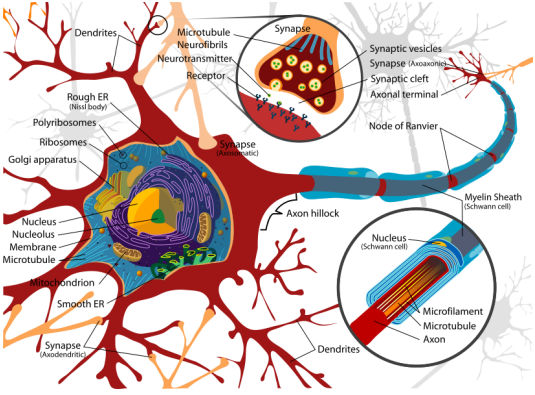
Deep Learning Basics

H. Hofbauer • M. Linortner • A. Uhl

Department of Artificial Intelligence and Human Interfaces (AIHI), Paris Lodron University Salzburg

Biological Neural Networks

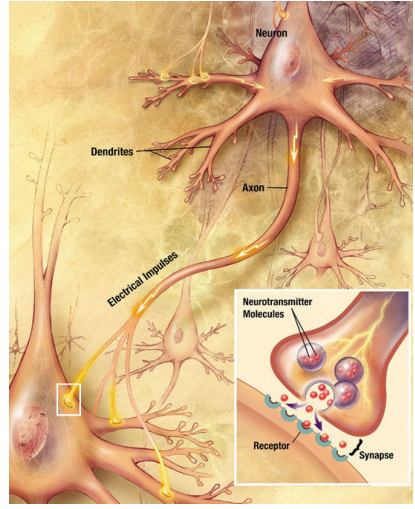
The brain is made up of a network of neurons.
The, very simplified, version of how they work is this:



A nerve cell, or neuron, gathers energy from it's dendrites, a treelike branching structure.
Signals reach the dendrites via synapse where a neuro transmitter is emitted and gathers on the receptors of the dendrite, which determines the potential (in an electrical sense) of the postsynaptic membrane.
This potential is an electrical signal that is transported along the dendrite to the soma, the body of the nerve cell.

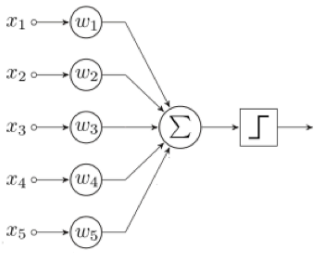
Biological Neural Networks II

The total of the signals reaching the soma are transferred to the axon hillock.
Here the neuron either fires or not (all or nothing activation) depending whether the total of the signals exceed an action threshold.
If the cell fires the signal is transferred along the axon to the dendrites of other nerve cell, thereby forming a neural network.



Rosenblatt Perceptron (1958)

The Rosenblatt Perceptron is built around a single neuron.
This model of a neuron is still used in modern ANNs/CNNs!



$$h(x) = \begin{cases} 1 \ if & w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_d \cdot x_d \geq \theta \\ 0 \ if & w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_d \cdot x_d < \theta \end{cases}$$

The equation can be re-written as follows including what it's known as the bias term: $x_0 = 1, w_0 = \theta$.

$$h(x) = \begin{cases} 1 \ if & w_0 \cdot x_0 + w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_d \cdot x_d \geq 0 \\ 0 \ if & w_0 \cdot x_0 + w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_d \cdot x_d < 0 \end{cases}$$

$$h(x) = \begin{cases} 1 \ if \ w^t \cdot x \geq 0 \\ 0 \ if \ w^t \cdot x < 0 \end{cases}$$

Inputs and outputs are binary, the weights are not.

Rosenblatt Perceptron and Learning I

The Rosenblatt Perceptron has only a single binary output and is thus a two class classifier (cat/dog; cat/not cat; food i like/dislike, ...).

- Learning is based on known input and output (supervised learning).
- The input $(x = x_1, \ldots, x_n)$ is fed into the perceptron and the output $h(x)$ is calculated.
- The actual output $h(x)$is compared to to the desired output y.
- If the two match we do not change anything (working as intended) if not we have to change the weights by some amount.
- This amount is usually called the learning rate (σ) and can change over time.

Now let's try to figure out how learning works with an actual example.

A toy example

x_1 (windy)	x_2 (cloudy)	y (nice day)
1	1	0
0	1	0
1	0	0
0	0	1

Let's start with weights of 1 ($w_0 = w_1 = w_2 = 1$ or $w = [1, 1, 1]$) and learning rate $\sigma = 1$.

If we input the first pattern we have:

$$\sum_{i=0}^{2} w_i \times x_i = \underbrace{1 \times 1}_{\text{bias: } w_0 x_0} + 1 \times 1 + 1 \times 1 \geq 0 \mapsto 1 \qquad \text{(desired output: 0).}$$

The sum should be less than 0 but is not, therefore we have to reduce the weights. Conversely if the output is 0 (then the sum is less than 0) and should be 1 then we want to increase the weights. Given that all values are wither 0 or 1, we can conveniently put this as:

$$\delta_w = y - h(x) = \text{adjusted by the learning rate}) = \sigma(y - h(x)).$$

In our case $\delta_w = 1(0 - 1) = -1$, and the weights thus become

$$w_0 \leftarrow w_0 + \delta = 1 - 1 = 0 \quad \dots \quad w = [0, 0, 0].$$

Rosenblatt Perceptron and Learning IV

Let us continue with the next pattern:

$$0 \times 1 + 0 \times 0 + 0 \times 1 \geq 0 \mapsto 1 \qquad \text{(desired output: 0)}.$$

Again we have $delta = -1$ to adjust the weights for the sum to go down.

But wait a second!

It isn't windy ($x_1 = 0$) so no matter w_1 that term would come out as zero. That means w_1 did not influence the results, and we have learned nothing about w_1 (and should thus not adjust it)!
Since the input values are binary, we can use them write the correct update step for the weights as:

$$w_i \leftarrow w_i + x_i \delta_w.$$

If $x_i = 1$ the term $w_i x_i$ did influence the sum and the weight is adjusted by $1\delta_w = \delta_w$ as previously, but if $x_i = 0$ then it did not and the adjustment also becomes 0.

Rosenblatt Perceptron and Learning V

Now we can simply run the pattern to update the weights until all the patterns we have are correctly handled. Weights are update from one line to the next.

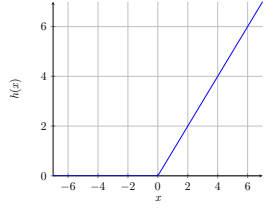
pattern	windy	cloudy	nice day	w_0	w_{windy}	w_{cloudy}	$h(x)$	δ_w
1	1	1	0	1	1	1	1	-1
2	0	1	0	0	0	0	1	-1
3	1	0	0	-1	0	-1	0	0
4	0	0	1	-1	0	-1	0	1
1	1	1	0	0	0	-1	0	0
2	0	1	0	0	0	-1	0	0
3	1	0	0	0	0	-1	1	-1
4	0	0	1	-1	-1	-1	0	1
1	1	1	0	0	-1	-1	0	0
2	0	1	0	0	-1	-1	0	0
3	1	0	0	0	-1	-1	0	0
4	0	0	1	0	-1	-1	1	0

The last rotation for all patterns did not change anything so we have learned all the inputs.

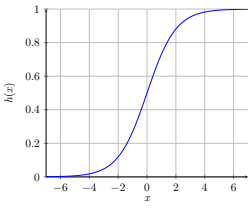
Extensions on the Rosenblatt Perceptron I

The first, and a relatively simple extension is the change from binary input to real-valued input.

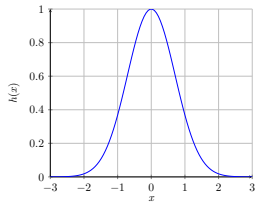
- Inputs are often normalized to be between $[0, 1]$.
- $\delta_w = \sigma(y - h(x))$ still works, the rate of adjustment simply becomes lower the closer the actual result is to the desired result.
- $w_i \leftarrow w_i + x_i \delta_w$ also still works, the adjustment of the is coupled to how much influence the input had on the result.
- $h(x)$ is (often) changed from the Heaviside function to a real value activation function. Typical ones include:



rectified linear unit



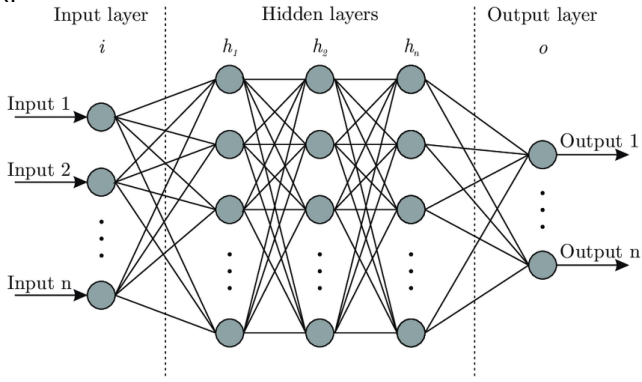
sigmoid



Gaussian

Extensions on the Rosenblatt Perceptron II

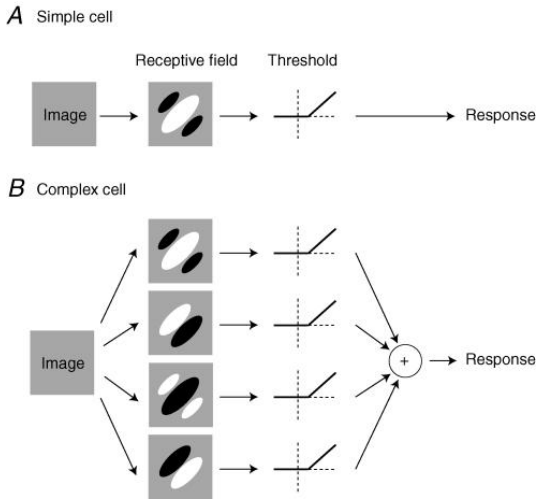
Extension to a sequences of interconnected neurons, i.e., from a single neuron to a brain like network.



- Using multiple layers of perceptrons is called a 'deep neural network' or a 'multi layer perceptron'.
- This breaks the simple learning for perceptrons.

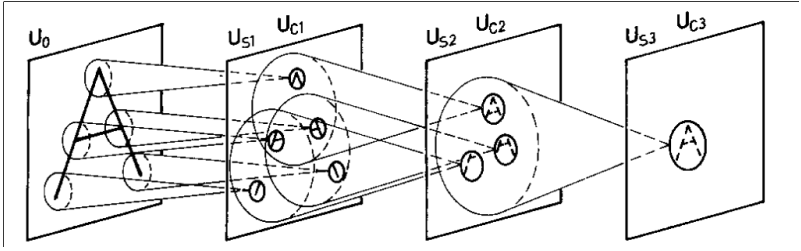
Simple and Complex Cells (Hubel and Wiesel 1962)

Hubel and Wiesel define simple cells as having distinct antagonistic regions in their receptive fields.
They define complex cells as any cell that was not simple. They reported that complex cells achieved position invariance within their receptive field: *they would respond to a stimulus of the appropriate orientation regardless of position within the receptive field.*



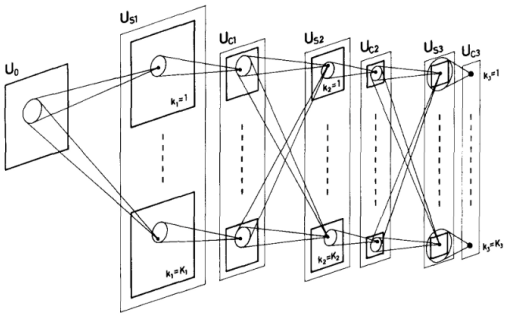
Neocognitron (Fukushima 1980) I

Fukushima used this idea to improve upon a prior network (the cognitron). The main problem of the cognitron was position dependence of the stimulus.



- Each neuron has a limited input area and every neuron of a layer learns the same pattern. A single layer corresponds to a single pattern but over the whole image (equivalent to s-cells).
- Multiple such patterns are learned, e.g. edges, corners, and so on.
- Layers further on combine inputs of previous layers to form more complex patterns (equivalent to c-cells).

Neocognitron (Fukushima 1980) II



This presents some huge problems

- Requires enforcement that each layer learns the same pattern at every position.
- Each specific pattern requires a whole layer of the same dimension as the input image.
  For example if edges with an orientation every 10° are learned the size increases by 18.
- A complex network like this has a huge number of parameters, making learning harder and requiring more input/output patterns to properly learn.

Neocognitron (Fukushima 1980) III

- This is a direct predecessor of the modern convolutional neural networks.
- Complexities surrounding learning and enforcing the same pattern over a whole layer were problematic.
- A solution to the data expansion, which is still in use today, are so called pooling layers.
  Since position invariance is a goal the layers could be reduced in size by only using the strongest response (max) in a given area, say 2×2 pixels. This max-pooling reduces the data by a factor of 4 and introduces slight position invariance.
  - Too large an area should not be used as the relative position of patterns is still important.
  - But repeat applications at later stages in the network can be used for further data reduction and slight introduction of position invariance in larger patterns.

Backpropagation Algorithm (Rumelhart 1986) I

The simple perceptron learning breaks down when using multiple layers. This was a huge problem for a long time.
A solution for feed forward networks was presented in 1986.

- A feed forward network processed one input at a time with no temporal information.
- A recurrent neural network has a temporal aspect, that is prior inputs can affect the current one.

The solution comes int he form of the backpropagation algorithm (backprop).

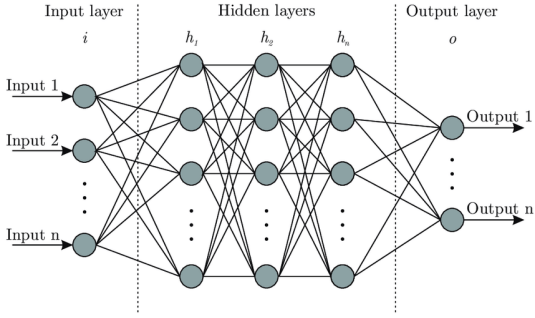
- The Backpropagation Algorithm is older.
- Has been developed multiple times independently.
- The Rumelhart paper showcased it for the learning of a neural network.

Backpropagation Algorithm (Rumelhart 1986) II

The backprop is a gradient descent method which adjusts the weights based on the difference in desired outcome and actual outcome.

- The input vector $x = (x_1, \ldots, x_s)$ is applied to the first layer of the network.
- It is passed through multiple (L) layers, h^l, $l \in \{1, \ldots, L\}$.
- Finally the last layer h^L produces an output vector $\hat{y} = (\hat{y}_1, \ldots, \hat{y}_t)$.
- The activation function for each layer is assumed to be the same for all neurons in that layer f^l.
- A cost function (or loss function) C calculates the error based on the outcome of the network $\hat{y}$ and the desired output y.

Backpropagation Algorithm (Rumelhart 1986) III



Assuming all layers are fully connected we get the outputs $o^l = (o_1^l, \ldots, o_m^l)$ of a layer h^l like this:

$$o_1^l = f^l\left(\sum_{j=1}^{n} w_{1j}^l o_j^{l-1}\right)$$

$$\vdots$$

$$o_m^l = f^l\left(\sum_{j=1}^{n} w_{mj}^l o_j^{l-1}\right)$$

where the previous layer had n outputs, w_{ij}^l is the weight between neuron j from h^{l-1} $(= o_j^{l-1})$ to neuron i in h^l.
Note that $o^0 = x$.

Backpropagation Algorithm (Rumelhart 1986) IV

The idea of the backpropagation is the same as for a perceptron, we calculate the error $C(y, \hat{y})$ for a given input/output pattern x, y. Then we adjust the all the weights, i.e. the weight matrices for each layer, such that the actual output $\hat{y}$ is moved closer to the desired output.
Obviously this is not as straight forward as for the perceptron, but we know the following:

- The derivative gives us the direction of change of a function.
- The partial derivative of a function for one of it's parameters gives us the direction of change for the function regarding that parameter.
- A composite function, $f(g(x))$, can be derived by using the chain rule $(f(g(x))' = f'(g(x))g'(x)$.

Our network is a composite function for which we can utilize this as long as f^l are differentiable.

Backpropagation Algorithm (Rumelhart 1986) V

In practice this is what it looks like.

Lets assume we want to adjust the weights of output node i, that is $w_i^L = (w_{i*}^L)$.

- The error of this node is $\delta_i^L = \frac{\partial C}{\partial o_i^L}(\hat{y}, y)$.
- This is the change required in the output of o_i^l, to facilitate that we have to change the weights of node i.
  - Again we know the differential of the activation function f^l gives us the required direction of change.

$$\delta_{w_{ij}^L} = \frac{\partial f^L}{\partial w_{ij}^L}(\delta_i^L).$$

- again we make the change of weight dependant on a learning parameter (σ) and the strength of the signal, i.e. no signal implies no change.

$$w_{ij}^L \leftarrow w_{ij}^L - \sigma \delta_{w_{ij}^L} o_j^{l-1}.$$

Backpropagation Algorithm (Rumelhart 1986) VI

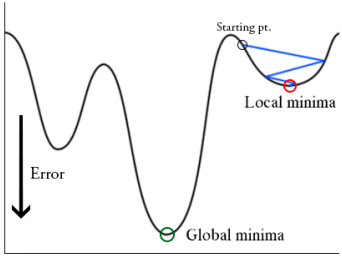
Now we want to adjust the weights of a node which is not in the output layer.
The problem here is that the output is dependant on the number of neurons in the next layer as the required change there is passed back to this neuron.

- $\delta_i^l = \sum_k \delta_{w_{ki}}^{l+1} w_{ki}$
  That is, the change of required is the sum of changes in the next layer propagated back to this layer. The influence is adjusted by the weight. Meaning if a weight is low we can't affect the neuron in the next layer very much, and in turn a required change in that neuron should not affect this neuron very much.
- $\delta_{w_i j}^l = \frac{\partial f^l}{\partial w_{ij}^l}(\delta_i^l)$, as before
- $w_{ij}^l \leftarrow w_{ij}^l - \sigma \delta_{w_{ij}^l} o_j^{l-1}$, again as before.

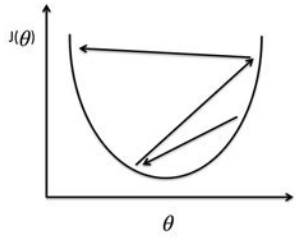
This now let's us train more complicated networks.

Backpropagation Algorithm (Rumelhart 1986) VII

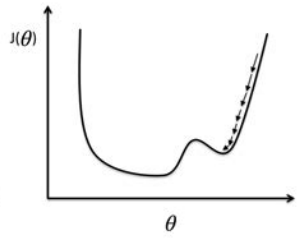
The backpropagation is not without problem though.



Depending on learning rate and starting point the gradient descent can be trapped at a local minimum.



Large learning rate: Overshooting.



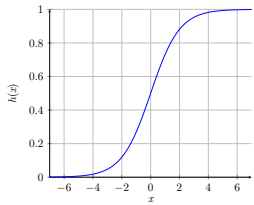
Small learning rate: Many iterations until convergence and trapping in local minima.

A solution to this can be to suppress rapid change in direction (momentum) or to adaptively scale the learning rate. A combination is adaptive moment estimation (Adam).

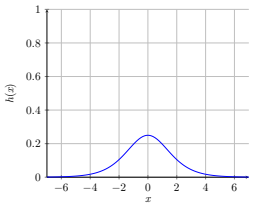
Backpropagation Algorithm (Rumelhart 1986) VIII

Another problem is 'vanishing gradients' or stuck activation functions, which are related problems.
We have seen that the backpropagation is relative to the derivative of the function, so if the output of the function is small then little change, if any happens.

Sigmoid



f

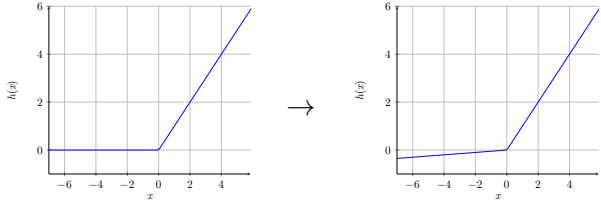


f'

Here the maximum of the derivative is 0.25, now if multiple layers use this activation function the derivative is repeatedly applied, so after two layers the maximum would be $\frac{1}{16}$ and so on.

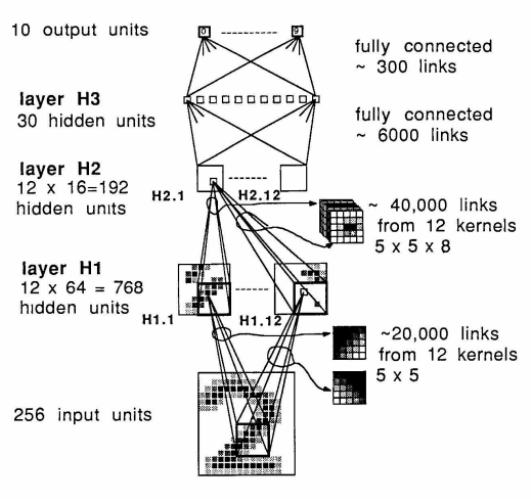
Backpropagation Algorithm (Rumelhart 1986) IX

- Stuck activation functions can happen when the weights of are adjusted to a specific range in the domain. Imagine large negative weights as at the inputs of a rectified linear unit (ReLU), the output will always be zero.
- Since we make changes to the weights relative to the output of the function the weights will no longer be adapted and the neuron will be stuck in a non-activation mode.
- A solution to this can be to prevent constant outputs, in the case of the ReLU the solution is a leaky ReLU which has a slight slope in the negative.



ReLU Leaky ReLU

Convolution on Zip Codes (LeCun 1989) I



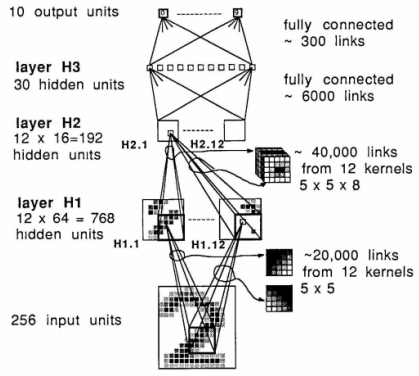
- Handwritten digit recognition for zip code recognition for the US postal service.
- "backpropagation can be used on fairly large tasks with reasonable training time"
- "connections are constrained to the same weights ... non-linear subsampling convolution"

Convolution on Zip Codes (LeCun 1989) II

This is basically an extension of the neocognitron.

- The complicated forcing of each connection in a layer (corresponding to a filter) to learn the same pattern is simplified by using a single connection type which is convoluted over the previous layer.
  - Learns single pattern per output layer.
  - Reduces number of weights since only one filter is used (biases are still per neuron).
- The complicated learning is simplified by using backpropagation.
  - Easy to implement.
  - Relatively fast convergence.
- The input data is vastly expanded, once per filter (like in the neocognitron). And a similar approach is used to reduce the data. Locality is discarded but introducing a step size of where the filter is applied. In modern CNN parlance this is called stride. Both stride and max-pooling are still in used to reduce data.

Convolution on Zip Codes (LeCun 1989) III

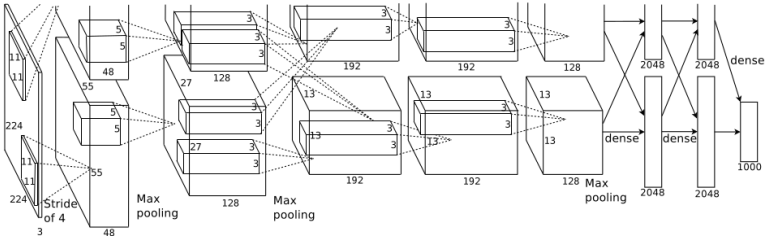


This is basically a modern CNN composed of

- input layer
- Convolutional layers with stride / and or reduction layers (max-pooling).
- towards the end the convolution is fed into a fully connected layer structure to perform the classification.

Basically the convolutional layers learn patterns, first simple (as in simple cells) but becoming more complex the farther along the data progresses (complex cells).
The fully connected layers in the end perform the classification based on the patterns found in the convolutional steps.

ImageNet, AlexNet and GPUs (Krizhevsky 2012) I

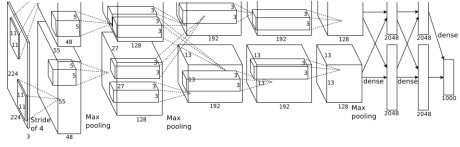


- AlexNet is basically an extension of the LeCun's Network.
- Heavily used GPUs to accelerate learning.
  But was not the first to do so.
- Won the ImageNet Challenge (visual object recognition database of 14 million images) by a large margin (10% better than the next contender).
- This brought Neural Networks back into the mainstream.
  "*Suddenly people started to pay attention, not just within the AI community but across the technology industry as a whole.*" (The Economist, June 2016)

ImageNet, AlexNet and GPUs (Krizhevsky 2012) II

The work combined various techniques, and evaluated them showcasing the extent to which they improved the results.

- Used exclusively ReLUs as activation functions to combat vanishing gradients of sigmoid type functions. These result in faster learning.
- Training on two GPUs with cross memory access. The network was to large to fit a single GPU at the time.
  - Two identical parallel networks with limited crossconnection (third layer and fully connected layers).



- They found that local response normalisation, no required in ReLUs as it mainly is used to combat saturation, improves the results.
- Pooling was performed in an overlapping manner.

ImageNet, AlexNet and GPUs (Krizhevsky 2012) III

A problem with training was overfitting. The CNN has 60 million parameters to sort the 14 million images into 1000 categories.

Overfitting

"*The production of an analysis which corresponds too closely or exactly to a particular set of data, and may therefore fail to fit additional data or predict future observations reliably.*" (Oxford living dictionary)

Overfitting is a constant problem for machine learning, and usually results if the model has too many parameters in comparison to the training data.
In practice this is a problem that can often not be solved. A way around that is to use a pretrained network on similar data. Then only the classification part of the network (the fully connected layers) have to be retrained for the actual classes.

ImageNet, AlexNet and GPUs (Krizhevsky 2012) IV

In this case overfitting was prevented by

- Data augmentation, i.e. the multiplication of available data by:
  - Sampling, patches of smaller images are extracted from a larger image and used as individual input, can be used to force translation invariance.
  - Mirroring of images or samples
  - Adjustment of color intensity and luminance but keeping the objects intact, to allow to learn to recognize objects in images recorded under different illuminations.
- Dropout learning, which means setting the output of some neurons (randomly) to 0 for a training step.
  - Prevents co-dependencies between neurons as a given input can not be relied upon.
  - Forces the network to learn more robust patterns, thereby reducing specificity and improving generalization.

Types of Networks I

There are different types of networks, in nomenclature but also in how they work. Some we know already, others will be introduced in other courses (like GANs). Some we will not cover, but it still is good to know about them.

Perceptron The original, not used on it's own but as building block for more complex neural networks.

Multilayer Perceptron (MLP), feed forward neural network (NN or FFNN) This is a combination of many Perceptrons, grouped into layers, with usually only one layer connecting to the next.
They can be *fully connected*, which means that a single neuron (perceptron) is connected to all the neurons in the previous layer.
These can be used on their own or a classification part in more complex networks (see CNNs).

Radial basis function networks RBFs are essentially multilayer perceptrons which change the activation function from monotonic to functions which are even with a shift (e.g. a gaussian function).

Types of Networks II

Convolutional Neural Networks These are in essence feed forward networks, but they are not fully connected (the input of a convolutional filter is limited). Furthermore, the weights are not trained separately for each neuron, but are shared in the filter.
These can be used on their own but are usually followed by some other network, typically a fully connected feed forward network for classification.

Recurrent neural network Are networks which are not purely feed forward. This essentially means that RNN has a memory of the previous step. This is typically used for input which has a time component, such as speech or handwriting.

Long short-term Memory Networks In essence RNNs with more memory. In reality it's a RNN with a different repeating module structure (the memory) which combats vanishing gradients over time to allow longer efficient memory.

Types of Networks III

Autoencoders (AE) Is a sort of MLP which is usually symmetric around a central code layer, the goal is for the first part to encode the input, culminating in the code layer. The second part then tries to decode the code to reconstruct the input. That is, the Autoencoder learns an encoding and decoding scheme specific to the data.

Generative adversarial networks (GAN) Consists of two networks working against each other in a zero-sum game, i.e. the gain of one network is the loss of the other. A generative network tries to generate content, e.g., a picture from a description, whereas the adversarial network tries to determine if the image is a real or generated image, i.e. it tries to foil the generative network.

Deconvolutional networks The reversal of a, typical convolutional, neural network. The input is used, to combine an ensemble of patterns, basically inverted convolutions, into a greater image or signal. Typically used as generative networks in GANs.

Excursion - Unsupervised Learning I

- We are dealing with *supervised learning* in this course:
  - The answer is known
  - The cost function is usually a measure of the difference between the known answer and the given answer
- What if we don't have the answer?
  - This is called *unsupervised learning*
  - Either the answer must be constructed
    - The principle is: input equals output
    - Either directly, typically with an autoencoder
    - Or indirectly (also with AE) but the input is programmatically changed.
  - Or the answer must assume something
    - This is probabilistic learning (a distribution is assumed)
    - Bolzman machines, for example, are stochastic extensions of Hopfield networks
  - Or there is no answer.
    - Learns patterns such that they can be reconstructed from parts (patterns are attractors in feature space).
    - Hopfield networks (a simple type of RNN)

Hopfield networks (idea only)

- based on Hebbian learning (neurons that fire together wire together)
- input is binary typically with values -1 and 1 for simpler math.
- no self connection ($w_{ii} = 0$ fixed)

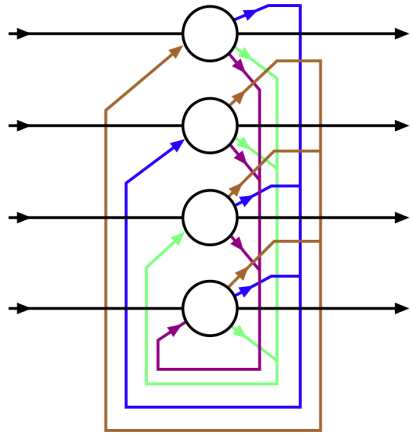
For P inputs x^p, weights are set to

$$w_{ij} = \sum_{p=0}^{P} x_i^p x_j^p$$

The weight matrix is always symmetrical ($w_{ij} = w_{ji}$)
Output of a neuron i is

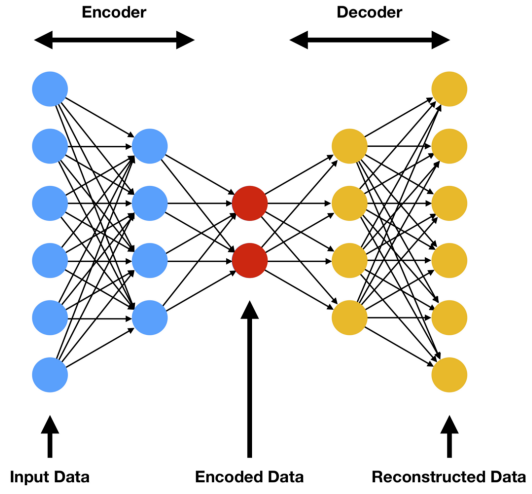
$$O^i(t) = \begin{cases} 1 & \text{if } \sum_{n=0}^{N} w_{ij} O^j(t-1) > \theta \\ -1 & \text{if } \sum_{n=0}^{N} w_{ij} O^j(t-1) < \theta \end{cases}$$

the threshold θ is usually 0

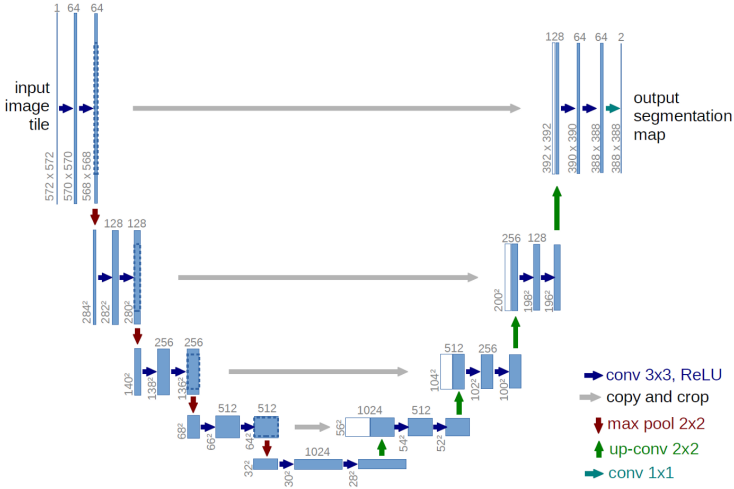


Autoencoders (idea only) I

- The encoder is a CNN which reduces the input to features in the 'latent' or 'feature' space.
- The output of the encoder is the encoded data.
- The encoded data is feed into a decoder network, basically an inverted CNN using deconvolution.
- The output should be a reconstruction of the input.
- Due to compression (the encoder should reduce the data) the output will likely be faulty.
- Minimization of the reconstruction error is our cost function.

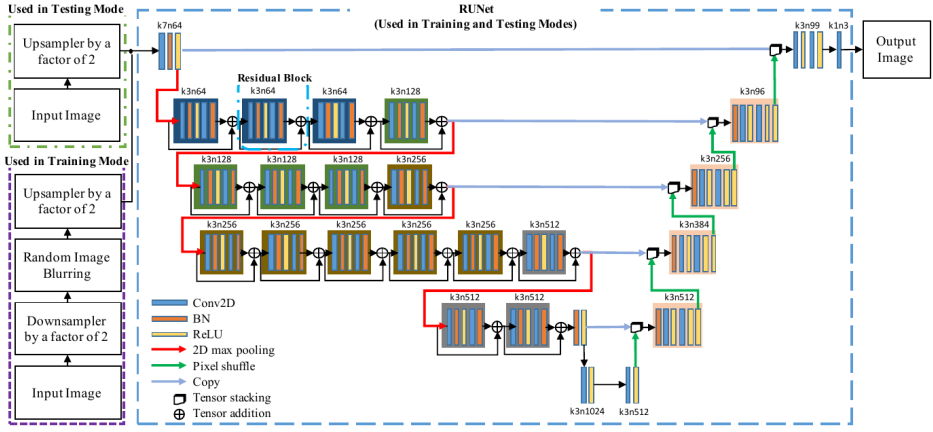


Autoencoders (idea only) II



- In practice structures like the U-Net are usually used.
- The decoder should revert the encoder
- Data is feed from encoder to decoder to simply, i.e., speed up, learning

Autoencoders (idea only) III



The RUNet is an example of the use of an /U-Net based autoencoder to do superresolution (the reconstruction of a high quality image from a low quality source). Similar ideas can be used for denoising etc.

Frameworks for Deep Learning I

There are various frameworks for CNNs for different programming languages.

- Tensorflow
- Keras
- Sonnet
- MXNet
- GLuon
- DL4J
- ONNX
- Caffe2
- Chainer
- Microsoft CNTK
- PyTorch

We will use PyTorch for this course `https://pytorch.org`.

Getting PyTorch I

We have seen from the historical examples that GPUs facilitated deep networks due to the computational effort required to train them. Most frameworks for deep learning can work with CPU and GPU.
GPUs implemented more and more general purpose code execution architecture starting with programmable shaders roughly around 2001. These gave rise to GPUs for general purpose computing and consequently to frameworks facilitating exactly that.
The currently largest, longest running SDK and API for general purpose GPU (GPGPU) computing is CUDA (compute unified device architecture) from Nvidia which was launched in 2006.
Another important one is OpenCL (Open Computing Language) launched in 2009 with the goal to provide a framework for writing programs which can execute across different platforms: CPUs, DSPs, FPGAs and GPUs.

Getting PyTorch II

PyTorch uses CUDA for execution of code on the GPU, specifically cuda toolkit which must match the installed CUDA version. Here's how you find the version:

- `nvidia-smi`

```
:~$ nvidia-smi
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.87.00    Driver Version: 418.87.00    CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
.............
```

- CUDA Version file

```
:~$ cat /usr/local/cuda/version.txt
CUDA Version 10.1.243
```

- `nvcc --version` if the nvidia-cuda-toolkit is installed

Getting PyTorch III

With this you can get an installation from the PyTorch website in an easy way by going to: `https://pytorch.org/get-started/locally/`

PyTorch Build	Stable (1.10)	Preview (Nightly)	LTS (1.8.2)	
Your OS	Linux	Mac	Windows	
Package	Conda	Pip	LibTorch	Source
Language	Python	C++ / Java		
Compute Platform	CUDA 10.2	CUDA 11.3	ROCm 4.2 (beta)	CPU
Run this Command:	`conda install pytorch torchvision torchaudio cudatoolkit=10.2 -c pytorch`			

Getting PyTorch IV

Normally the PyTorch page will suggest to use Anaconda for installing the *cuda* and *PyTorch* libraries.
Anaconda creates an environment separate from the global python environment where you can install specific version of libraries without conflict regarding the system installation.
Anaconda can support different environments which can be switched between easily to get the development environment desired (this is quite similar to *venv* (`python -m venv`) but with a global (for the user) storage and single command interface. Often CNNs requires specific package version of cuda and other libraries so this is suggested.
Alternatively pip can be used for the installation which will usually do the install on a user level.

Exercise: Getting started with PyTorch

Material: Workshop Course Materials

Topics:

- Getting started with PyTorch
- Install a pre-defined network
- Adapt data (image) to the network.
- Use data and network (classification)

Exercise: Handwritten Digit Detection

Material: Workshop Course Materials

Topics:

- Create your own (simple) neuronal network.
- Train the network.
- Evaluate on test data.
- Using the PyTorch data loader.

Building Blocks of CNNs I

In PyTorch there are a number of layer types of which we will give the most important here.
The primary layer types are

- Convolutional Layers
- Linear Layers (fully connected layers)
- Activation layers (the activation functions are split from the above layers so that they can easily be switched out)
- Pooling layers
- dropout layers
- Normalization layers (batch normalization)

Dimensions in PyTorch

Dimensionality in PyTorch refers to the layer. That is a 2D convolution layer, can be of the third dimensions if applied to multiple layers. The 2D simply means that each layer is two dimensional, an image for example.

The following parameters primarily describe the behaviour of a convolution. Groups is primarily a PyTorch description, in literature a parallel filter bank would be shown.

padding Extending the region to prevent loss at the signal border (extend by zero, mirror, continuation).

size The input size of the convolution per layer.

dilation Spread of the filter input.

stride The filter is not taken at every position but every *stride* positions, can limit data expansion.

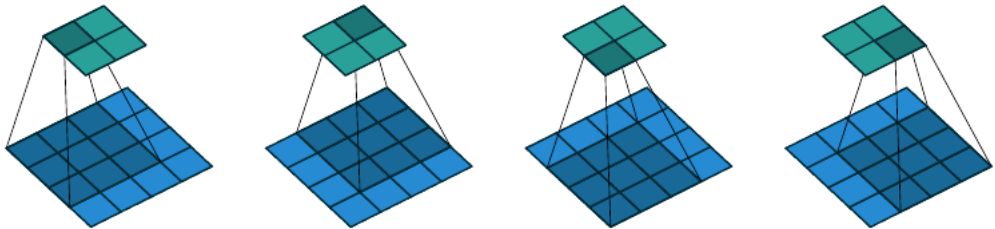
Convolutional Layers II

groups Affects input dimensions by splitting input into groups (output must of course also be divisible by the group count).
Example: 4 input layers with a 3x3 size would results in a convolution with total input size of 3x3x4.
If groups were two it would result in two different convolutions where each would see only *group* layers, so two 3x3x2 convolutions.

The following images are take from—*A guide to convolution arithmetic for deep learning* by Vincent Dumoulin and Francesco Visin, 2018—which is good read if more information about convolution is desired.

Convolutional Layers III

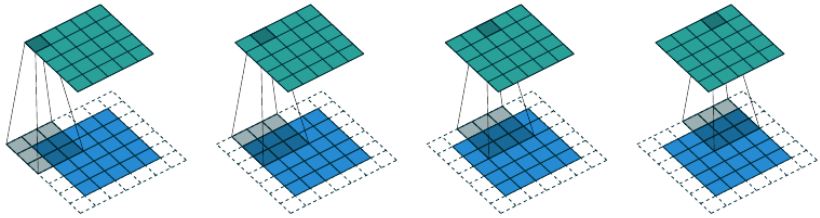
Regular convolution of a 3×3 filter with stride 1 and dilation 0.



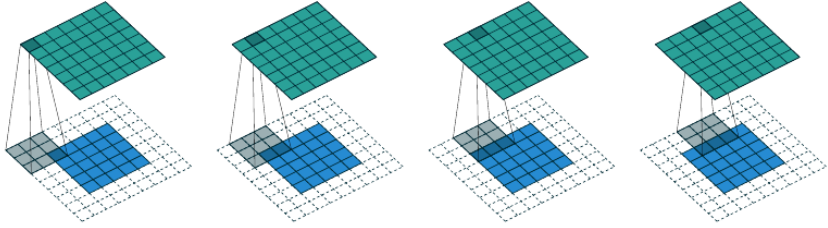
Notice the reduction in size of the output (2×2 from the 4×4 input).

Convolutional Layers IV

Half padding (half the filter size, rounded down) results in a constant signal size.

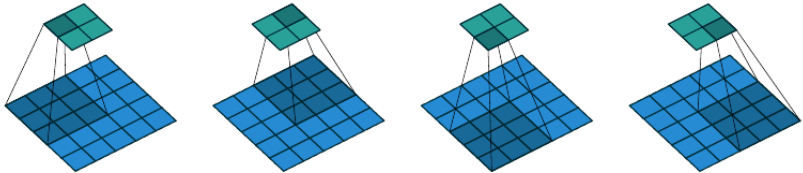


Full padding (filter size -1) leads to data expansion.

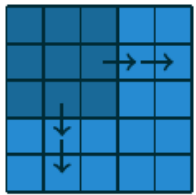


Convolutional Layers V

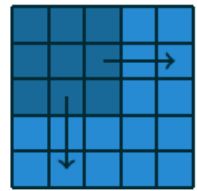
Stride is the step size of where the filter is applied, or alternatively the number of filter outputs skipped in relation to the regular form. Here we have stride 2.



Stride 1:

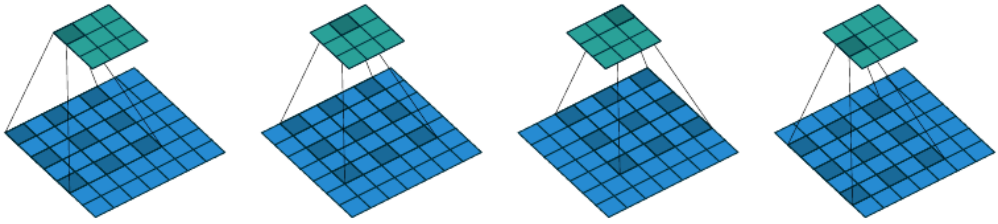


Stride 2:

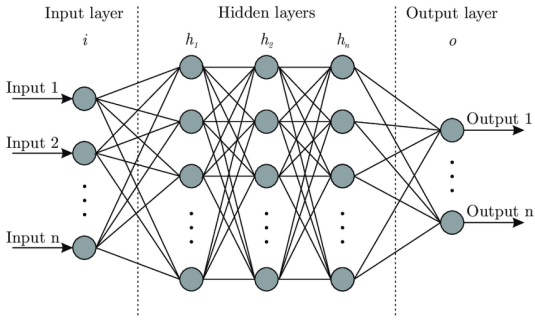


Convolutional Layers VI

Dilation increases the 'spread' of the filter while not increasing it's size.



Linear/Fully Connected Layer



The Layers depicted here are fully connected layers

Each input dimension is connected to each output dimension (i.e. fully connected).

Nota Bene: In PyTorch the linear layer expects a 1-dimensional input, in case of more dimensions in the layer prior `torch.flatten` should be used to convert them to a 1-d signal.

Activation Layers

Activation functions are split from the regular layers so that they can easily be switched out.
We have already discussed some of the common activation functions, PyTorch includes those, variations on them and a lot more besides. Just as a brief overview (in case of doubt start with the bold ones):

- Normalizing activation functions:
  - $[0:1]$ **Sigmoid** (`torch.nn.Sigmoid`)
  - $[-1:1]$ tangens hyperbolicus (**tanh** `torch.nn.tanh`), a smooth approximation of the sign functions (`torch.nn.softsign`)
- Thresholding functions:
  - Positive activation: **rectified linear units** (`torch.nn.ReLU`), a leaky version to prevent zero-gradient (`torch.nn.LeakyReLU`) and a smooth approximation (`torch.nn.Softplus`)
  - settable threshold: `torch.nn.Threshold`

Pooling Layers

Pooling layers are used to reduce the amount of data in the network by grouping neighbouring activations.
This conveniently also increases the reach of later convolutions (complex cells) and reduces reliance on specific locations.
The most common used is max pooling, which simply takes the highest activation in a given pool (`torch.nn.MaxPool[1,2,3]D`.
The pooling relies on the same parameters as convolutions, *size* gives the area from which the maximum is taken, *stride* gives the step size of how often that area is sampled from the input signal and *dilation* can spread the samples of the input area out.

Dropout Layers

Dropout layers (`torch.nn.Dropout`) can help prevent overfitting. This is done by randomly setting outputs to zero during training:

- Works with a probability p. During evaluation this is set to 0.
- Each neuron is set to 0 with probability p.
- To keep the input vector size intact each output which is not zeroed is scaled (to $\frac{1}{1-p}$).
  During evaluation $p = 0$ and no scaling is performed.

Normalization layers (batch normalization) I

The idea of data normalization is well known (and should be done for input data). The idea is to conform the features to zero mean and unit standard deviation. With a random weight initialization of around 0 this puts the outputs into the non-saturated area of typical activation functions. Which improves adaptation rate as we avoid low/zero gradients. For input data this has to be done once (the data does not change).

Batch normalization extends this idea to deeper layers by normalizing the output of a layer to the same $N(0, 1)$ distribution. The idea here is that internal covariance shift is reduced.

Internal covariance shift is the change of output distribution due to a shift in the layer parameters. However, the following layer adapted itself to the previous output distribution. This has an impact on learning.

Normalization layers (batch normalization) II

Basically the mean and variance over a batch (with size m) are calculated

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i,$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2.$$

This is then used to convert the output to the normalized output

$$\hat{x}_i = \frac{x_i - \mu_B}{\sigma_B}.$$

Just one problem, this totally negates any adaptation of bias and stretching of the distribution.

Normalization layers (batch normalization) III

To allow that in an explicit way post normalization the following output is produces by a batch normalization

$$y_i = \gamma \hat{x}_i + \beta,$$

where γ and β are learnable parameters.
Batch normalization *usually* has the following benefits:

- Allows for higher learning rates as it prevents fast deterioration to low/zero gradients
- Makes sigmoid-type activation functions more usable (for the same reason)
- Has regularization properties which can sometimes have similar effects as dropout.

Exercise: Handwritten Digit Detection continued

Material: Workshop Course Materials

Topics:

- Implement a CNN from ‘scratch‘ (LeNet5).
- Application of different CNN layers.
- Optimizer (Adam vs. SGD).

Data and Training

We have already seen that data is important for training.

- Over fitting can happen.
  That is the network can specifically learn each of the data items in the training set if we have to little data. Generalization suffers.
- Learning only a subset.
  If the data is biased only a subset of the task will be learned. For example if an object is always in the lower right side of the image the network might learn to disregard all other areas, leading to narrow localizations etc. In essence generalization suffers.
- Failure to converge.
  If the network is not complex enough to deal with the given data it might fail to converge, switching from one solution to another. This results in incomplete learning of the task.

Data Augmentation I

Data Augmentation can help produces more data from what we have and also help the network to generalize.

- Advantage: reduce chance for over-fitting.
- Advantage: help the network to generalize (by location, mirror, zoom etc.).
- Disadvantage: Have to generate the images, usually on the CPU.
- Consider: Not all augmentation methods can/should be used.

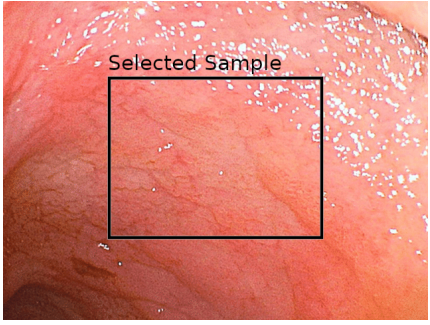
Transformations are found in `torchvision.transforms` (or `torchaudio.transforms` and so on).

Data Augmentation II

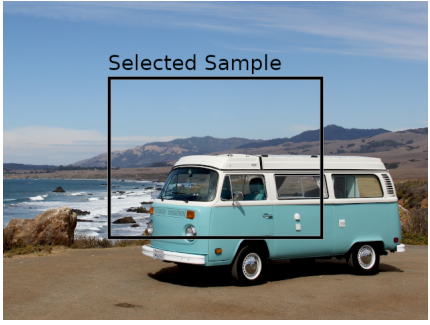
The first option simply to use a sample the required size and crop it from a fixed place. The sample size is defined by the input of the CNN

Original with central sample

Mucosa (duodenum)



VW Bus

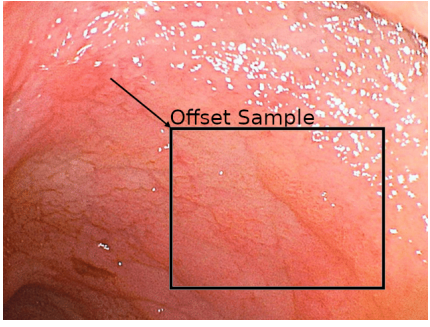


Data Augmentation III

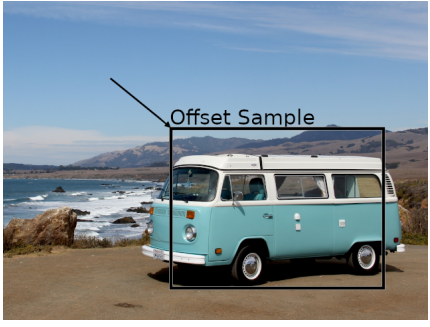
Or samples can be chosen with an offset.
This can be fixed if for a training run of samples with randomized offsets can be used.
This can be useful to strengthen the learning of location invariant patterns.

Sample with Offset

Mucosa (duodenum)



VW Bus



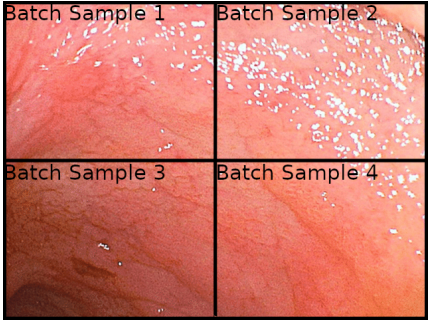
Data Augmentation IV

To greatly increase the number of samples batch sampling can be used. This can be done either regularly (as shown here) or in a random pattern (i.e. with a random offset).

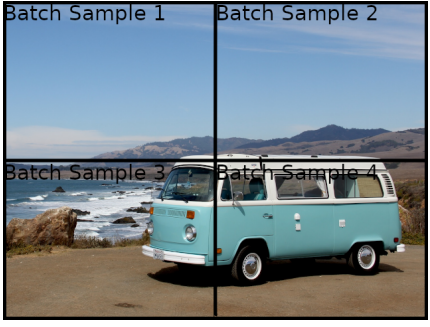
This drastically increases the number of samples, care has to be taken to make sure the sample actually contains what we are learning.

Batch Sampling (grid)

Mucosa (duodenum)



VW Bus

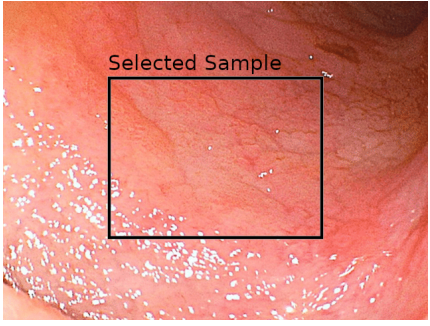


Data Augmentation V

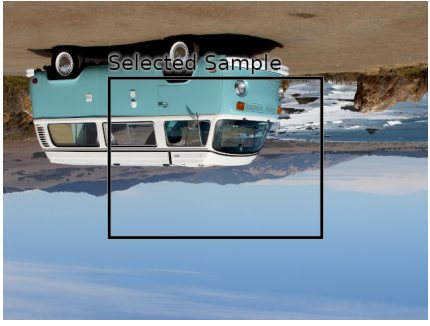
Mirroring along either or both the horizontal and/or vertical axis can be used.
Improves the data but also forcing to learn the other version of non-symmetrical filters.

Mirror vertical + horizontal

Mucosa (duodenum)



VW Bus

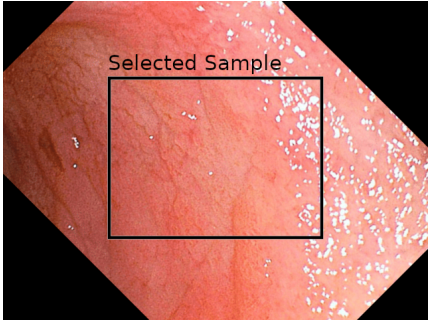


Data Augmentation VI

Like mirroring the rotation of the image strengthens the learning of general features (in this case rotation invariant features.
Unlike mirroring (and rotation with multiples of 90°) this however comes at the cost of information as certain sample positions are no longer available.

Rotation 45°

Mucosa (duodenum)



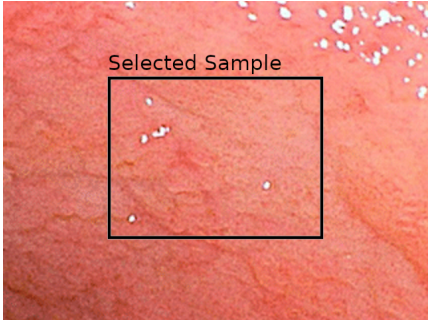
VW Bus

Data Augmentation VII

Zooming in or out (scale change) can increases the scale invariance of the network by forcing the CNN to learn different higher level patterns (complex cells).

Zoom (x2)

Mucosa (duodenum)



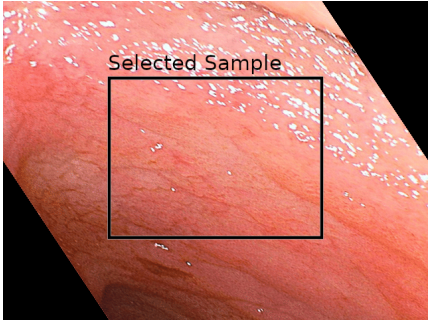
VW Bus

Data Augmentation VIII

Shearing or other transformations can potentially enhance the resistance of the CNN to different distortion types. While they can be used to increase the pool of training data it should be carefully evaluated if the transformation makes sense to include.

Shearing

Mucosa (duodenum)



VW Bus

Data Augmentation IX

In addition to the spatial transformations there are also a variety of image property transformations which mainly help to allow to learn information despite different recording conditions. But some like a transformation in hue can also help to not fixate filters a any given color layer.
Typical are changes in:

- brightness
- gamma
- contrast
- saturation
- hue
- blur
- sharpening

Data in Training I

- We have seen how we can get the most out of the data we have.
- But, how do we use the data to:
  - Train a neural network
  - Training and designing a neural network
  - Compare different algorithms
  - Evaluate a data set

Nota Bene: The nomenclature used here for the different datasets is not well defined and will vary widely in literature, however the data set used for training the network is almost always called training set. The meaning of test set on the other hand can have different meanings.

Training a neural network

For the training of a neural network the available **data set** is split into two parts:

- The **training set** which is used to for training the neural network. That is repeated forward passes and error back propagation to tune the free parameters of the network.
- The **validation set** which is used to validate the neural network.

If the network is trained purely on a single data set the end result would almost certainly be over-fitting. This is the reason that a validation set is used to see if over-fitting is happening. A typical indicator is that the error rate on the validation set will increase (beyond the usual fluctuation).
It is of utmost importance that the training set and the validation set are kept disjointed in every sense except what is supposed to be learned.

Data in Training III

Example

We want to learn to recognize a VW Bus and the dataset consists of VW Buses and other cars. If the VW Buses are all red and blue and the other cars are gray and black it is possible that the NN would learn the colors. If we split the test/validation set such that the red VW Buses are in the training set and the blue VW Buses are in the validation set, can be caught easily.

Example

We want to learn to recognize a disease based on samples from patients (multiple per patient). We should not have images from one patient in the validation as well as the test set. The NN might learn some other property of that patient and use that instead of markers for the disease.

The later example also shows a problem here, we might not be aware of what other distinguishing features are contained in the data. We can only be mindful of the problem and do the best we can.

How to split the data into training and validation set?

> *It is difficult to give a general rule on how to choose the number of observations in each of the three parts, as this depends on the signal-to-noise ratio in the data and the training sample size. (The Elements of Statistical Learning)*

While there is an answer:

> *For cross-validation stopping we computed the optimal split between training and validation examples and showed for large m [ed. m is the number of changeable parameters]that optimally only $r'_{opt} = \frac{1}{\sqrt{2m}}$ examples should be used to determine the point of early stopping in order to obtain the best performance. (Amari et al., Asymptotic statistical theory of overtraining and cross-validation)*

This is of a more theoretical nature, as the authors acknowledge: "Nevertheless note, that this asymptotic range is often inaccessible in practical applications due to the limited size of the data sets".

Data in Training V

As an Example let us look at refinement learning (more on that later) of the classification layers of the LeCun net for handwritten number recognition.

Example

The final two layers (the fully connected layers) have a total of m=6300. The optimal number of image in the validation set would be

$$\frac{1}{\sqrt{2 * 6300}} \approx 0.009 = 0.9\% .$$

Assuming a dataset of 1000 images that would be 9 image for the detection of early stopping. Given that there are 10 different digits this is quite obviously too few in this case.

Luckily the authors also provided this insight:

> *... the gain in the generalization error is small if we perform early stopping, even if we have access to the optimal stopping time.*

Data in Training VI

Quite frequently a 80/20 split is encountered in practice. This is usually not motivated and it could be speculated that this is done purely because the 80/20 ratio is known from other topics.

- So in practical terms it is suggested to use most of the data for training and a decent amount for early stopping, i.e., prevention of overfitting.
- If possible stick to the $r'_{\text{opt}} = \frac{1}{\sqrt{2m}}$.
- But make sure the early stopping set has a decent coverage over the input domain.

Data in Training VII

The tasks of

- **Training and designing a neural network**
- **Compare different algorithms**

are very similar in concept. The goal is to train various neural networks and compare them to find which is best suited to the task.
This means we have to split our data again:

- The **development set** which is used to for training the neural networks.
- The **benchmark set** compare different architectures to see which is best.

Data in Training VIII

The benchmark set is kept to perform a final evaluation of the different architectures, the development set is split further.
If training a single neural network it is split into a training set and a validation set as per the previous discussion.
However, if we do not have a given architecture we want to compare design a network and again need a sort of benchmark set.

- The training set as for a single neural network.
- The validation set as for a single neural network.
- The **test set** which is used to do a final validation of the neural network such that over-fitting of architecture/parameters can be prevented.

Data in Training IX

The considerations for splitting a test set or benchmark set are similar as for the split between training set and validation set.

> *If we call N is the number families of recognizers, h_{max} the largest complexity of those families, f the validation set size and g the training set size, the ratio $\frac{f}{g}$ scales like $\sqrt{\ln N / h_{max}}$. (Guyon, A scaling law for the validation-set training-set size ratio)*

More specifically

$$\frac{f}{g} = \sqrt{\frac{C \ln(\frac{N}{\alpha^2})}{h_{max}}},$$

where N, h_{max}, g and f are as above. $C = 1.5$ is the constant of the Chernoff bound, $alpha$ is the risk of being wrong (typically 0.05).

Data in Training X

The tricky part is usually to get h_{max} right, however:

- The authors give an approximation for $h_{max} \approx \frac{F}{3}$, where F is the number of free parameters, for neural networks trained with back-propagation and early stopping.
- Otherwise h_{max} it is the Vapnik-Chervonenkis dimension (VC-dim):
  - If the weights come from a finite family (i.e. 32/64 bit representation in computers) and with sigmoid or sign activation functions, then the VC dimension is at most $h_{max} \approx F$.
  - Generally an upper bound is $h_{max} \approx l F \ln F$ with l the number of layers in the neural network.

A note on the confidence

A frequent problem is the adherence to the $\alpha \approx 0.05\%$, i.e., the 5% confidence interval. This is usually the error pro comparison (this case of N comparisons, each to the optimal solution). A 5% error for each does not mean a 5% error for all.

Example

Example for $N = 10$ and $\alpha = 0.5\%$. The chance to make at least 1 error in 10 comparisons is

$$1 - (1 - \alpha)^N = 1 - 0.95^{10} \approx 1 - 0.599 = 0.401$$

That means that there is a 40% chance to make at least a single error in our 10 comparisons!.

It's ok to choose $\alpha = 0.05$ but be aware of the implications!

Evaluate a data set

This is a bit of a separate problem, where the network should be evaluated on the whole data set while also use it for training.

This is most often used to compare to non-learning algorithms which were evaluated on the whole dataset.

The problem is that what is used in the evaluation should not be used in training. If we stick to this, we either can't train the network or not evaluate it.

The solution is a trade-off (time and computational power). We split the data set into k **folds** so that

- The folds are disjointed (in the learning sense)
- The folds are as big as possible.
- Every combination of $k-1$ folds is sufficient for training.

Data in Training XIII

The k folds do not need to be of the same size, but it simplifies things if they are roughly equal.
The way to perform the evaluation is called k-**fold crossvalidation** and requires to train the neural network k times.

$D \leftarrow$ Dataset
$F \leftarrow$ partition of D into k folds
for $i = 0$ to k, $E_i \in F$ **do**
    Evaluation Set $\leftarrow E_i$
    Development Set $\leftarrow D \cap E_i$
    Train the NN_i on the Development Set
    $R_i \leftarrow$ evaluate E with trained NN_i
end for
Full Evaluation $\leftarrow \bigcup_{i=0}^{k} R_i$

Training and Parameters I

We have already seen how training looks (gradient descent by back-propagation) and we have seen how to augment and split data for training.
What is left is how to apply the data in backpropagation.

epoch The whole training set was once used for forward and backwards propagation.

iteration Usually means one back propagation.

gradient descent / batch gradient descent All entries in the test set are run through the network, the total error is calculated and then propagated back.

stochastic gradient descent Each subset of the test set is run through the network (forward propagation), the error is calculated and then propagated back. This includes mini-bnatch gradient descent.

mini-batch gradient descent A number of entries (the **batch size**) from the test set is forward propagated, the error is accumulated. After the whole batch is done, the combined error is propagated back.

Training and Parameters II

Some remarks:

- Batch gradient descent sometimes refers specifically to the case that the batch size is equal to the size of the trainings set. In this case mini-batch gradient descent refers to batch sizes smaller than than.
- Testing for early stopping should usually be done based on iterations rather than epochs (as it reflects the change in the network). Usually it is however a good idea to not test more than once per epoch to put the available data to good use. Overfitting by presenting different data is also not a problem.
- Gradient descent is usually more computationally expensive than stochastic gradient descent. The upside is that it is less likely to be caught in a local minima. That is batch gradient descent usually gives better results but stochastic gradient descent is faster.
- It is a good idea to shuffling the training set for mini-atch gradient descent to prevent batch overfitting (not useful for batch size equal to training set size obviously).

Refinement training I

In practice it is unlikely that a CNN is designed from the ground up as it takes

- a lot of time (for training primarily), and
- very large mounts of data.

Especially the required amount of data to fit millions of parameters without overfitting is usually the problem. Remember AlexNet had 14000000 images (14M) and had serious problems with overfitting.
The most practical approach is usual to refine an existing CNN.
This approach uses an already trained CNN from literature as a base, alleviating the need to construct the CNN and train it (for the most part).

Refinement training II

- Find a trained CNN which is
  - as small as possible (but still sufficient for you use case).
  - trained on similar images than what you use.
- Use the trained convolutional layers.
  As this is the part that was trained to detect objects in the data.

EITHER Retrain the classification part (usually a/the fully connected layer(s) following the convolutional layers).

OR Train new classification layer(s) if they existing ones do not fit your requirement (mostly output size).

Since only a relatively small part of the CNN has to be retrained/refined it is usually sufficient to have a smaller data set, especially if it can be extended with augmentation. Care still has to be taken not to overfit the classification layers.

Exercise: Texture classification

Material: Workshop Course Materials

Topics:

- Use predefined CNN.
- Adapt classification layers.
- Write your own data loader.
- Refine the network (classification layer).