HLRS

# Can we use deep learning for a small dataset?

## HLRS Summer School Trust and ML

Dr. Khatuna Kakhiani

26.7.- 28.7.2023, HLRS, Universität Stuttgart

# Can we use Deep Learning for a small dataset?
# Part 2

**HLRS Summer School Trust and ML**

Dr. Khatuna Kakhiani

26.7.- 28.7.2023, HLRS, Universität Stuttgart

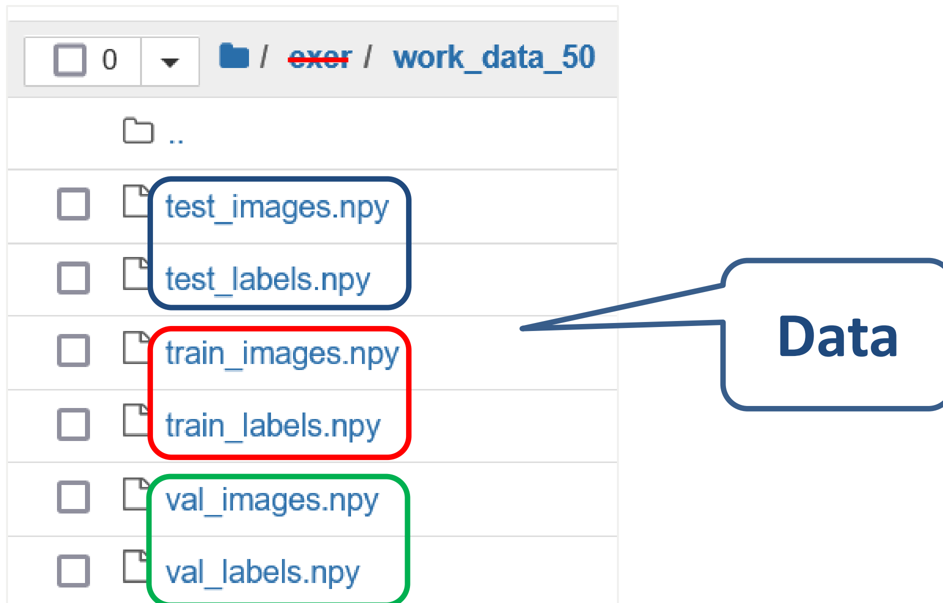
# Outline

- A Brief Introduction to Deep Learning
- Data Processing → Image Classification
- Waste Image Classification
  - Neural Networks
  - Recap of Convolutional Neural Networks (CNN)
    - Tutorial

# Input

| Training set | Validation set | Testing set |
| --- | --- | --- |
| • Model is trained<br>• ~ 80% of the dataset | • Model is assessed<br>• ~ 10% of the dataset | • Model is tested<br>• ~ 10% of the dataset |

**The ground truth**: train, validation & test **label** sets



# Learning a model

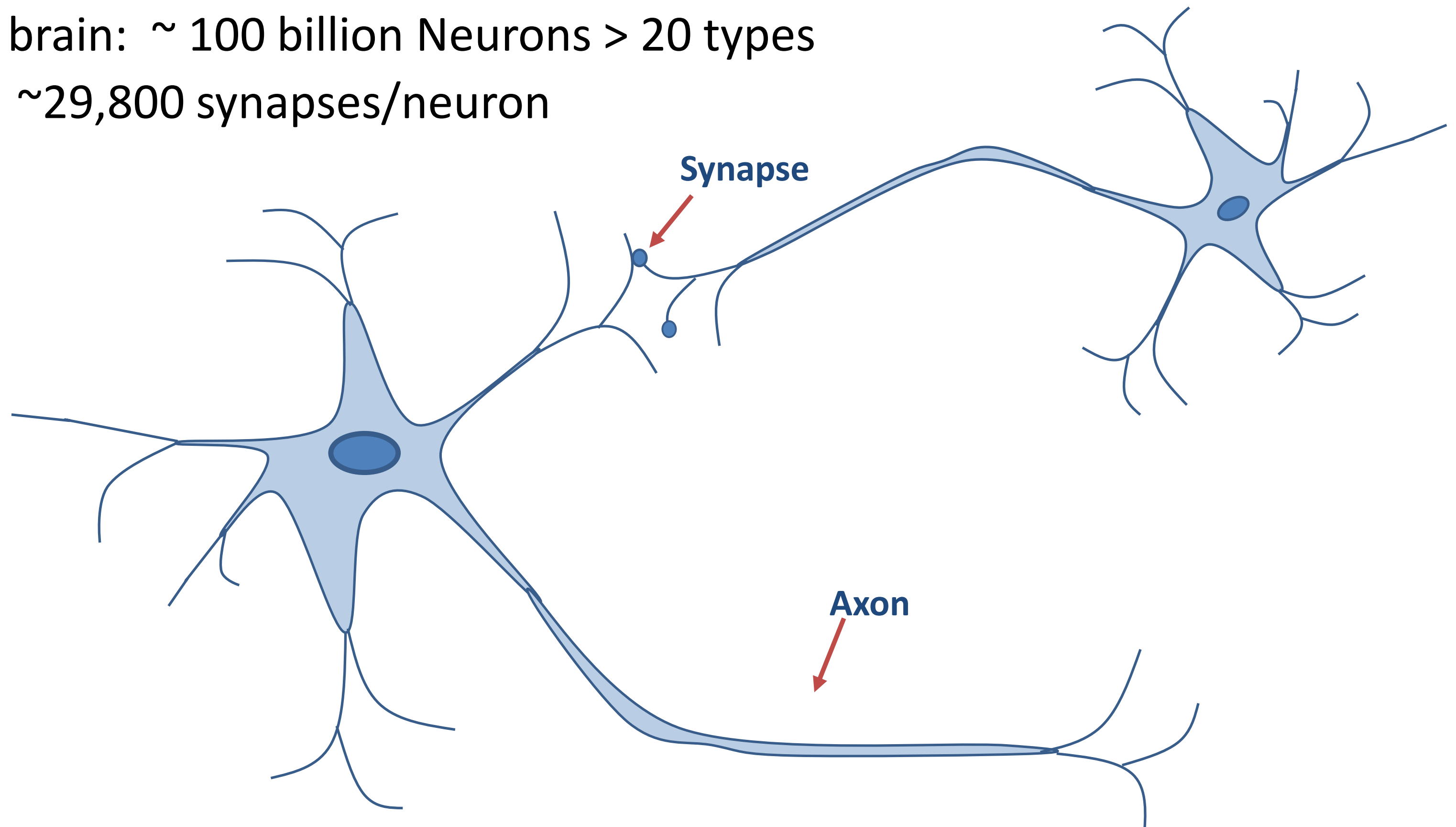
| Configure the Model Architecture | Compile the model | Train the Model |
| --- | --- | --- |
| • Artificial Neural Networks (ANN)<br>• Multilayer Perceptron (MLP)<br>• Convolutional Neural Networks (CNN) | • Loss (to measures how accurate the model is during training)<br>• Optimizer (to minimize Loss with respect of parameters)<br>• Metrics (to evaluate performance) | • Performance at Task improves with an Experience<br>• Train to classify images<br>• Track epochs, let model see every pictures many times; babysit process |

# Neural Networks

# Brain Neurons

**Neurons** are excitable cells which chemically transmit electrical signals through connections called synapses.

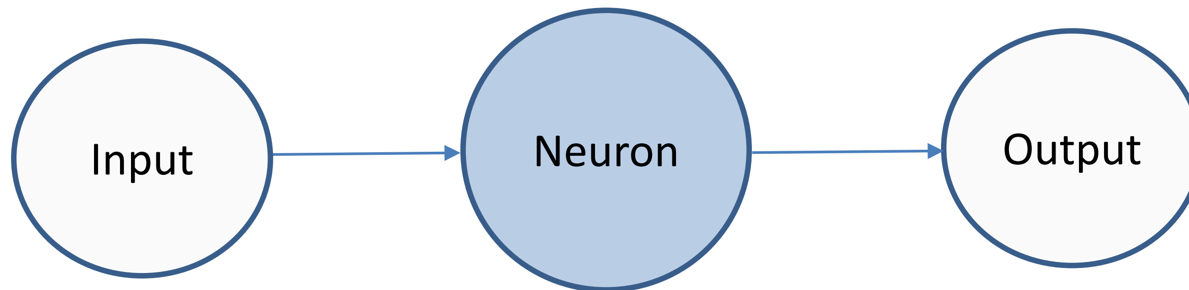
- Human brain: ~ 100 billion Neurons > 20 types
- Cortex: ~29,800 synapses/neuron



# Artificial Neuron

A very coarse model of a **biological neuron**

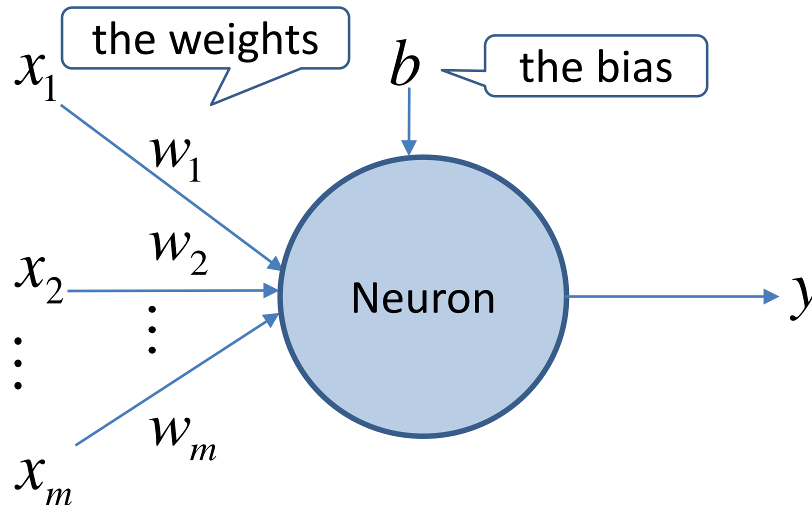
- The smallest unit of a neural network: **a single neuron**




- Such a neuron can handle input with several values, where each values can be weighted differently
- A neuron has the functionality of a **logistic regression**

# Artificial Neuron

- Input of a Neuron $z = x_1 \cdot w_1 + x_2 \cdot w_2 + x_m \cdot w_m + b$

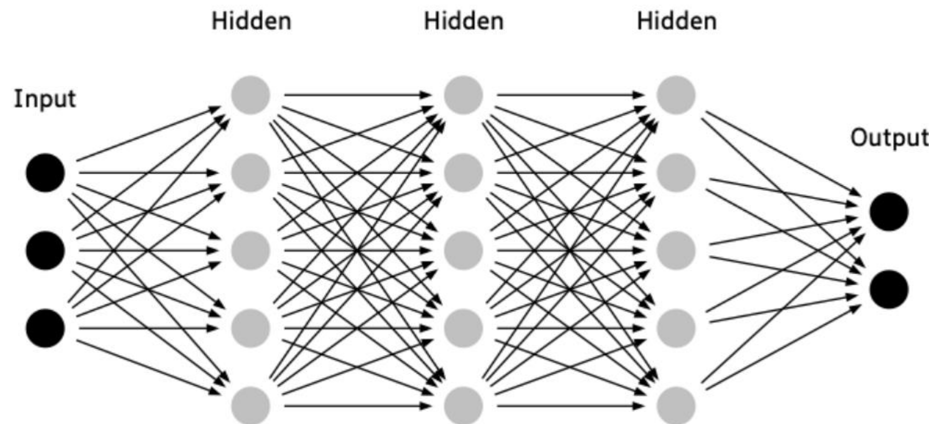



- The affine transformation, a linear transformation of input features via weighted sum, combined with a translation via the added bias.
- Neuron calculates output if we apply **activation function** $f$ on an input function $z$ : $y = f(z)$.
- Combining & connecting of many neurons → **Neural Network**

# Feedforward Neural Networks

**In** Feedforward Neural Networks(FNN)/Multilayer Perceptrons (MLPs):

- **set of neurons** make one **layer;** interlayer nodes - fully connected;
- **transform an input** through a series of **hidden layers**
- every input influences every neuron in the hidden layer, and each of these → every neuron in the output layer
- output layer represents the class scores (i.e., in classification)




**MLP** with 3 inputs, 3 hidden layers of 5 neurons (nodes) each, and 1 output layer.

# Feedforward Neural Networks

Examples of usage:

- Convolutional NNs (object recognition from photos)
- Recurrent NNs (in many natural language applications)

# Feedforward Neural Networks

Weights & Biases

- The goal in FNN is to approximate some function $f^*$.
- For a classifier $y = f^*(x)$ maps an input $x$ to a category $y$.
- A FNN defines a mapping $y = f(x; \theta)$, learns the value of the parameters $\theta$ that result in the best function approximation.
- **Networks is represented by many different functions** (i.e., 3 here) **connected in a chain** to form:

$$f(x) = f^{(3)}\left(f^{(2)}\left(f^{(1)}(x)\right)\right)$$

- With $f^{(1)}$, $f^{(2)}$, $f^{(3)}$ being the first, second & third network layers respectively.
- During neural network training we drive $f(x)$ to match $f^*(x)$.

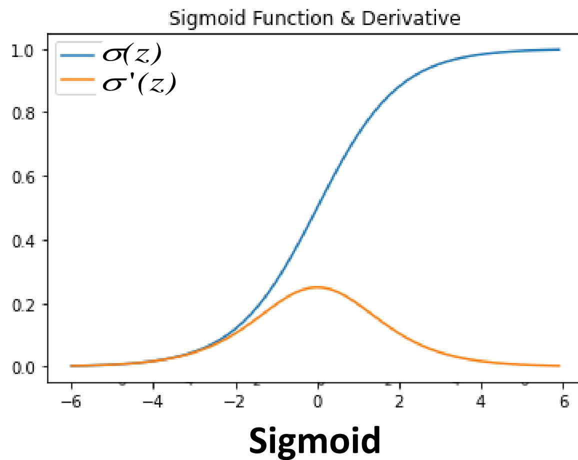
# Feedforward Neural Networks

- Let start with linear models and their limitations. *
- Linear models, i.e., logistic regression and linear regression
  - can be fit efficiently either in closed form or with convex optimization;
  - are limited to linear functions, no understanding about the interaction between any two input variables;
  - If $f^{(1)}$ were linear:
    - the FNN as a whole would remain a linear function;
    - stacking of neurons in network would be useless;
    - its derivative with respect to $x$ will be constant; constant gradient …
- To extend linear models to represent nonlinear functions of $x$ apply the linear model not to $x$ but to a transformed input $\sigma(x)$, where $\sigma$ is a nonlinear transformation.

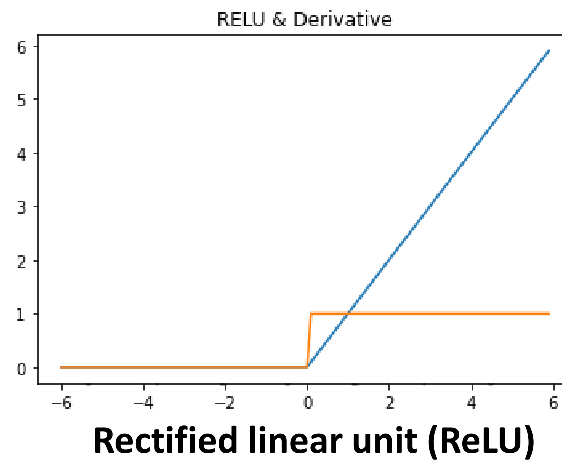
* Goodfellow et. al., Deep Learning, MIT Press, 2016.

# Activation functions

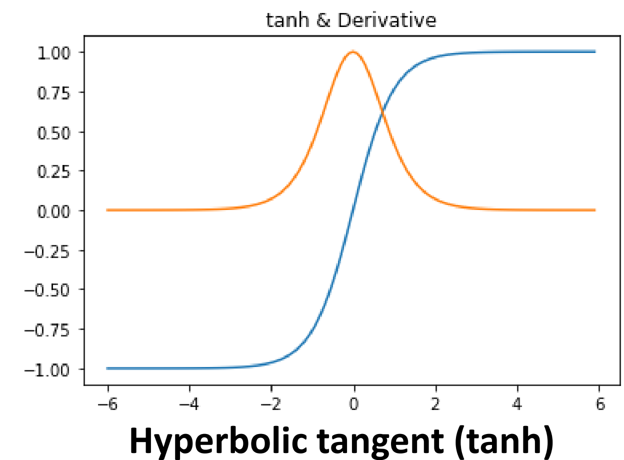
- Introduces non-linearity to Neural Network
- **Non-linear transformation of input** to allow complex tasks




**Sigmoid**



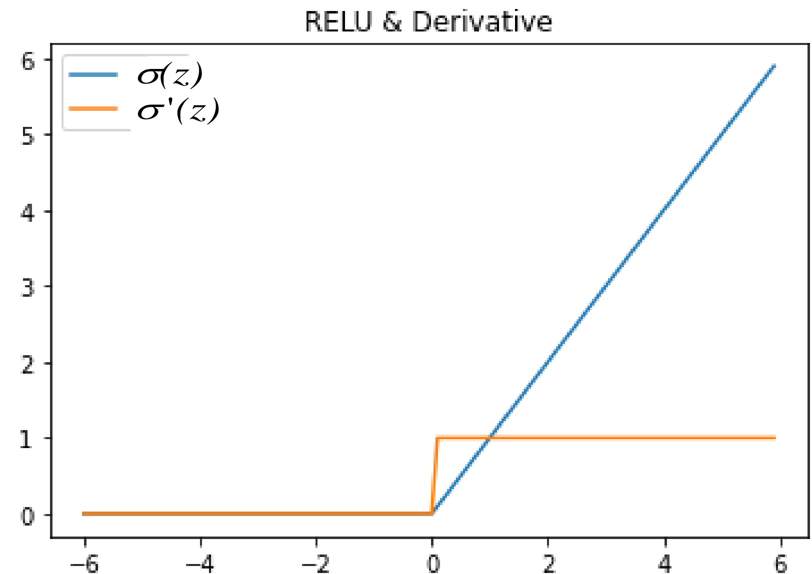

**Rectified linear unit (ReLU)**




**Hyperbolic tangent (tanh)**

# Activation functions - ReLU

- Rectified linear unit activation function
- Fast convergence (sparse activations)
- Constant values
- Negative values do not get activated
- For **CNN ReLU performs faster ***




$$\sigma(z) = \max\{0, z\}$$

Problem:

- Dying ReLU: neurons get stuck at 0
- Can lead to model not learning

Solution: Leaky ReLU w/ small slope for negatives

* 1906.01975.pdf (arxiv.org)

# Softmax

– used as the output of a classifier (last output layer)

– to represent the probability distribution over **n** different classes

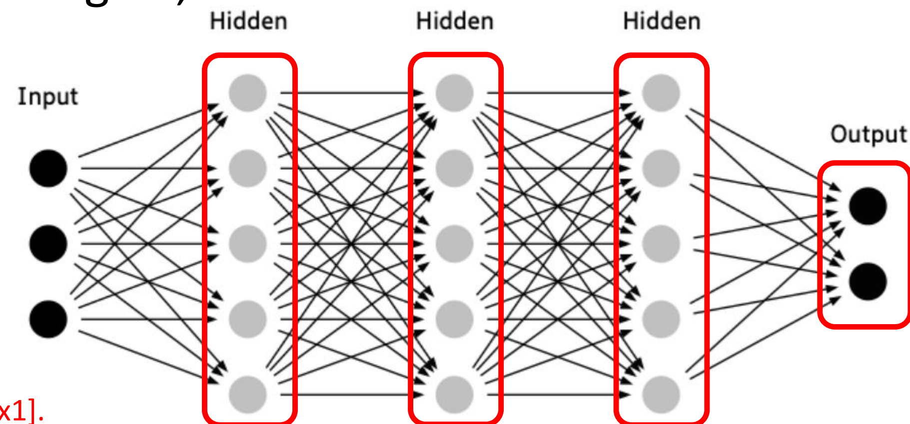
– receives vector as an input and returns a normalized probability distribution of a list of outcomes

```
In [11]: def softmax(x):      # x is vector
             return (np.exp(x))/sum(np.exp(x))

         x = np.array([1, 0.3, 3, 0.5])
         prob = softmax(x)        # converts list of numbers to a list of probabilities
         print(prob)              # output - probabilities
         print(sum(prob))         # sum of the probabilities gives 1

[0.10534997 0.05231524 0.77843681 0.06389798]
1.0
```

# Feedforward Neural Networks

Given **MLP** with 3 inputs, 3 hidden layers of 5 neurons each, and 1 output layer.

- 5 + 5 + 5 + 2 = 17 neurons (not counting the inputs),
- [3 x 5] + [5 x 5] + [5 x 5] + [5 x 2] = 75 weights,
- 5 + 5 + 5 + 2 = 17 biases.
- A total of 92 learnable parameters:
  75 + 17 = 92




For our tutorial example:

$x$ input column vector containing all pixel data of the image [2500x1].

Neurons most commonly do not have an activation function (or you can think of them as having a linear identity activation function).

# MLP

$x_i \in \mathrm{R}^D;$

here $\mathrm{i} = 1\ldots\mathrm{m}$

m images, each with $\mathrm{D} = 50 \times 50 \times 1$ px

$y_i \in 1\ldots\mathrm{n};$ here $\mathrm{n} = 2;$

$x\left[2500 \times 1\right];\ \mathbf{W}\left[2 \times 2500\right]; b\left[2 \times 1\right]$

$f : \mathrm{R}^D \mapsto \mathrm{R}^n$

```
Model: "sequential"
_________________________________________________________________
Layer (type)                Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 2500)              0
_________________________________________________________________
dense (Dense)                (None, 128)               320128
_________________________________________________________________
dropout (Dropout)            (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 2)                 258
=================================================================
Total params: 320,386
Trainable params: 320,386
Non-trainable params: 0
```

# Forward Propagation

- Given the weights W, biases B, and the activation of the input layer $x = x_0$, the **output** $f(x, \mathbf{W}, \mathbf{B})$ of the **neural network** can be computed using forward propagation, matrix multiplication followed by a bias offset and an activation function.

$$z_l = \mathbf{W}_{l-1}\ x_{l-1} + \mathrm{B}_{l-1}$$

$$x_l = \sigma(z_l)$$

$$f(x, \mathbf{W}, \mathrm{B}) = x_{L+1} = \mathbf{W}_L\ x_L + \mathrm{B}_L$$

- For MLP with *L* hidden leyers, each with $h_l$ neurons. $z_l$ *and* $x_l$ denote the input and activation of all neurons in layer $l$.

- $x_{l-1}$ and $\mathrm{B}_{l-1}$ are vectors of size $h_l$ and $\mathbf{W}_{l-1}$ is a matrix of size $h_l \times h_l$.

# Backpropagation

- The algorithm is used to effectively train a NN through a **chain rule** method**.**
- After each **forward pass** through a network, backpropagation performs a **backward pass** while **adjusting** the model's parameters (**weights and biases**) given through the **error**.
- **The gradient descent** algorithm is used to **optimize** (min/max) some **function.**
- By **moving** in the **opposite direction of the slope**, given by **derivative** of this function, we can **improve this function**.
- The **speed** of movement down the gradient is controlled by the **learning rate**, which can be adjusted.
- A higher learning rate might miss the global minimum (optimum), while a low learning rate may get stuck on a local minimum.

Baydin et al., Automatic Differentiation in Machine Learning: a Survey, 2018
Mathieu et al., Fast Training of Convolutional Networks through FFTs, 2014

# Backpropagation

Part1 tutorial

**Backpropagation:**

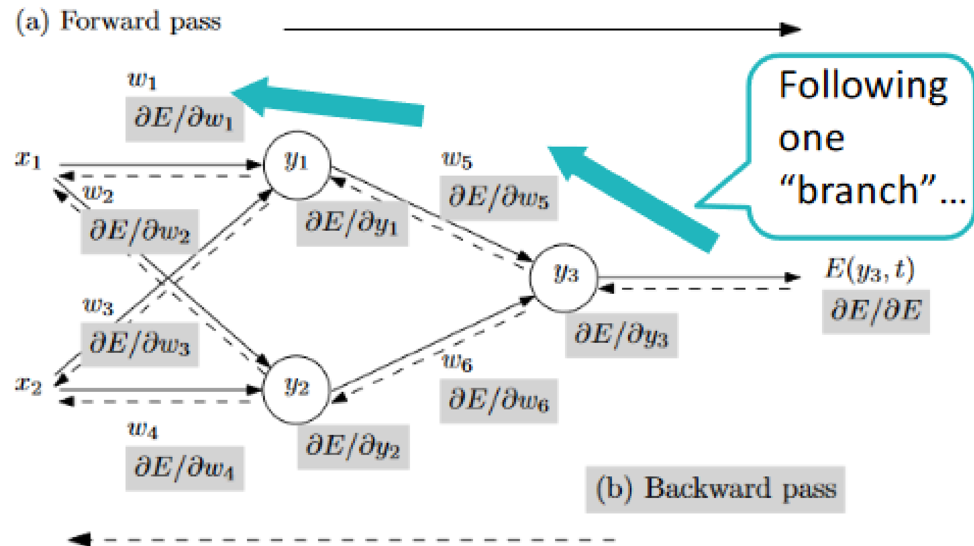
- Training input $x_i$ are fed forward, generating corresponding activations $y_i$.
- $E$ is the error between the final output ($y_3$) and the target ($\widehat{y_3}$, in the paper: $t$), same as the loss function.
- Through the chain rule:

$$\frac{\partial E}{\partial w_5} = \frac{\partial E}{\partial y_3}\frac{\partial y_3}{\partial w_5}$$

$$\frac{\partial E}{\partial w_1} = \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial w_1} =$$

$$= \left(\frac{\partial E}{\partial y_3}\frac{\partial y_3}{\partial y_1}\right)\frac{\partial y_1}{\partial w_1}$$

...




Baydin et al., Automatic Differentiation in Machine Learning: a Survey, 2018
Mathieu et al., Fast Training of Convolutional Networks through FFTs, 2014

# Deep Learning Tutorial

# Tutorial - Simple Neural Network

- **Hands-on:** Simple Neural Networks with Sigmoid
- **Outcome:** Basic understanding of how neuron works with non-linear activation function, feedforward and backpropagation, parameters update.

# Tutorial

Jupyter notebook:

- notebooks/NB-sigmoid-N-Model.ipynb
- ~~notebooks/NB-3_simple_NN.ipynb~~

Tasks to complete in NB-sigmoid-neuron.ipynb:

- Define sigmoid activation function & its derivative; visualize
- Initialize parameters
- Define the input data and the ground truth; perform feedforward calculation; calculate the error, how far are we? adjust the weights accordingly to minimize error
- Get familiar with other activation functions: ReLU etc.
- Experiment with the Softmax function

# Tutorial

H L R | S

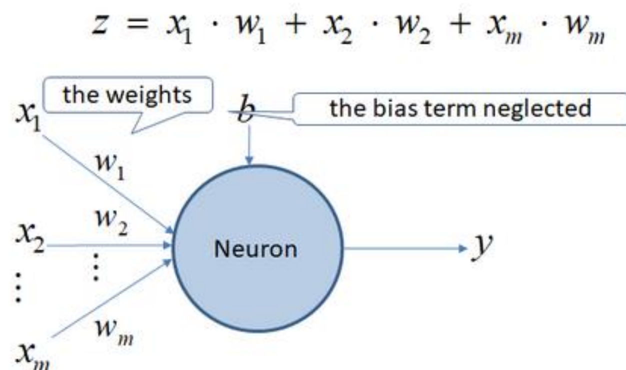
NB-sigmoid-N-Model.ipynb

```
In [ ]: # We will use Sigmoid function for a simple demonstration of Neural Learning.
        # Note code might not work with other data [2]
```

```
In [ ]: #dataset 3 x 4 Matrix
        training_data = np.array([[0,0,1],
                                  [1,1,1],
                                  [1,0,1],
                                  [0,1,1]])
```

```
In [ ]: #output dataset ground truth - i.e., classes {0, 1, 1, 0}
        # y vector
        ground_truth = np.array([[0,1,1,0]]).T
        print(ground_truth)
```

The goal is to find the combination of weights which minimizes the error function, to get training output that is close to the ground truth:




10 min

# Tutorial

NB-sigmoid-N-Model.ipynb

10 min

## Sigmoid

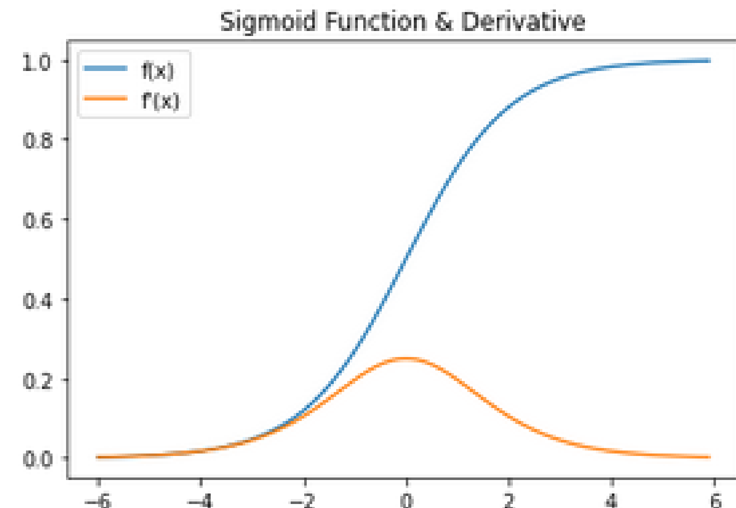
Given a number N, the sigmoid function would map that number between 0 and 1, which means we can use this as probability distribution.

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

```
The derivative of the sigmoid function with respect to x:
```

```
# lets compute

def sig(x):
    return 1/(1+np.exp(-x))

# Sigmoidal derivative
def dsig(x):
    return sig(x) * (1-sig(x))

# Generating data to plot
x = np.arange(-6., 6., 0.1)
y = sig(x)
dy = dsig(x)

# Plotting
plt.plot(x, y, x, dy)
plt.title('Sigmoid Function & Derivative')
plt.legend(['f(x)','f\'(x)'])
plt.show()
```

$$f'(x) = \sigma(x)(1 - \sigma(x))$$



# Tutorial

HLRS

## NB-sigmoid-N-Model.ipynb

For a Feed forward calculations in Neuron we will need:

- input data matrix 3 x 4
- weight matrix 3 x 1 (3 input & 1 output nodes)
- activation function (here sigmoid), to apply on an input function Z (product of the input data with initial weights).

In [ ]:
```
np.random.seed(1)
```

In [ ]:
```
# weight matrix 3 x 1 (3 input & 1 output nodes)
#initialize weights randomly (-1 to 1) with mean 0
weights = 2*np.random.random((3,1))-1

print('Random starting weights', )
print(weights)
```

Training process

1. Feed forward calculation
   - Input layer: training data
   - calculate training_output in Neuron model using sigmoid activation function (bias term is neglected)
2. "Backpropagation" basics
   - Calculate the error (loss), the difference between the ground_truth and actual output
   - Calculate update term
   - Adjust the weights accordingly to minimize error
   - Repeat this 10000 times

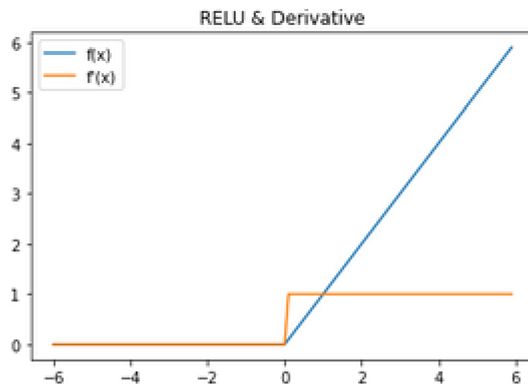
10 min

# Tutorial

NB-sigmoid-N-Model.ipynb

## ReLU

```
# Plotting
plt.plot(x, y, x, dy)
plt.title('RELU & Derivative')
plt.legend(['f(x)','f\'(x)'])
plt.show()
plt.savefig('relu.png', bbox_inches='tight')
```




## Softmax

```
In [11]: def softmax(x):      # x is vector
             return (np.exp(x))/sum(np.exp(x))

         x = np.array([1, 0.3, 3, 0.5])
         prob = softmax(x)        # converts list of numbers to a list of probabilities
         print(prob)              # output - probabilities
         print(sum(prob))         # sum of the probabilities gives 1

[0.10534997 0.05231524 0.77843681 0.06389798]
1.0
```

10 min

# Learning a model

| Configure the Model Architecture | Compile the model | Train the Model |
| --- | --- | --- |
| • Artificial Neural Networks (ANN)<br>• Multilayer Perceptron (MLP)<br>• Convolutional Neural Networks (CNN) | • Loss (to measures how accurate the model is during training)<br>• Optimizer (to minimize Loss with respect of parameters)<br>• Metrics (to evaluate performance) | • Performance at Task improves with an Experience<br>• Train to classify images<br>• Track epochs, let model see every pictures many times; babysit process |

# Compile the model

- **Loss function:**
  measures how accurate the model is during the training.

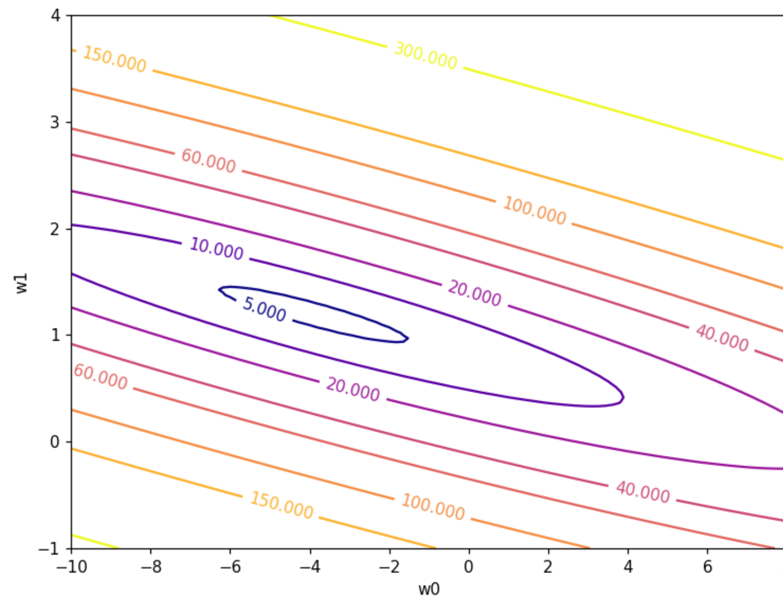
Minimize the loss function

- **Optimizer:**
  the model update based on the data it sees and its loss function.

- **Metrics:**
  used to monitor the training and evaluation steps.

# Compile the model

- **Loss function:** measures how accurate the model is during the training.
- This can be as simple as MSE (mean squared error) or more complex like cross-entropy.
-



# Optimization

- Based on the **gradient descent** algorithm
- Minimization of error through optimization of function
- Moving in the opposite direction of the gradient
- Backpropagation adjusts parameters (backward) given the error
- Training Data may be iterated multiple times
- Complete pass over the data - „epoch“

- Optimizers: GD, SGD, Adam, PMSProp …

# Classification metrics

- We need a Performance **measure P**
  - to assess the performance of the model
  - to monitor the training and evaluation steps

- Default metric for classification is **accuracy,** the fraction of the images that are correctly classified

- This **metric is not useful** when there is a **data imbalance**
  - the distribution of examples in the training dataset across the supercategories is not equal
  - e.g. proportion in supercategory < 50%

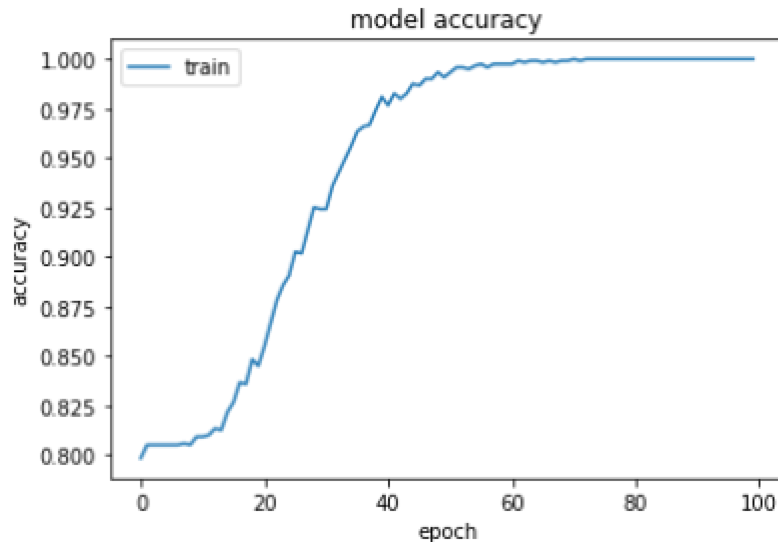
# Classification metrics

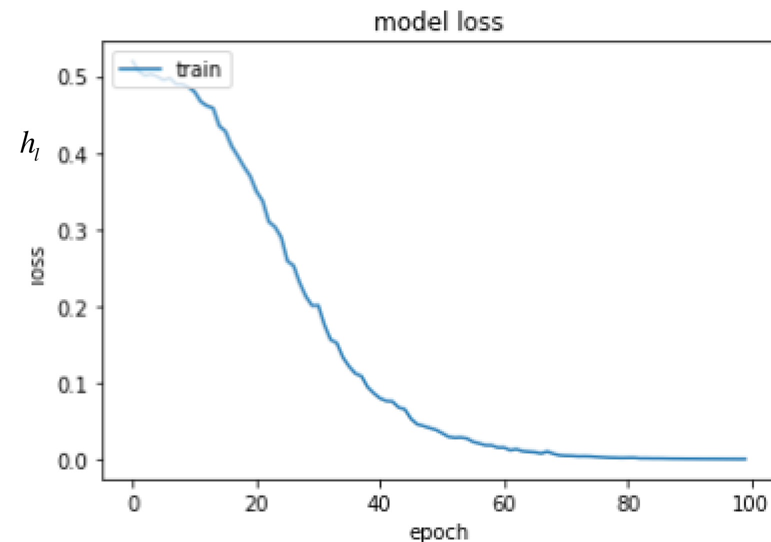
**Problem:**

- Only 37% of the validation set and 21% test set with bottles .
- Predicting every image **as not containing** a bottle would give ~63%  and ~79 % accuracy, which is not representative of how well the model is doing on predicting bottles.

# Classification metrics

The **low performance** on the minority class (supercategory) **is** not **captured** in the accuracy metric.




Almost perfect accuracy according to the model training history.

# Classification metrics

More complete picture according to the **confusion matrix**

- how many classes were correctly classified vs misclassified?
- The simplest confusion matrix for a 2-class classification problem, with negative (0 - no bottle) and positive (1  bottle) classes
- Precision - percentage of relevant results
- While recall is characterized as the percentage relevant results that are correctly classified

| | | **Predictions** + | **Predictions** - |
|---|---|---|---|
| **Actual** class | + | **TP** True Positive | **FN** False Negative Type II error |
| **Actual** class | - | **FP** False Positive Type I error | **TN** True Negative |

what if FN represents one with COVID-19 ?

# Classification metrics

**Main metrics:** Breakdown the accuracy formula even further

| Metric | Formula | Interpretation |
| --- | --- | --- |
| Accuracy | $\frac{TP + TN}{TP + TN + FP + FN}$ | Overall performance of model |
| Precision | $\frac{TP}{TP + FP}$ | How accurate the positive predictions are |
| Recall<br>Sensitivity | $\frac{TP}{TP + FN}$ | Coverage of actual positive sample |
| Specificity | $\frac{TN}{TN + FP}$ | Coverage of actual negative sample |
| F1 score | $\frac{2\,TP}{2\,TP + FP + FN}$ | Hybrid metric useful for unbalanced classes |

# Learning a model

| Configure the Model Architecture | Compile the model | Train the Model |
| --- | --- | --- |
| • Artificial Neural Networks (ANN)<br>• Multilayer Perceptron (MLP)<br>• Convolutional Neural Networks (CNN) | • Loss (to measures how accurate the model is during training)<br>• Optimizer (to minimize Loss with respect of parameters)<br>• Metrics (to evaluate performance) | • Performance at Task improves with an Experience<br>• Train to classify images<br>• Track epochs, let model see every pictures many times; babysit process |

# Training of a NN

- From the training example Inputs get the neurons output (feedforward)
- Calculate the error (loss), the difference between the output we got after calculation and actual output (the ground truth - input labels)
- Adjust the weights accordingly to minimize error (Backpropagation)
- Backpropagation computes the derivative of the loss with respect of weights (different optimizer: GD, SGD, Adam, PMSProp …)
- Repeat this many times (i.e., Epoche = 20 or more)
- A small loss leads to a good prediction

# Evaluation

**Evaluate**

- Model performance is evaluated on validation set
- Trained Model gives predictions on unseen data
- Chosen Metrics suffice

# Learning a model

| Configure the Model Architecture | Compile the model | Train the Model |
|---|---|---|
| • Artificial Neural Networks (ANN)<br>• Multilayer Perceptron (MLP)<br>• Convolutional Neural Networks (CNN) | • Loss (to measures how accurate the model is during training)<br>• Optimizer (to minimize Loss with respect of parameters)<br>• Metrics (to evaluate performance) | • Performance at Task improves with an Experience<br>• Train to classify images<br>• Track epochs, let model see every pictures many times; babysit process |

Instead of MLP

# Image Classification task

- **Dataset** → annotated, RGB images of different size
- **Image Classification task** → predict a single label (or a distribution over labels to indicate confidence) for a given image.

- Resize: 1000 pixels wide, 1000 pixels tall.
- RGB → gray scale images
- Results → 1000 x 1000 x 1, or a total of 1 Million numbers
- Pixel range: from 0 (black) to 255 (white)
- The task: to turn numbers into a single label, such as *"bottle"*

# Image Classification challenges

- **Viewpoint variation** of a single instance of an object (bottle)
- **Scale variation** - size in the real world vs in the image
- **Deformation** - i.e., deformed plastic bottle
- **Occlusion** - only a small portion of an object visible
- **Illumination conditions** - direct effects on the pixel level
- **Background clutter** - making hard to identify object
- **Intra-class variation** - many different types of these objects

➢ CNNs systematize this idea of **spatial invariance**, exploiting it to learn useful representations with fewer parameters.

# Why Convolutions?

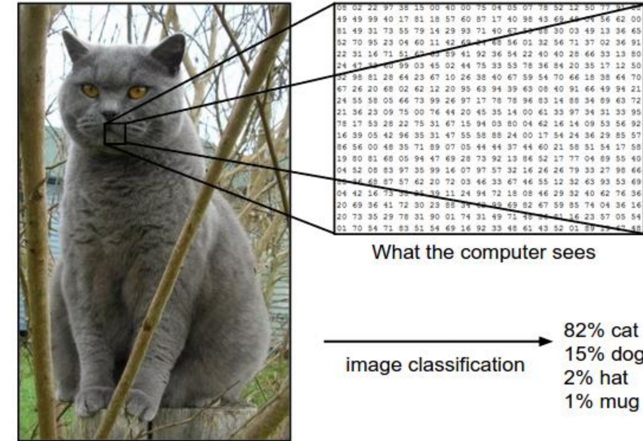
**Problems:**

- **Impractical** to use ANNs for real-world image classification
  - a 2D image - 1 Million numbers per image
  - If the first hidden layer has 1000 nodes
  - the matrix of input weighs → 1000 x 1000 x 1000
  - increasing the number of layers **increase numbers rapidly**
- **Vectorising** an image **ignores** the complex **2D spatial structure**

- How to build a system that overcomes both these disadvantages?
  - **Convolutional neural networks (CNNs)** are one creative way

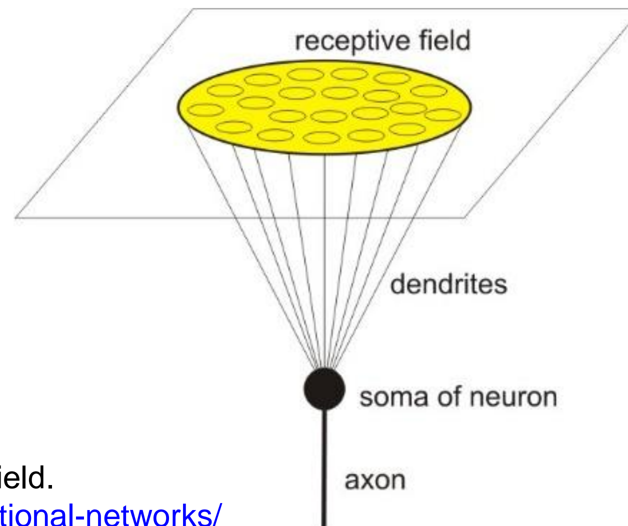
# CNN

- Human perception is very accurate
- Computers see images as 2D arrays of pixels
- Algorithms need to be trained on lots of images




Source: http://cs231n.github.io/classification/

- CNN mimics human eye




A a single sensory neuron's receptive field.
Source: http://cs231n.github.io/convolutional-networks/

# CNN - advantages and disadvantages

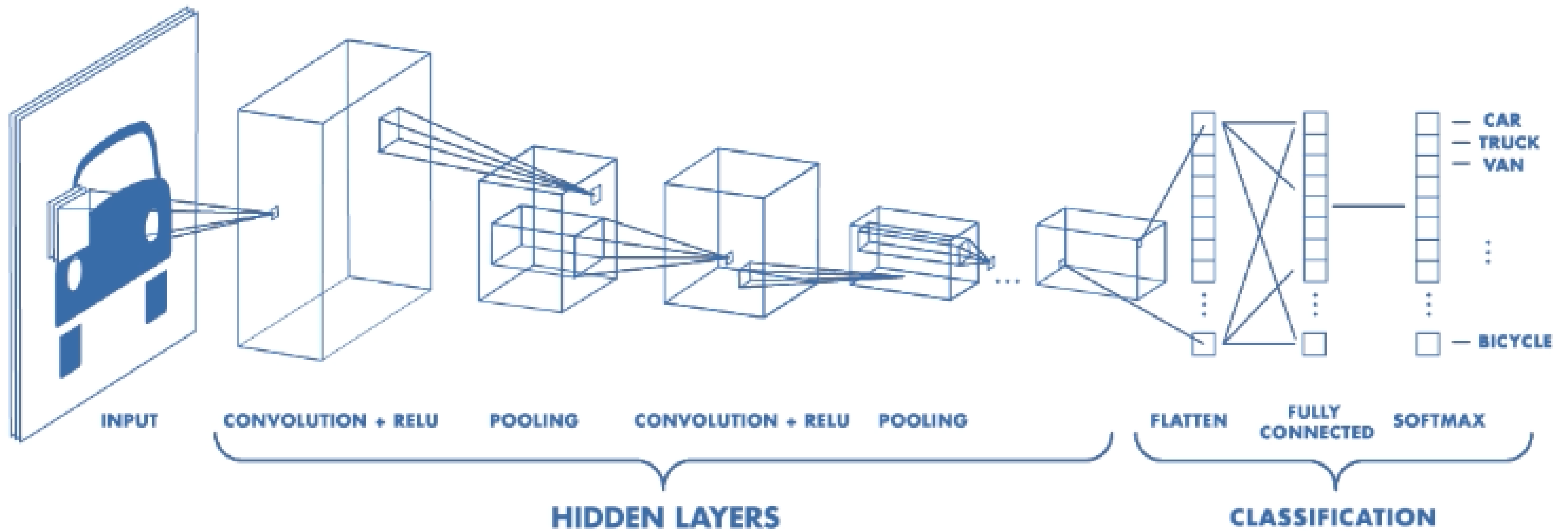
**Pros**

- CNN - is a class of deep feedforward ANN, that **contain convolutional layers**
- Improves the performance by using **spatial information** of pixels of an image
- Convolutional layers **require fewer parameters** than fully-connected layers
- **Larger the data, greater the accuracy** - the first fully connected layer with thousands of weights
- **Translation invariance** in images **automatically obtained**
  - all patches of an image are treated in the same manner
  - the same weights across the whole space
- **Locality** - from a small neighborhood of pixels to the corresponding hidden representations

**Cons**

- Downside of deep CNN: a bad learning performance could be improved with hyperparameter tuning

# Components of CNN

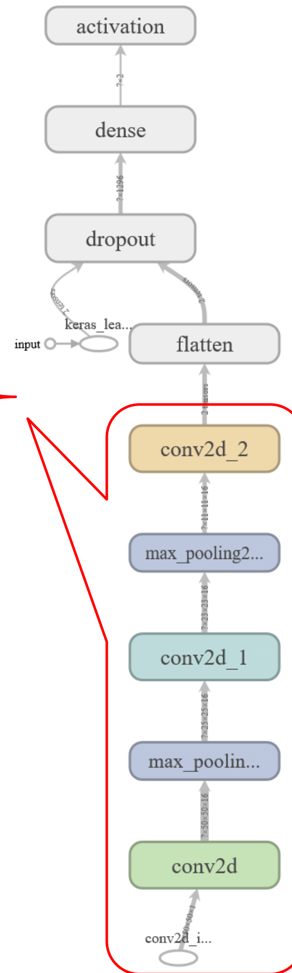



Source: https://www.mathworks.com/videos/introduction-to-deep-learning-what-are-convolutional-neural-networks--1489512765771.html

# CNN MODEL

| Layer (type) | Output Shape | Param # |
|---|---|---|
| conv2d (Conv2D) | (None, 50, 50, 16) | 160 |
| max_pooling2d (MaxPooling2D) | (None, 25, 25, 16) | |
| conv2d_1 (Conv2D) | (None, 23, 23, 16) | 2320 |
| max_pooling2d_1 (MaxPooling2 | (None, 11, 11, 16) | |
| conv2d_2 (Conv2D) | (None, 9, 9, 16) | 2320 |
| flatten (Flatten) | (None, 1296) | 0 |
| dense (Dense) | (None, 2) | 2594 |
| activation (Activation) | (None, 2) | 0 |

Total params: 7,394
Trainable params: 7,394
Non-trainable params: 0




→ Results of the labels; a class {1, 0}

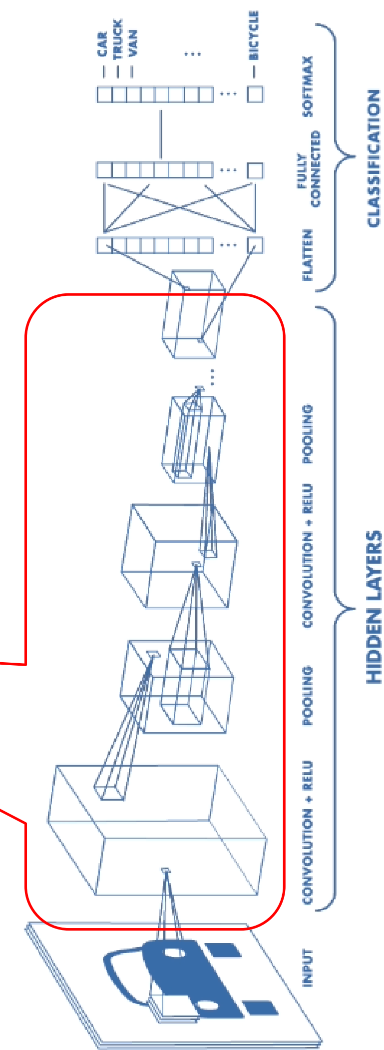
→ Computation

→ Regularization

→ a single vector

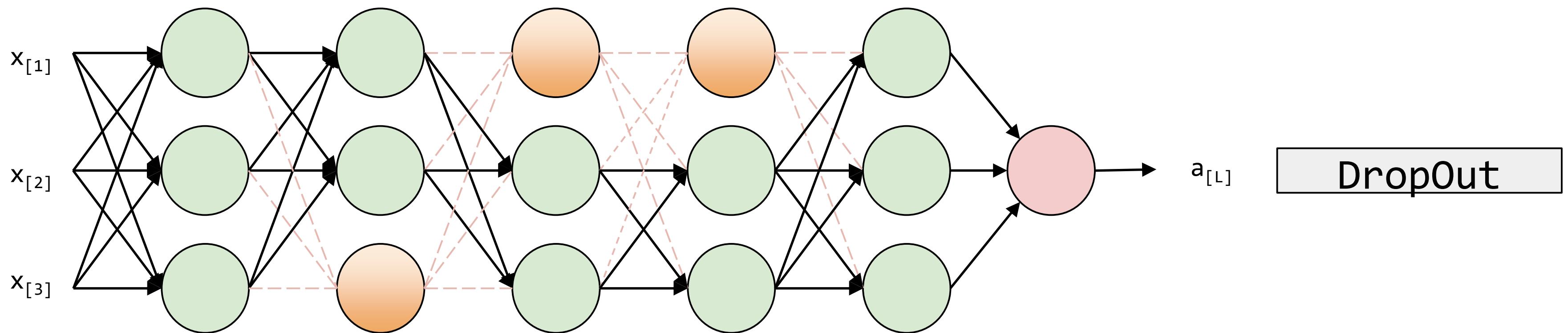
→ Feature mapping in Convolution layer

→ down-sampling of the data



# Dropout

- Regularization technique
- Prevents overfitting making memorization difficult
- Method: randomly throw activations away (e.g. p=0.5),
- Early dropout coupled with RELU – preventive



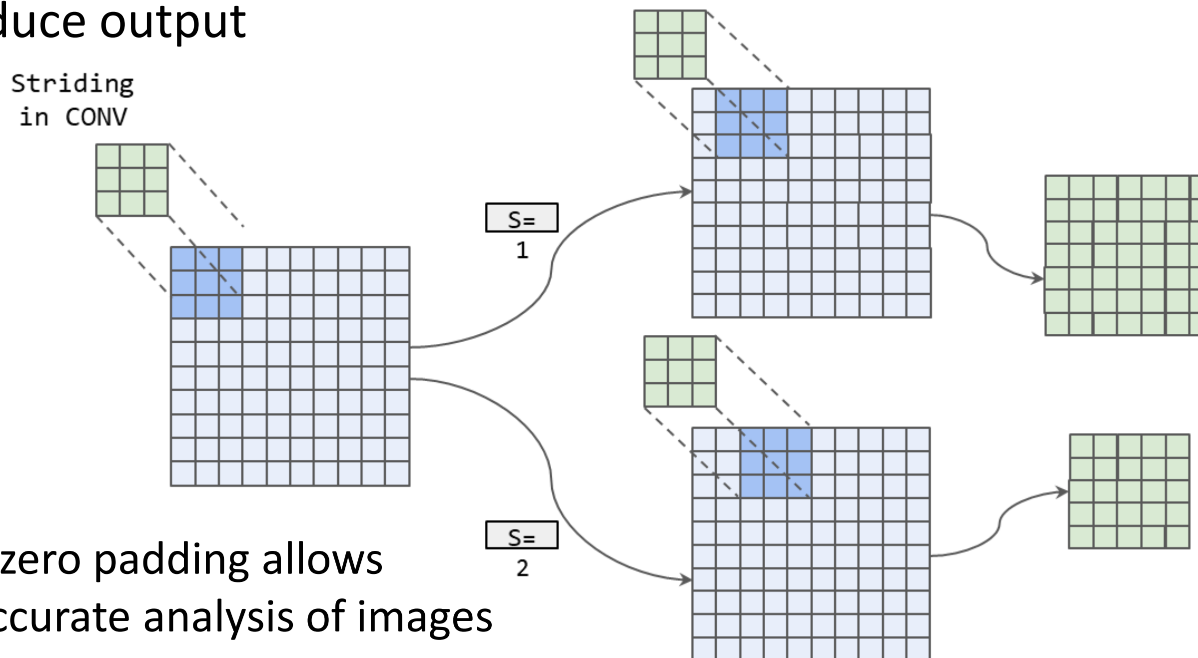

Source: https://github.com/dair-ai/ml-visuals

# CNN

- Convolution
- Pooling operation
- Activation functions
- Dropout
- Backpropagation

# Convolution Operation

- Combination of 2 functions to produce a third function
- Input, kernel (e.g. 3x3), feature map (output)
- Stride kernel across the input and compute matrix multiplication to produce output

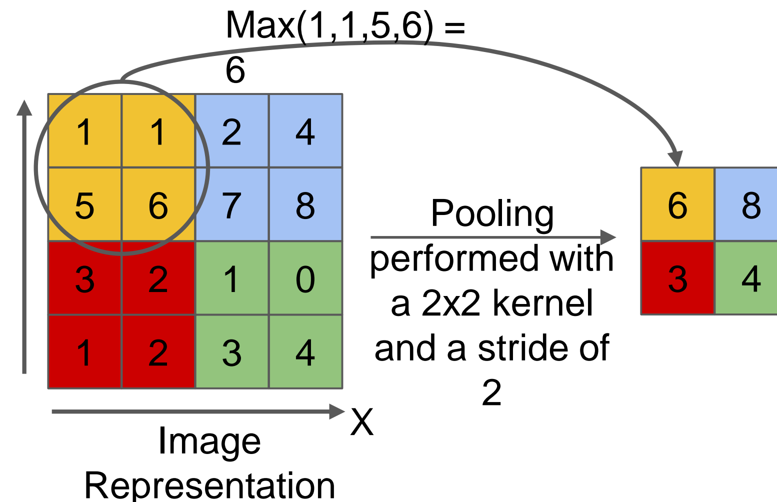



- Adding zero padding allows more accurate analysis of images

Source: https://github.com/dair-ai/ml-visuals

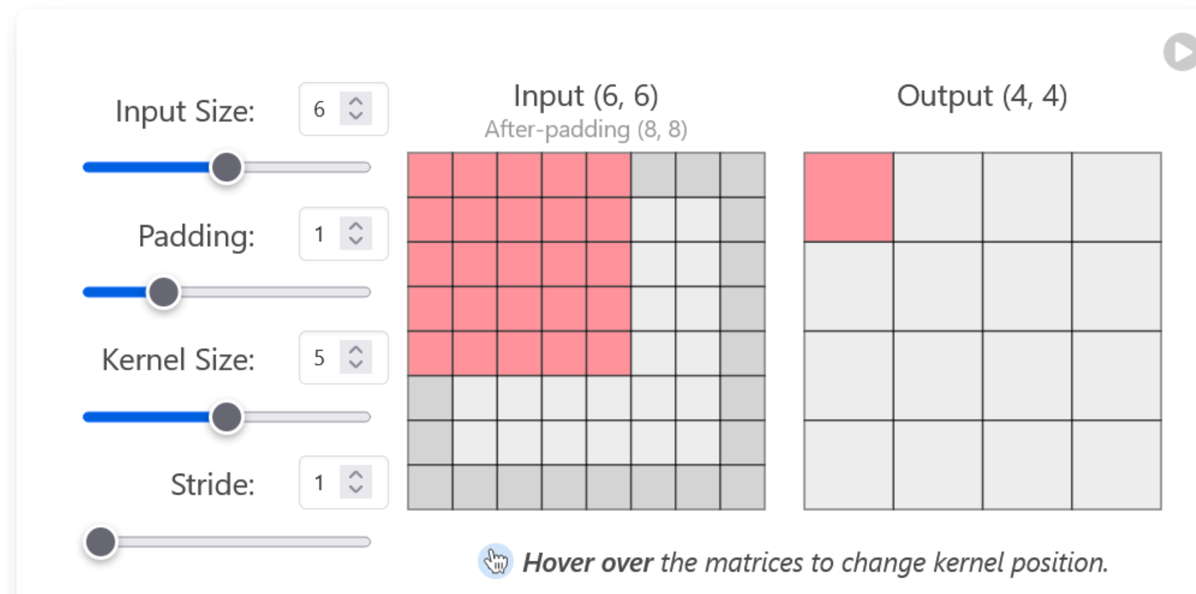
# Pooling Operation

- Summarizes the output of a region
- Helps reduce the effect of invariants (small changes to the input)
- Max vs mean-pooling



# CNN Explainer

An interactive *visualization* system designed to help non-experts learn about Convolutional Neural Networks (CNNs).




https://poloclub.github.io/cnn-explainer/

# Deep Learning

# Tutorial

# Tutorial - Image Classification with CNN

- **Hands-on:** Image Classification with CNN
- **Outcome:** Basic understanding of CNN architecture and building blocks. Ability to explain difference between CNN and FNN; advantages of CNN; modify the model architecture, compile, train CNN and evaluate. Experiment with model hyperparemeters and proper metrics for the unbalanced dataset.

# Tutorial

## Jupyter notebook

> > Open a **notebooks/**NB-train_CNN_50.ipynb

Tasks to complete:

- Load saved numpy arrays (3 image sets, 3 label sets)
- Summarize training, validation, and test data.
- Normalize, scale, experiment
- **Configure model for CNN**
- Experiment with the hyperparamenter (learning rate etc.)
- During experiment use number of epochs = 30
- Visualize training history
- Observe how changes affecting results in confusion matrix

# Outlook

- **No Dimensionality reduction:**
  - Work with RGB channel images in CNN
- **Data Augmentation:**
  - Check the influence of data augmentation on the model performance
  - From unbalanced data → balanced data

# Optional

To work in terminal for optional data augmenting part you can use

> **solution/preprocess.py**

**> solution/job_preprocessing.pbs**

# Thank you!

Email: kakhiani@hlrs.de