Artificial Neural Networks

So-called "**Convolutional Neural Networks**" are like cascaded filter banks, but with a non-linear function at the output of each filter and a constant offset, see for instance http://deeplearning.net/tutorial/lenet.html.
Its coefficients are called "weights".
The so-called "**(Artificial) Neural Networks**" simply use a weighted sum instead of the convolution, see
https://en.wikipedia.org/wiki/Artificial_neural_network
and
"An Introduction to Neural Computing", I. Aleksander, H. Morton, Chapman and Hall, 1990.

A nice application example for inage understanding can be seen in:
http://cs.stanford.edu/people/karpathy/deepimagesent/

Both types use several "layers" of cascading. If there are more than 3 layers, they are called "**Deep Neural Networks**", with "**Deep Learning**". These are current active research areas, for instance for speech recognition and

image recognition. A popular example is the MNSIT handwritten digit recognition, described for instance here:

https://machinelearningmastery.com/handwritten-digit-recognition-using-convolutional-neural-networks-python-keras/

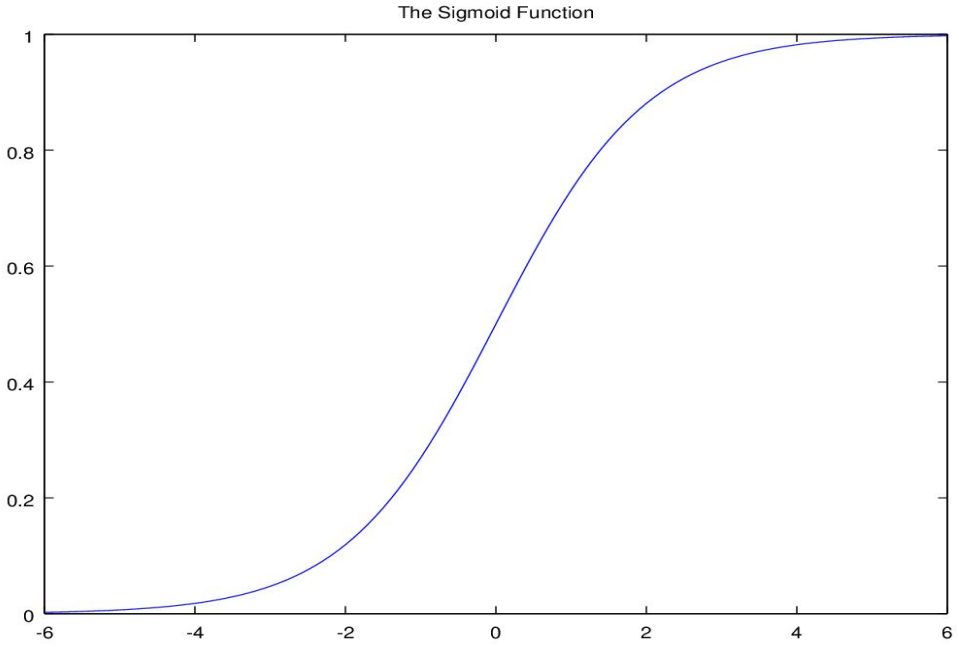
This example, as most Neural Network implementation, uses Python and its Library "Keras".

The non-linear function $f(x)$, also called "activation function" is often the so-called **sigmoid function**, see also https://en.wikipedia.org/wiki/Sigmoid_function which is defined as

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

We can plot it with ipython:

```
ipython3 -pylab
x=arange(-6,6,0.1)
y=1./(1+exp(-x))
plot(x,y)
title('The Sigmoid Function')
```



We see that it behaves like a “soft-limiter” function.
Its derivative is

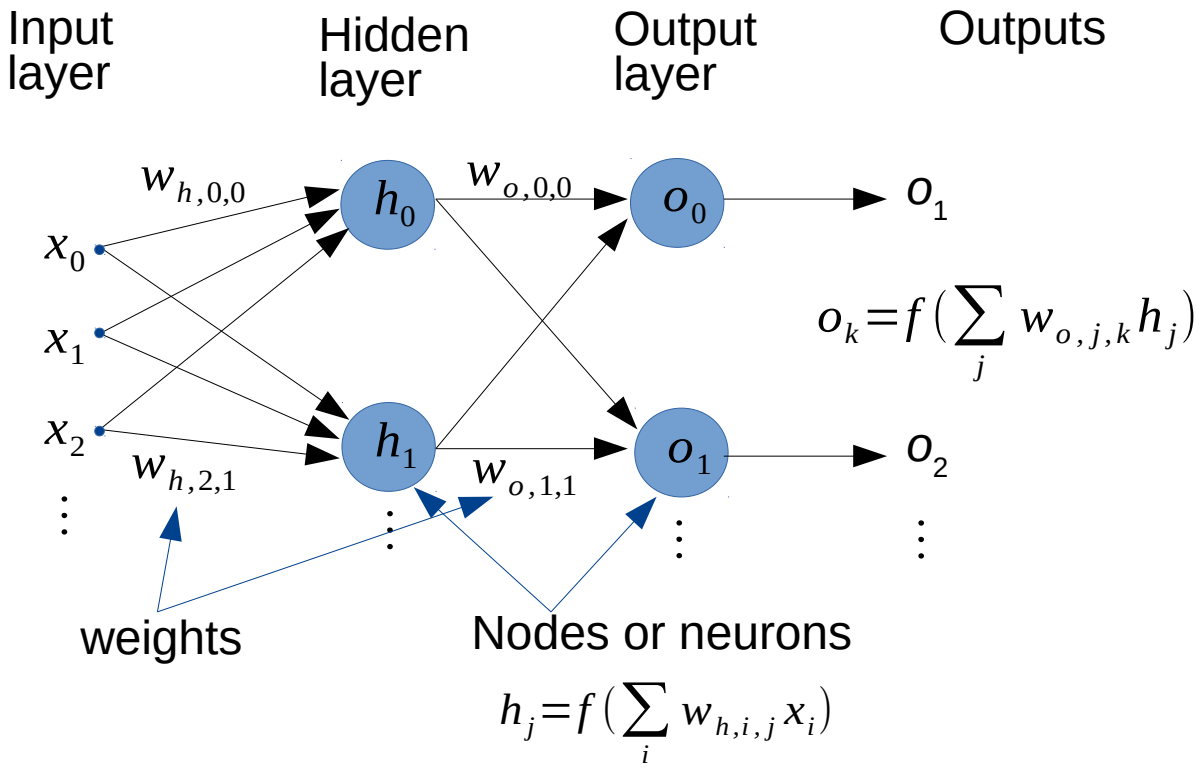
$$f'(x)=\frac{d}{dx}f(x)=\frac{e^x}{(1+e^x)^2}$$

Often used **alternative activation functions** are “tanh”, “relu” (Rectified Linear Unit, which returns

`x if x > 0, alpha * x if x < 0`

and more (see https://keras.io/activations/)

The following diagram shows a 3-layer artificial neural network,



Where $f(x):=\frac{1}{1+e^{-x}}$

In general we have several outputs in our neural network, **output k** is the non-linear function of a sum $s_{o,k}$,

$$o_k=f(s_{o,k})$$

The sum is the weighted sum from the hidden layer outputs h_j . We choose $x_0=h_0=1$, such that we also obtain a constant offset as part of the sum,

$$s_{o,k} = \sum_j w_{o,j,k} h_j$$

The output of the hidden layer h_j is again a non-linear function f of a sum

$$h_j = f\left(s_{h,j}\right)$$

The hidden layer sum is a weighted sum of input values x_i :

$$s_{h,j} := \sum_i w_{h,i,j} x_i$$

The output of our neural network depends on the weights $\boldsymbol{w}$ and the inputs $\boldsymbol{x}$ (we assume a fixed sigmoid function). We assemble the inputs in
the vector x which contains all the inputs,

$$\boldsymbol{x} = \left[x_{0,} x_{1,\ldots}\right]$$

and vector w which contains all the weights (from the hidden and the output layer),

$$\boldsymbol{w} = \left[w_{h,0,0}, w_{h,0,1}, \ldots, w_{o,0,0}, w_{o,0,1} \ldots\right]$$

To express this dependency, we can rewrite the **output k** as

$$o_k\left(\boldsymbol{x}, \boldsymbol{w}\right)$$

Now we would like to “train” the network, meaning we would like to determine the weights such that if we present the neural network with a training pattern in x, the output produces a desired value. For instance, if we present an

image with an object in it, the output indicates that the object was there with a desired output value. Hence we have **training inputs**, and **desired outputs** d_k (also called the "**target**"). We now use **optimization** to update the weights $\boldsymbol{w}$ to obtain outputs o_k as close as possible to the desired outputs d_k with a given input $\boldsymbol{x}$.

We use **Gradient Descent** for this optimization. We start with an error function delta,

$$\delta_k(\boldsymbol{x},\boldsymbol{w}) = o_k(\boldsymbol{x},\boldsymbol{w}) - d_k$$

An always positive Error function is its square,

$$Err_k(\boldsymbol{x},\boldsymbol{w}) = \frac{1}{2}\left(\delta_k(\boldsymbol{x},\boldsymbol{w})\right)^2$$

This is also called "**Loss Function**", and this particular choice the ean_squared_error.
To this error or function we can now **apply Gradient Descent** (see above eq. (1)), also called **Stochastic Gradient Descend** ('**sgd**') in our case of Neural Network training,

$$\boldsymbol{x}_{new} = \boldsymbol{x}_{old} - \alpha \cdot \nabla f(\boldsymbol{x}_{old})$$

in our case this becomes

$$\boldsymbol{w}_{new} = \boldsymbol{w}_{old} - \alpha \cdot \nabla Err_k(\boldsymbol{x},\boldsymbol{w}_{old}) \quad (2)$$

For the gradient ∇ we need to compute the derivative to each weight w_i of the weight vector $\boldsymbol{w}$. In our case we apply the chain rule (outer derivative times inner derivative),

$$\frac{d}{dw_i} Err_k(\boldsymbol{x},\boldsymbol{w}) = \frac{d}{dw_i}\frac{1}{2}\cdot\delta_k^2(\boldsymbol{x},\boldsymbol{w}) =$$

$$= \delta_k(\boldsymbol{x},\boldsymbol{w})\cdot\underbrace{\frac{d}{dw_i}\delta_k(\boldsymbol{x},\boldsymbol{w})}_{inner\ derivative}$$

$$= \delta_k(\boldsymbol{x},\boldsymbol{w})\cdot\underbrace{\frac{d}{dw_i}o_k(\boldsymbol{x},\boldsymbol{w})}_{inner\ derivative} \quad (3)$$

We compute this inner derivative first for the **output weights**, again with the chain rule,

$$\frac{d}{dw_{o,j,k}} o_k(\boldsymbol{x},\boldsymbol{w}) =$$

$$= \frac{d}{dw_{o,j,k}} f(s_{o,k}) = f'(s_{o,k})\cdot\frac{d}{dw_{o,j,k}} s_{o,k}$$

$$= f'(s_{o,k})\frac{d}{dw_{o,j,k}}\sum_{j'} w_{o,j',k} h_j$$

finally we get

$$\frac{d}{dw_{o,j,k}} o_k(\boldsymbol{x},\boldsymbol{w}) = f'(s_{o,k})\cdot h_j \quad (4)$$

We now plug this result (4) into eq. (3) and eq. (2), and obtain the update rule for the Gradient Descent for the output weights,

$$w_{o,j,new} = w_{o,j,old} - \alpha\cdot\delta_k(\boldsymbol{x},\boldsymbol{w}) f'(s_{o,k})\cdot h_j$$

which says: **update** = **alpha** times **output delta** times **output derivative** times its **input** h_j from the hidden nodes. Observe the only **local processing** for the update, which makes it suitable for massively **parallel processing**, e.g. in a Graphics Processing Unit!

For the hidden weights the inner derivative of eq. (3) becomes a little more complicated,

$$\frac{d}{dw_{h,i,j}} o_k(\boldsymbol{x},\boldsymbol{w}) =$$

$$= \frac{d}{dw_{h,i,j}} f(s_{o,k}) = f'(s_{o,k}) \cdot \frac{d}{dw_{h,i,j}} s_{o,k}$$

$$= f'(s_{o,k}) \frac{d}{dw_{h,i,j}} \sum_j w_{o,j,k} h_j$$

$$= f'(s_{o,k}) \cdot w_{o,j,k} \frac{d}{dw_{h,i,j}} h_j$$

$$= f'(s_{o,k}) \cdot w_{o,j,k} \frac{d}{dw_{h,i,j}} f(s_{h,j})$$

apply the chain rule:

$$= f'(s_{o,k}) \cdot w_{o,j,k} \cdot f'(s_{h,j}) \frac{d}{dw_{h,i,j}} s_{h,j}$$

$$= f'(s_{o,k}) \cdot w_{o,j,k} \cdot f'(s_{h,j}) \frac{d}{dw_{h,i,j}} \sum_i w_{h,i,j} x_i$$

finally we get

$$\frac{d}{dw_{h,i,j}} o_k(\boldsymbol{x},\boldsymbol{w}) = f'(s_{o,k}) \cdot w_{o,j,k} \cdot f'(s_{h,j}) \cdot x_i \quad (5)$$

We now plug this result for the hidden weights (5) again into eq. (3) and eq. (2), and obtain the **update rule** for the Gradient Descent for the **hidden weights**

$$w_{h,i,j,new} = w_{h,i,j,old} - \alpha \cdot \delta_k(\boldsymbol{x},\boldsymbol{w}) f'(s_{o,k}) \cdot w_{o,j,k} \cdot f'(s_{h,j}) \cdot x_i$$

We can simplify this update equation if we define a new "**back propagated delta**" term as

$$\delta_{h,k}(\boldsymbol{x},\boldsymbol{w}) := \delta_k(\boldsymbol{x},\boldsymbol{w}) \cdot f'(s_{o,k}) \cdot w_{o,j,k}$$

Hence we get

$$w_{h,i,j,new} = w_{h,i,j,old} - \alpha \cdot \delta_{h,k}(\boldsymbol{x},\boldsymbol{w}) \cdot f'(s_{h,j}) \cdot x_i \quad (6)$$

which says: **update** = **alpha** times **back propagated delta** times **derivative of hidden function** times its **input** x_i .

This algorithm is also called **Back-Propagation**.

We need to do this update for each output node k .

This is in principle the same rule as for the output nodes, **just with its corresponding input and output**.

This means: **Back-Propagation is** just a consequence of applying the **Gradient Descent** algorithm to Neural Networks. A popular alternative to Gradien Descent in Neural Networks it the algorithm "**Adam**", see also https://keras.io/optimizers/

Hence for all nodes we obtain a **"local" processing**. We just look at one node, call its input x_i , its output o , its weights w_i , and its desired output d . Then we obtain its output with

$$s=\sum_i w_i x_i$$

$$o=f\left(s\right)$$

if it is an **output node** the difference delta to the desired d is:

$$\delta=o-d$$

if it is a **hidden node** we use the "**back propagated**" δ from an output difference δ_o , weight to the output node w_o , and sum for the output node s_o :

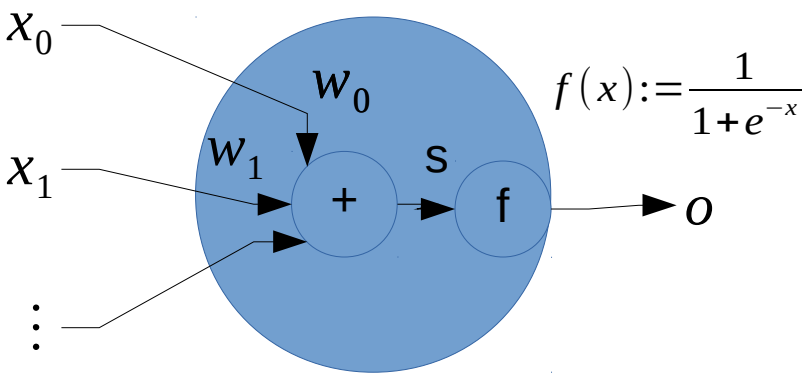
$$\delta=\delta_o\cdot f'\left(s_o\right)\cdot w_o$$

Then we get the **local weight update** as

$$w_{i,new}=w_{i,old}-\alpha\cdot\delta\cdot f'\left(s\right)\cdot x_i$$

This is depicted in the following diagram,

1 **Node**, also called **Neuron**



Weight update during training with output difference δ:

$$w_{i,new} = w_{i,old} - \alpha \cdot \delta \cdot f'(s) \cdot x_i$$

Python Example for the MNIST digit recognition:

```
python3 mnistexample.py
```

Convolutional Neural Networks

In Convolutional Neural Networks, each layer basically consist of **filter banks**, were fixed **bias** terms are added to each output, and the result is passed through a non-linearity. The filters are called "**Receptive Fields**", in analogy to the processing e.g. in the retina of the eye.

The filter coefficients and bias terms are then optimized according to a target function and a loss function, which computes the "distance" to the target function.

Python Keras Convolutional Neural Network Example

Most of Neural Network development is done in Python. It has powerful libraries for it, for instance "Keras" and "Theano" or "Tensorflow". Python Keras and Theano are libraries to compute and train neural networks. See also: https://keras.io/getting-started/faq/

To install it, we type in a terminal:

```
sudo pip3 install keras
sudo pip3 install Theano
```

edit `~/.keras/keras.json` (after running keras at least once) to replace "tensorflow" by "theano".

```
sudo apt install python3.h5py
```
(see also https://keras.io/backend/)

In our example, we want to detect a signal in a sequence, here a “ramp” function `np.arange(5)`, using a convolutional neural network.

In the Keras example program “keras_simpl_convnet.py”, we first define input (data) and desired output (target).
For the input, we prepend and append a few zeros to the ramp function, such that the neural network has to find the position of the ramp:

```
X=
np.hstack((np.zeros((1,9)),np.expand_dims(np.arange
(5),axis=0),np.zeros((1,9))))
```

We need at expand dimensions because Keras expects X with shape (batch, length, channels), with “batch”: training examples (here only 1 batch)
“length”: length of our signals (here 23)
“channels”: Channels of our data, for instance for stereo, here 1 channel.
The target signal has the same sizes on this case, because the “causal” network returns the same number of samples at the output as the input. We set it up as containing all zeros, except

for a 1 at the time of detecting the pattern:

```
Y = np.zeros((1,23))
   Y[0,16]=1 #Detecting the signal at its end
```

Expanding dimensions:

```
Y=Y.transpose()
   Y=np.expand_dims(Y, axis=0)
```

Then we set up a neural network model starting with:

```
model = Sequential()
```

A convolutional layer is added with:

```
model.add(Conv1D(filters=1, kernel_size=(8),
strides=1, padding='causal', activation="linear",
use_bias=False, kernel_initializer='glorot_uniform',
input_shape=(23,1)) )
```

A filter here is used to detect a pattern, similar to a matched filter in our lecture “Advanced Digital Signal Processing”, Slides 13 (see our Website).
The “kernel_size” is the size of our filters impulse response, here it can detect patterns of length 8 samples.
“strides=1” means there is no downsampling,
“padding=’causal’” means there is zero-padding of kernel_size-1 zeros before the beginning of our signal,

“activation=’linear’” means there is no non-linearity after the summation,
“use_bias=False” means the bias is zero,
“kernel_initializer” is the (random) initialization of the weights at the beginning of training,
“input_shape” is the size of the signal at the layers input.

The we need to “compile” the model and specify the used error function and optimizer:

```
model.compile(loss='mean_absolute_error',
optimizer='adam')
```

“loss” is the error function, here the mean absolute error between network output and the target,
“optimizer” is the optimization function to use to obtain the “best” weights, here “adam”, which seems to be the best.

Then we can train the model with:

```
model.fit(X, Y, epochs=5000, batch_size=1)
```

where “X” is the data or signal,
“Y” is the traget,
“epochs” is the number of iterations the optimizers uses,
“batch_size” is the number of examples we have,

here just 1.

After the optimization finished, we can use our model for prediction, here detecting the signal (usually a separate test signal is used, which differs from the training signal, but here we take the same for simplicity):

```
predictions=model.predict(X)
```

we can read out the resulting weights with

```
ww = model.get_weights()
```

The first index refers to the layer, here we only have 1, hence the index is 0, then the next index refers to the input dimension, the next to the output dimension (we have only one output neuron, hence the index is 0), and finally the channels index, again 0.
Hence our filter weights are:

```
weights=ww[0][:,0,0]
```

We can also save the weights in a “pickle” file:

```
with open("convnetweights.pickle", 'wb') as weightfile:
    pickle.dump(ww, weightfile)
```

Using pickle has the advantage that we can more easily analyse and modify the weights if we wish, compared to “`model.save_weights('weights.hdf5')`”.

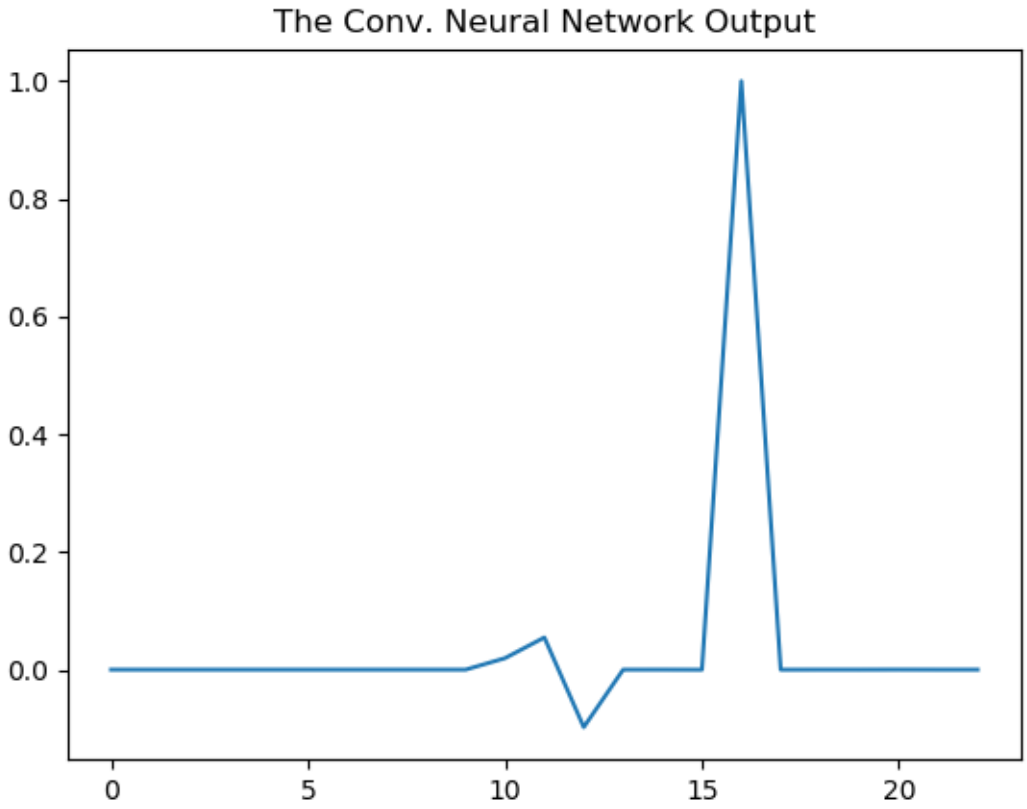
We can now plot the output of the neural network, the weights, and the input. The output and the weights have the same fomat as the input, (batch, length, channels), in our case the first and last index is 0,

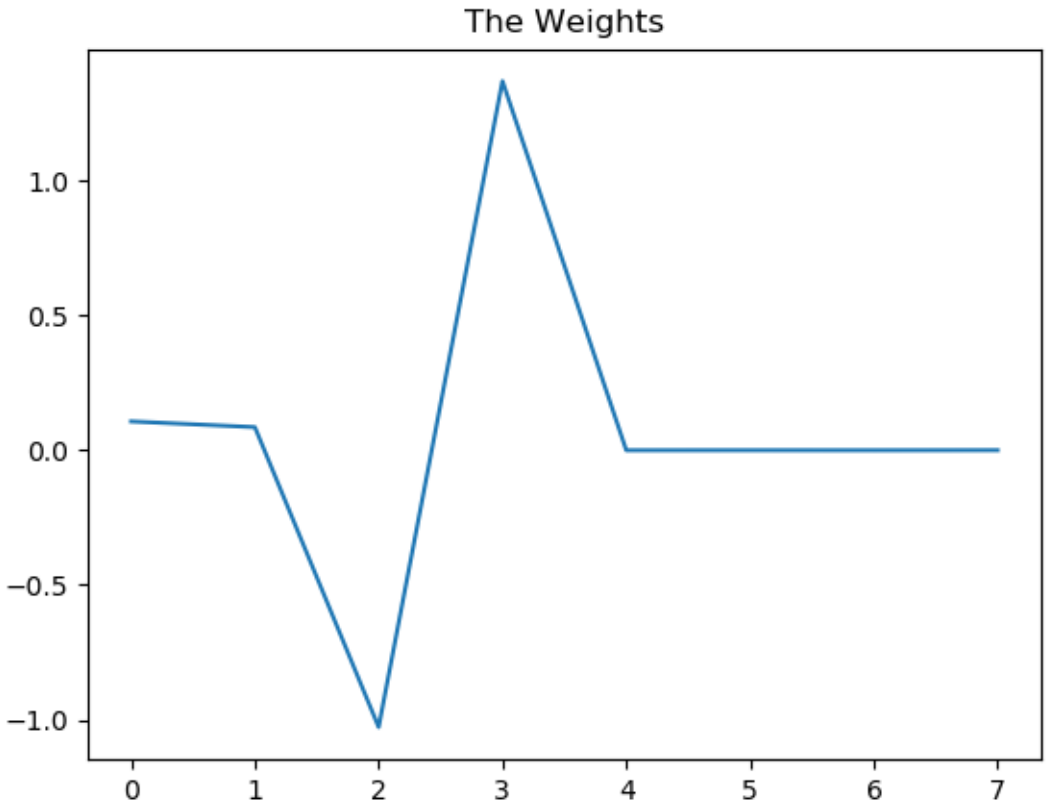
```
plt.plot(predictions[0,:,0])
plt.title('The Conv. Neural Network Output')
plt.figure()
plt.plot(weights)
plt.title('The Weights')
plt.figure()
plt.plot(X[0,:,0])
plt.title('The Input Signal')
plt.show()
```

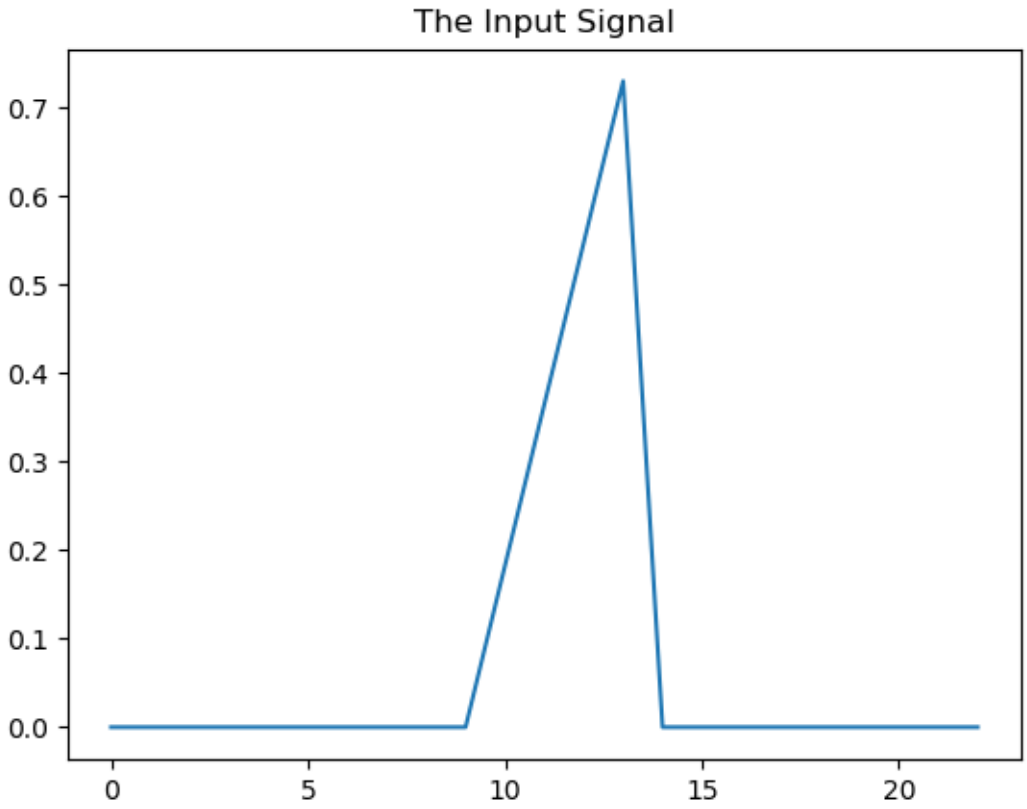
We start it with

```
python3 keras_simpl_convnet.py
```

After the 5000 iterations of the optimization, were we can observe the decline of the loss function, we get the following plots,


The Conv. Neural Network Output
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20


The Weights
1.0
0.5
0.0
−0.5
−1.0
0
1
2
3
4
5
6
7



Look at the weights plot, and compare it to a matched filter, which is used to detect a signal with a maximum possible Signal-to-Noise ratio (our lecture “Advanced Digital Signal Processing”, Slides 13). In the matched filter case, the weights should be the signal to detect, but time reversed (here a reversed ramp function). But our weights function looks quite different. This is because we specified a target function which only has a very narrow “peak”, and this is what we also observe at the output of our convolutional neural network. This may not have a maximum SNR, but a more narrow peak

at detection, which enables a more precise location of our pattern!

If we choose “**padding='valid'**” in our convolutional layer (as is the default), then no zero-padding of kernel_size-1 zeros before the signal starts is done, and hence the output of the convolutional layer will be accordingly shorter than the input. But we can obtain the same resluts if we remove accordingly kernel_size-1 (or filterlength-1) samples from the beginning of the target function with

```
gap=int(filterlength)-1
Y=Y[:,gap:,:]
```

Just remember that the output is accordingly shorter, kernel-size-1 samples in the beginning are missing compared to padding=”causal”.
We can let it run with

```
python3 keras_simpl_convnet_valid.py
```

Implementation using Python Pytorch

Pytorch is an alternative to Keras. Pytorch is used when more control of the neural network structure is desired. Our example for a convolutional neural network with causal padding (corresponing to `keras_simpl_convent.py`) can be executed with

python3 pytorch_simpl_convnet.py

Observe that the weights appear time-reversed compared to Keras. This is because Pytorch uses **correlation** instead of convolution for its filtering process (see https://pytorch.org/docs/stable/nn.html).

Implementation using a Dense Net

In audio processing, the Conv1D layer is useful when the entire audio signal is in memory. Then the convolution can be seen as shifting the time reversed impulse response (the weights) along the audio signal to produce the output of the convolution.
But when we would like to do real time audio processing, the audio signal arrives block by block from the sound card, and we should process them as they arrive. This can be seen as shifting the audio signal along the time reversed impulse response (the weights). We can implement this using a “dense” neural network layer, using a keras “Dense” layer. This layer has different defintions for the weights and the input, hence we need to translate them for this layer. The “Dense” layer basically implements a matrix multiplication, where the input signal is a row vector from the left and the matrix from the right

contains the weights (in our case this matrix is a column vector).
First the weights. They don't have the "channel" dimension, hence we need to remove it:

```
weights[0]=weights[0][:,:,0]
```

then we need to apply the time reversal from the convolution to it:

```
weights[0]=np.flip(weights[0], axis=0)
```

The we can set up a dense neural network model:

```
filtlen=len(weights[0])
model = Sequential()
model.add(Dense(units=1, activation='linear',
use_bias=False, input_shape=(filtlen,)) )
model.set_weights(weights)
```

Finally we need to transpose the input to be a row vector:

```
X=np.transpose(X,axes=(0,2,1))
```

Then we can loop over the signal. Here it still comes from the signal in memory, but it could also come from the sound card:

```
for n in range(siglen-filtlen):
   #cut out the current signal block:
```

```
Xblock=X[0,:,n:n+filtlen]
prediction[n]=model.predict(Xblock)
```

We can let the program run with
`python3 keras_simpl_densenet.py`

We can see that the output is indeed the same as from `keras_simpl_convnet_valid.py` .

Real-Time Online-Implementation of Convolutional Neural Networks

For a keras convolutional network it is assumed that the entire data is already in memory, but in real time, online processing, for instance for real time audio or wireless processing, it needs to be processed as the data arrives, sample- or block-wise.
Instead of the Convent "sliding" the filter along the samples, here we slide the samples along the filter, implemented using a Dense net.
For that we need to convert the weights data format! For our **Conv1D** layer, the **weights** have the following **format**. It is an array of an array:

[0: filter weights, 1: bias for first layer]
[filterlength, channels (subbands) in previous layer, neurons/filters in this layer]

The **weight format** for our desired “**Dense**” layer is the following:

[0: weights, 1: bias for first layer]
[total weights=filterweights * channels (subbands or neurons) in previous layer (time-reversed), neurons/filters in this layer]

We also need to observe that the convolution is a correlation with the flipped coefficients. In Our implementation we chose to keep the order of the weights, but flip the input signal instead. Hence the weigh conversion from Conv1D to Dense is mainly a dimensionality reduction:
`weights[0]=weights[0][:,0,:]`
The signal is input into a shifting buffer, such that it appears fliped in time, meaning the latest sample appears at index 0:

```
#"slide" buffer contest one sample "up":
signalblock[:,1:]=signalblock[:,0:-1]
#assign current value to "buttom".
#This flips the signal, since the newest value
appears at the lowest index:
 signalblock[0,0]= X[0,sampind]
```

Our example program reads in the weights from

the training of “keras_simpl_convnet.py”, from the file “convnetweights.pickle”.
We start the example program with
python3 keras_simpl_convdensenet.py

Observe: We get the identical output as from our Convnet, but now we get 1 output sample for each new input sample.