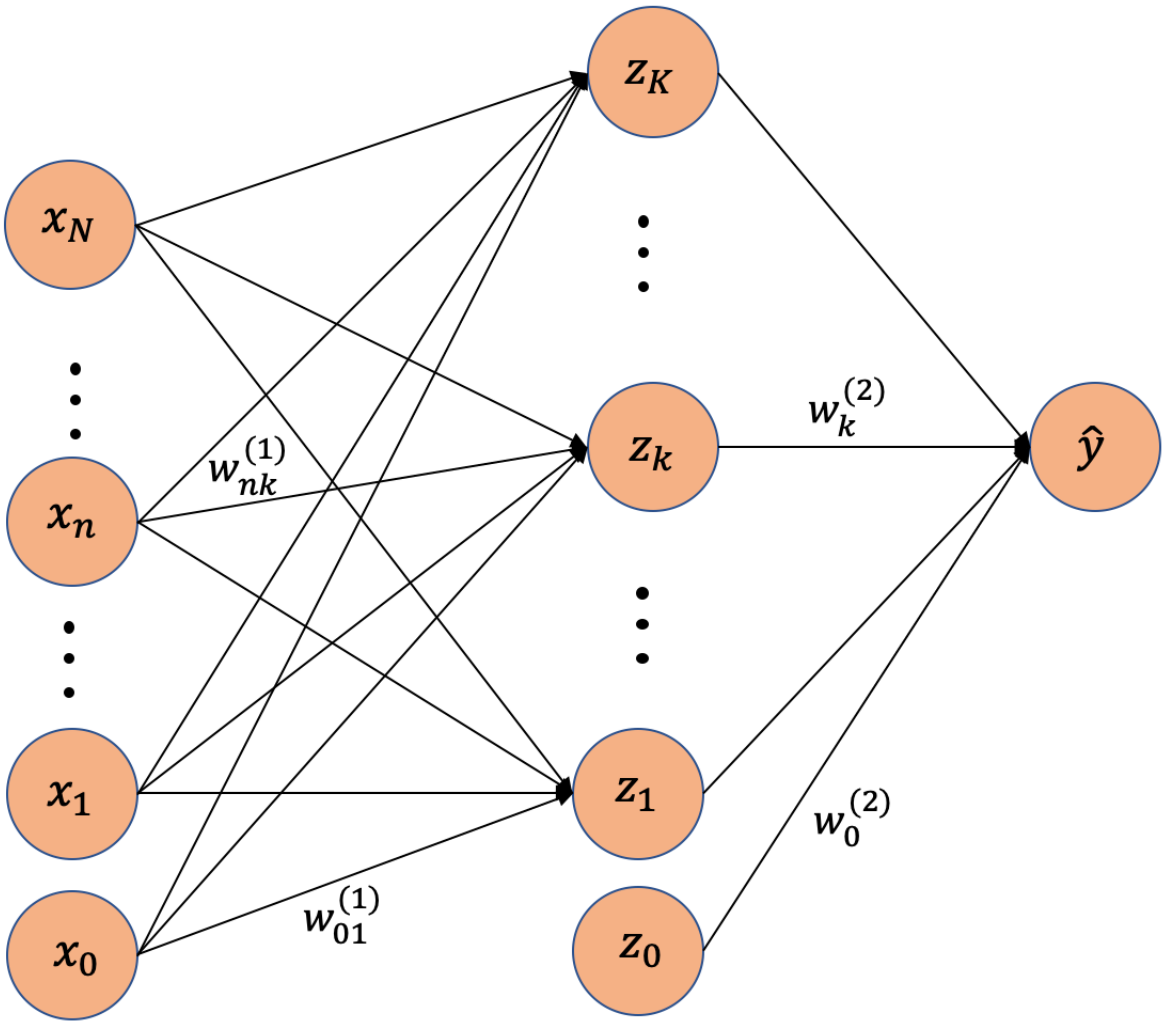
The choice of activation for output units depends on the task that we want to solve with our model. The output layer should use either

- softmax activation for classification as it gives probabilities for different classes or
- linear activation for regression

Task 1 - Linear Activation Function

Consider an MLP with an input layer with N input neurons, one hidden layer with K hidden neurons and an output layer with a single output neuron. The input of the n^{th} input neuron is denoted by x_n which is euqal to the output of the same input neuron. The input and output of the k^{th} hidden neuron are denoted by a_k and z_k, respectively, while the output of the single output neuron is denoted by $\hat{y}$. $w_{nk}^{(1)}$ defines the weight connecting the n^{th} input neuron to the k^{th} hidden neuron, and $w_k^{(2)}$ the weight connecting the k^{th} hidden neuron to the output neuron. $w_{0k}^{(1)}$ and $w_0^{(2)}$ are the biases in the first and second layer, respectively. x_0 and z_0 are bias nodes, do not receive any input, and are always $x_0 = z_0 = +1$.

Task 1 - Linear Activation Function



Task 1 - Linear Activation Function

In this task all neurons have a linear activation function, thus the output of a hidden neuron can be written as

$$z_k = -w_{0k}^{(1)} + \sum_{n=1}^{N} w_{nk}^{(1)} x_n \tag{1}$$

and the total output becomes

$$\hat{y} = -w_0^{(2)} + \sum_{k=1}^{K} w_k^{(2)} z_k \tag{2}$$

Show that there exists an MLP without hidden layers, which models the same function.

Task 2 - Multilayer Perceptron

Consider a three-layered network (one input, one hidden, one output layer) together with a sum-of-squares error, in which the output units have linear activation functions, so that $\hat{y}_j = a_j$, while the hidden units have sigmoidal activation functions given by

$$h(a) = \tanh(a) \tag{3}$$

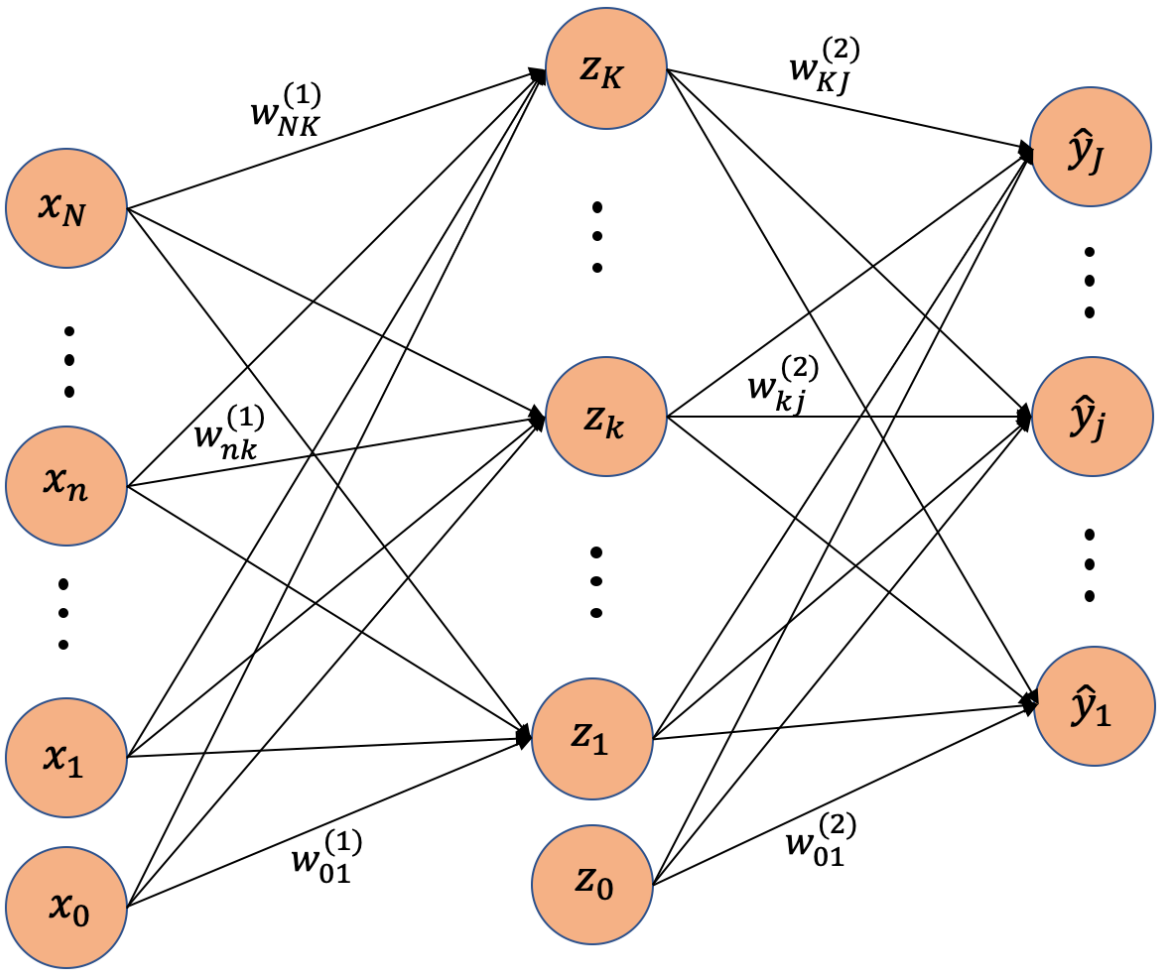
$$\text{with} \qquad \tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}} \tag{4}$$

$$\text{and} \qquad h'(a) = 1 - h(a)^2. \tag{5}$$

We also consider a standard sum-of-squares error function, so that for data point n the error is given by

$$E_n = \frac{1}{2} \sum_{j=1}^{J} (\hat{y}_j - y_j)^2 \tag{6}$$

Task 2 - Multilayer Perceptron



Task 2 - Multilayer Perceptron

$\hat{y}_j$ denotes the output of the j^{th} output neuron (e.g. a predicted label in case of classification) whereas y is the corresponding label for a particular input x_n, i.e. the "true label" or also sometimes referred to as "ground truth". The output layer starts at index j=1 as it does not contain a bias for the subsequent layer (makes sense since it's per definition the last layer of the network).
Compute forward propagation, i.e. compute the activation of the neurons in the hidden layer a_k and the output of the output layer neurons $\hat{y}_j$.

a) $a_k =$

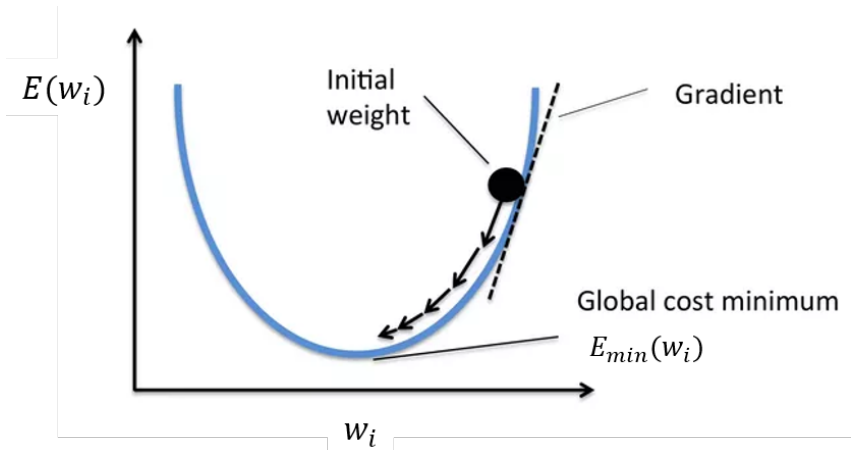
b) $\hat{y}_j =$

Gradient Descent

We calculate the cost of our model with an error function, e.g. squared error $E_n = \frac{1}{2}\sum_{j=1}^{J}(\hat{y}_j - y_j)^2$, where $\hat{y}_j$ is predicted and y_j is the desired output, i.e. the true label.

We then use gradient descent to minimize the cost function by iteratively updating the weights

$$w_{ij} \leftarrow w_{ij} - \eta \frac{\partial E}{\partial w_{ij}}$$



where η is the learning rate.

The Chain Rule of Calculus

We use the chain rule to efficiently compute the gradients

$$z = f(g(x))$$

$$y = g(x)$$

$$z = f(y)$$

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$

Back Propagation

To perform back propagation we need the derivative of the activation function:

$$z_k = h(a_k) = \tanh(a_k)$$

$$\frac{\partial z_k}{\partial a_k} = 1 - h(a_k)^2$$

We also consider a standard sum-of-squares error function, so that for datapoint n the error is given by

$$E_n = \frac{1}{2} \sum_{j=1}^{J} (\hat{y}_j - y_j)^2$$

Task 3 - Back Propagation

Consider the model in Task 2. To perfom backpropoagation, we need to compute the derivative of the error function w.r.t. the corresponding weights. Compute the derivative w.r.t. the first and second-layer weights. Hint: Use the chain rule.

(a) $\frac{\partial E_n}{\partial w_{kj}^{(2)}} =$

(b) $\frac{\partial E_n}{\partial w_{nk}^{(1)}} =$

Code Example of MLP Implementation in Keras

The MNIST database of handwritten digits (0-9) has a training set of 60,000 images and a test set of 10,000 images. After importing keras and MNIST dataset, an empty Sequential model is built. We will now add three fully connected layers (one input, one hidden, one output) with dropout layers in between.

Code Example of MLP

As described in Chapter MLP Architecture, Dropout is a hyperparameter randomly deactivating a certain percentage of nodes, changing each training epoch to avoid overfitting data. Afterwards the optimizer (sgd: stochastic gradient descent) is defined, the model compiled and evaluated.

Code Example of MLP

```
1  import keras
2  from keras.models import Sequential
3  from keras.layers import Dense, Dropout,
       Activation
4  from keras.optimizers import SGD
5  from keras.datasets import mnist
6  # Load Mnist data
7  (x_train, y_train), (x_test, y_test) = mnist.
       load_data()
8  print("Training data shape: ", x_train.shape)
9  # (60000, 28, 28) -- 60000 images, each 28x28
       pixels
10 print("Test data shape", x_test.shape)
11 # (10000, 28, 28) -- 10000 images, each 28x28
```

Code Example of MLP

```
12 # For Multi-Layer Perceptrons input dimension
13 # has to be a vector but the loaded MNIST
14 # images are matrices with dimension 28 x 28
15 # -> we need to flatten them (i.e. reshape
       them into vectors)
16 # matrix: 28 x 28 --> vector: 1 x 784
17 x_train = x_train.reshape(x_train.shape[0],
       784)
18 x_test = x_test.reshape(x_test.shape[0], 784)
19 x_train = x_train.astype('float32')
20 x_test = x_test.astype('float32')
```

Code Example of MLP

```
21 # normalizing data improves training
22 x_train /= 255
23 x_test /= 255
24 # one-hot encoding, i.e.
25 # Number 3 -> [0,0,0,1,0,0,0,0,0]
26 # there are ten different classes to predict
27 y_train = keras.utils.to_categorical(y_train,
       10)
28 y_test = keras.utils.to_categorical(y_test,
       10)
```

In the following slide we will actually build the model. Note: Hyperparameters in the input layer and hidden layer are the number of nodes $= 512$ as well as the chosen activation function ReLU.

Code Example of MLP

```
29 # building the model itself
30 model = Sequential()
31 model.add(Dense(512, activation = "relu",
       input_dim = 784))
32 model.add(Dropout(0.25))
33 model.add(Dense(512, activation = "relu"))
34 model.add(Dropout(0.25))
35 model.add(Dense(10, activation = "softmax"))
```

Code Example of MLP

```
36 # defining optimizer, compile model
37 sgd = SGD(lr=0.01)
38 model.compile(loss='categorical_crossentropy',
39                optimizer = sgd,
40                metrics = ['accuracy'])
41 # train model on training data
42 model.fit(x_train, y_train,
43           epochs = 10,
44           batch_size = 100)
45 # evaluate model
46 score = model.evaluate(x_test, y_test,
       batch_size = 100)
```

Code Example: Conclusion

It is important to preprocess data before feeding it into the model.
Implementing a model requires a multitude of hyperparameters to be set. They greatly determine the performance of the model.
To find the best hyperparameters (i.e. hyperparameter optimization) for a specific model and data set is not trivial but rather an art.
In playground.tensorflow.org you can change hyperparameters of a MLP as you wish and see the result immediately.

Cognitive Tasks

Counting animals in complex scenes (humans: age 2)

Face Recognition (humans: a few months old)

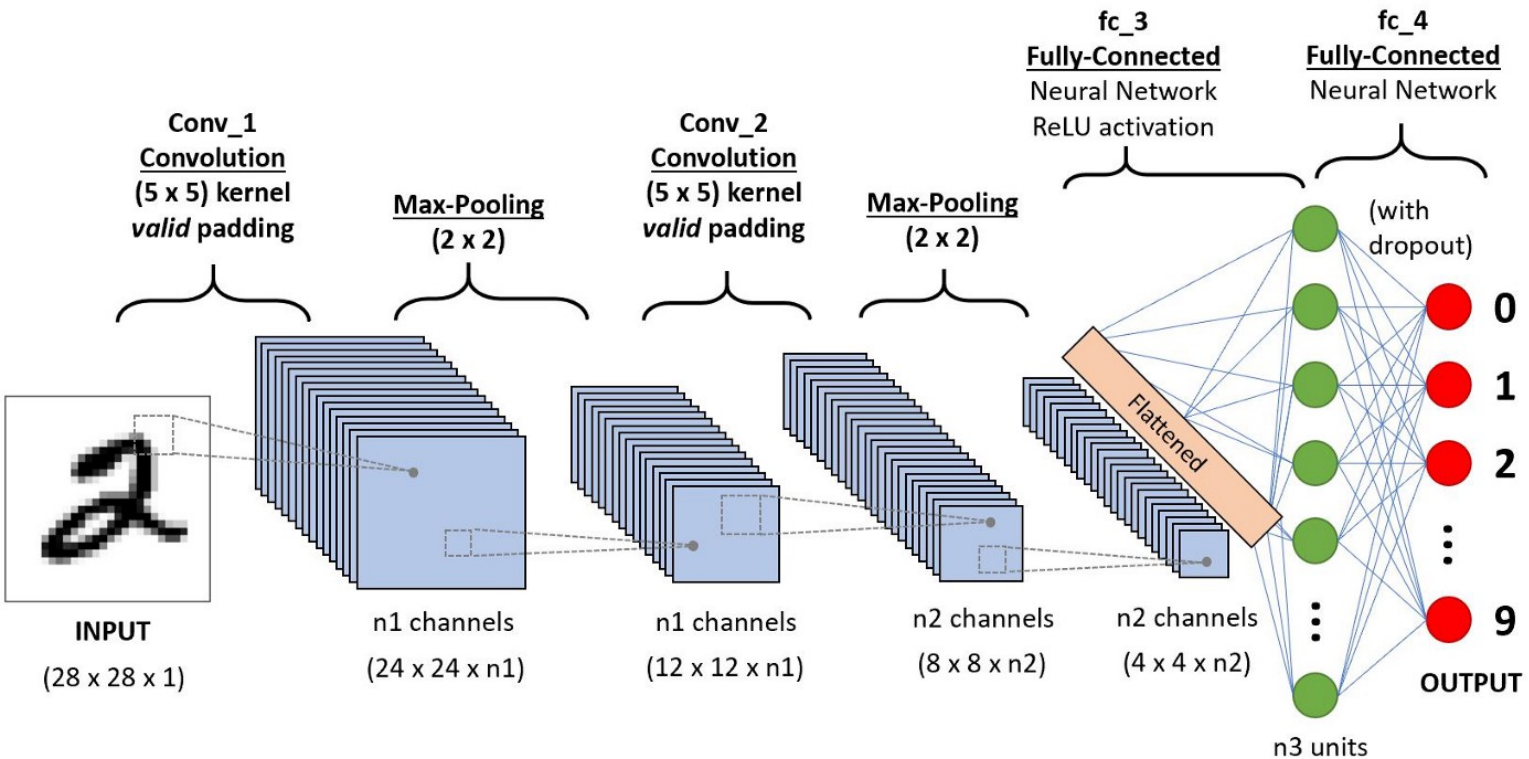
MLP as components of modern, more sophisticated model structures

In real world applications MLP are rarely used on their own due to their limited ability to abstract and learn features. They are rather used as a component of more complex neural networks like

- Convolutional Neural Networks (also known as ConvNet or CNN)
- Recurrent Neural Networks (RNN)

For instance: Most state-of-the-art image classification algorithms use CNN. At the end of a CNN model there is always at least one fully-connected layer to classify the learned features.

Convolutional Layers



References

[1] Barret Zoph and Quoc V. Le. "Neural Architecture Search with Reinforcement Learning". In: (2017). URL: https://arxiv.org/abs/1611.01578.