Technische Hochschule Rosenheim

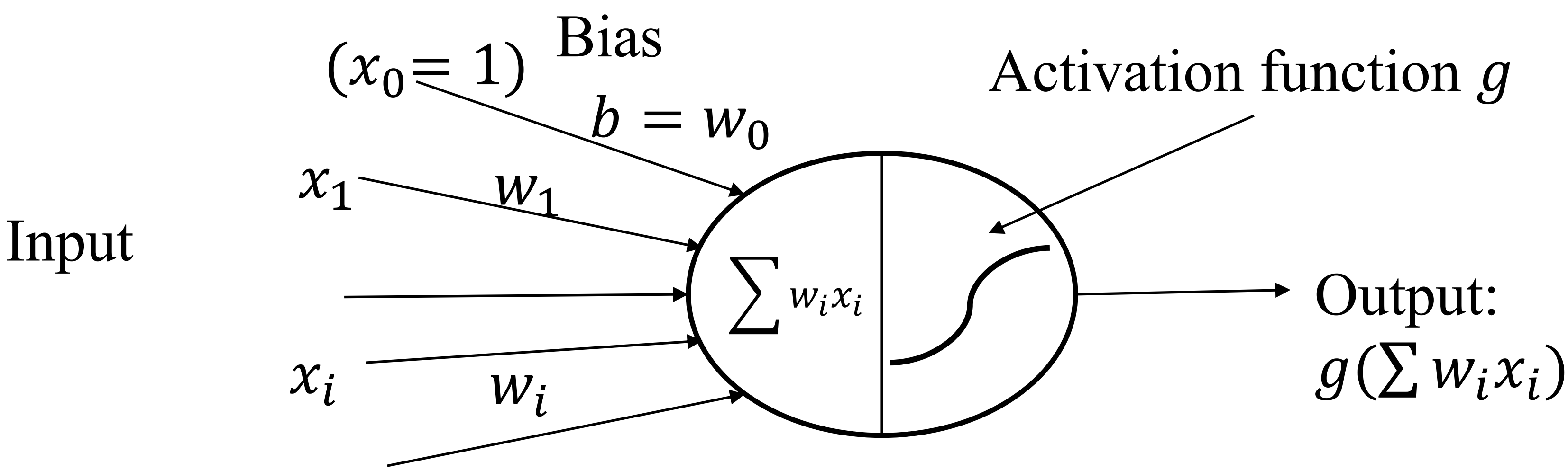
Supervised Learning

Neural Networks 2

Prof. Dr. Johannes Jurgovsky

Recap: Logistic Regression

- A binary classifier
- Output: linear function of input +
  (non-linear) activation function (logistic function)
  $\hat{y} = g(\sum w_i x_i)$
  $\hat{y}$ is interpreted as $p(y = 1 \mid x)$



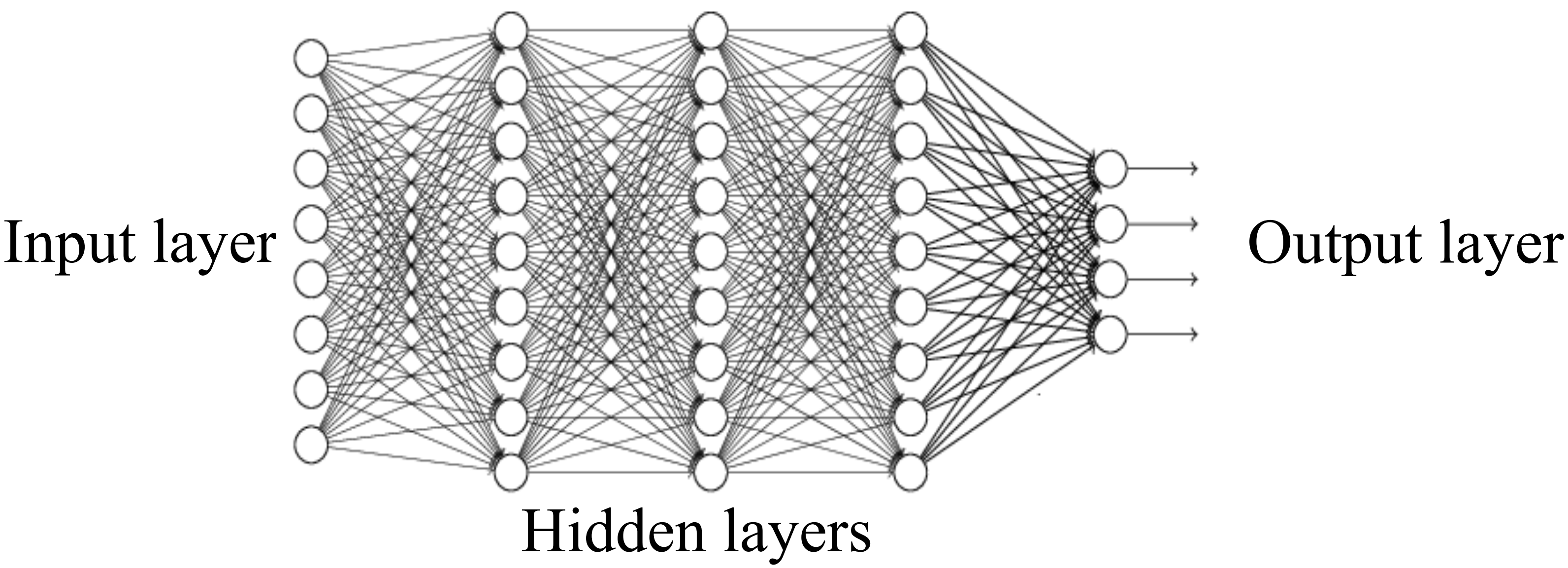
Multi-Layer Perceptron (aka Neural Network)

- Try it here: http://playground.tensorflow.org

Multi-Layer Perceptron (MLP)

- Neurons are arranged in layers
- Layer n is (fully) connected with layer $n+1$
  - no connections within layer
  - no connections to any other layers
  - no feedback
- Information flow from one layer to the next: feed-forward
- network has no internal state

- number of input/output units is problem dependent
- number of hidden units determined by developer



Slides based on material from M. Breunig, J. Schmidt

Technische Hochschule Rosenheim

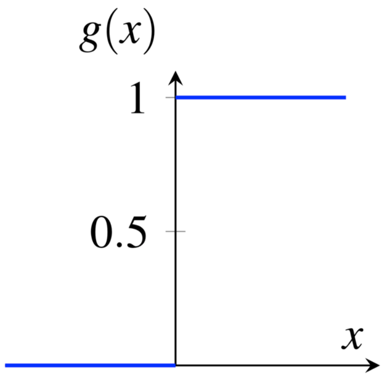
MLP – Properties

A MLP can

- compute any Boolean function (AND, OR, XOR, …)
- approximate any non-linear function
  - a 2-layer MLP with a finite number of hidden neurons suffices; however, a lot of neurons may be required in that hidden layer
- define arbitrary class boundaries
- be trained in a supervised fashion using a sample set (Error Backpropagation)
- be adapted to binary classification, multi-class classification and regression

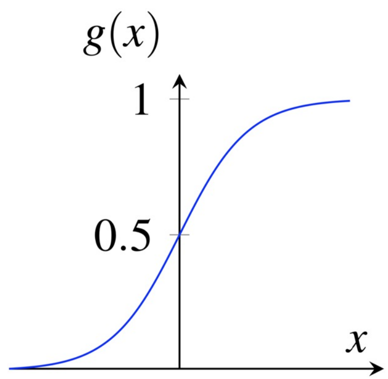
Activation Functions (classic)

Step function/Threshold



not differentiable

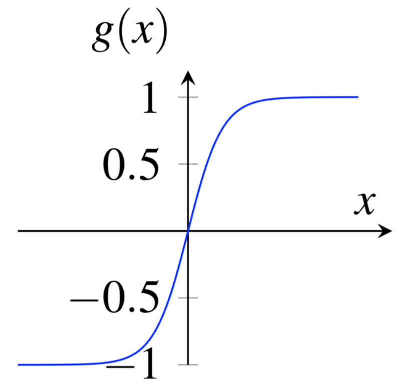
Sigmoid/logistic function



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

$$g'(x) = g(x)(1-g(x))$$

Sigmoid/tanh



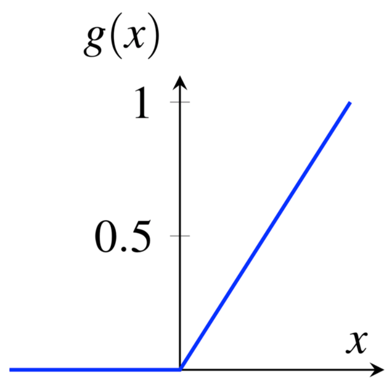
$$g(x) = \tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

$$g'(x) = 1 - g^2(x)$$

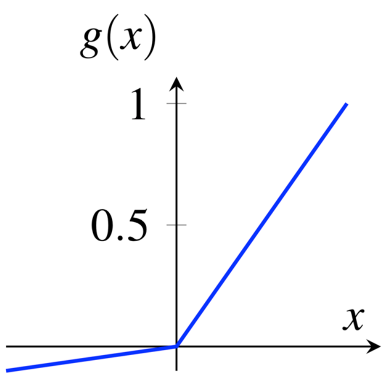
Changing the bias $b = w_0$ moves threshold

Activation function – (Leaky) ReLU / Softplus

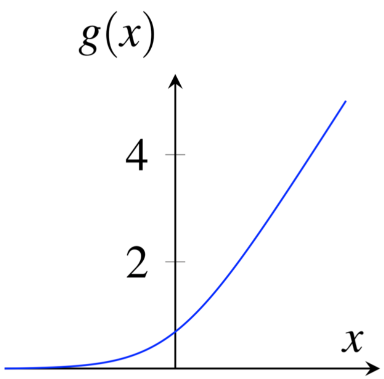
ReLU



Leaky ReLU



Softplus



$$g(x) = \max(0, x)$$

$$g(x) = \begin{cases} 0.01x & x \leq 0 \\ x & x > 0 \end{cases}$$

$$g(x) = \ln(1 + e^{x})$$

$$g'(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}$$

$$g'(x) = \begin{cases} 0.01 & x \leq 0 \\ 1 & x > 0 \end{cases}$$

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

- Ramp function
- ReLU = Rectified Linear Unit
- by now the most common activation functions in deep neural networks (for inner neurons)
- Variant Softplus: smooth approximation of ReLU

Activation Function – Softmax

- A normalized exponential function $g(x)_j = \frac{e^{x_j}}{\sum_i^C e^{x_i}}$
- Smooth approximation of MAX-Function
  - lifts high values and suppresses low values
- Used in output neurons
- Acts like a probability mass function over C classes
- Example

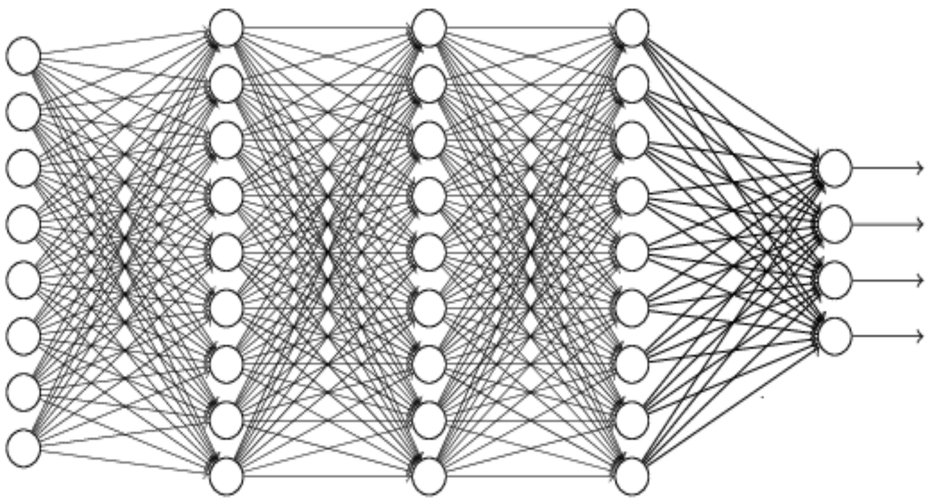
in	1	2	3	4	1	2	3
out	0,024	0,064	0,175	0,475	0,024	0,064	0,175

Technische Hochschule Rosenheim

Training

Training = Determine weights

1. Feed training examples through the network (forward pass)
2. Error-Backpropagation (backward pass)



Deep learning

- = representation learning = learn feature extraction
- use many layers of neurons

- Example: AlexNet (image recognition)
  - Convolutional Neural Network (CNN) → Computer Vision (winter term)
  - 8 layers (not all of these fully connected)
  - 650,000 neurons
  - 60 million parameters
  - trained on 1.2 million images
- Example: ResNet – 50 to 150 layers deep

Technische Hochschule **Rosenheim**

MLP – Training (Error-Backpropagation)

- Feed training examples through network
- Compute error for each example n and each output neuron j:
  $e_j(n) = y_j(n) - a_j(n)$ (y: desired output, $a = \hat{y}$ actual output)
- Minimize total error of all output neurons: $\varepsilon(n) = \frac{1}{2}\sum_j e_j^2(n)$
  - here: Mean Squared Error (MSE)
  - using non-linear optimization
    (e.g. gradient descent; requires partial derivatives of error function)
  - result: *local* optimum
  - the required update of a weight between neuron j of layer l and neuron i of layer $l-1$ is:
    $$\Delta w_{ji}(n) = -\alpha \frac{\partial \varepsilon(n)}{\partial z_j(n)} a_i(n)$$
    where $z_j(n)$ is the weighted sum of neuron inputs and α is the learning rate
- Partial derivative for neurons in output layer:
  $$\frac{\partial \varepsilon(n)}{\partial z_j(n)} = -e_j(n) g'\left(z_j(n)\right)$$
- Partial derivative for hidden neurons in layer l:
  $$\frac{\partial \varepsilon(n)}{\partial z_j(n)} = g'\left(z_j(n)\right) \sum_k \frac{\partial \varepsilon(n)}{\partial z_k(n)} w_{kj}(n)$$
  where k iterates over all neurons of layer $l+1$

Technische Hochschule Rosenheim

Normalization / Weight Initialization

- Input data:
  Normalize, such that all values are of similar magnitude

- Weight Initialization:
  - Initialization of all weights with zeros or a single constant is bad :
    Neurons in hidden layers collapse to a single neuron
    (because of symmetry)

  - Random initialization: common heuristics, e.g.
    - uniformly from the interval $[-0{,}01;\ +0{,}01]$
    - uniformly from the interval $[-\frac{1}{\sqrt{n}};\ +\frac{1}{\sqrt{n}}]$, where $n =$ number of neurons of previous layer
    - for deep neural networks: uniformly from the interval $[-\frac{\sqrt{6}}{\sqrt{m+n}};\ +\frac{\sqrt{6}}{\sqrt{m+n}}]$, where $m =$ #neurons of previous layer, $n =$ #neurons of subsequent layer

- Initialization of bias: Zero

Notes

- Advantages:
  - parallel computation within layer possible
  - can be formulated as matrix multiplications
    → well suited for GPUs
- Disadvantages:
  - Number of hidden layers and number of neurons per hidden layer is a design decision
    - theoretical minimum of 2/3 layers
      (but more may be better, e.g., faster convergence, less neurons)
    - there are presently no well-founded results on how to choose these hyper-parameters → needs to be done experimentally
  - high computation times for training as well as classification
  - non-linear optimization guarantees only local minimum
- Try it here: http://playground.tensorflow.org

Learning Rate

- Learning Rate too high
  - Optimization gets stuck on plateau
  - or even diverges
- Learning Rate too small
  - slow convergence

- Start with high values, e.g. $\alpha = 0.1$ or even higher
- If diverging: Decrease learning rate

- Typical learning rates are (depending on optimization method) 0.01 or 0.001

Optimization Method – Mini Batch SGD

- SGD = Stochastic Gradient Descent (SGD)
- Most used
- Faster than standard gradient descent

- For each iteration k:
  - select a small set of training examples randomly (mini-batch)
  - compute gradients
  - update weights
  - decrease the learning rate linearly:
    $\alpha_k = (1 - \beta)\alpha_0 + \beta\alpha_\tau$ where $\beta = \frac{k}{\tau}$
    learning rate stays constant after iteration τ

Optimization Method – Momentum

- accelerated trainings for
  - high curvature
  - small, but consistent gradients
  - noisy gradients
- Idea: Store gradients from previous updates
  - as floating mean
  - influence of older values decreases exponentially
- Accumulate gradients

$$m_t = \gamma m_{t-1} + \alpha \delta d$$

- Update parameters

$$w_t = w_{t-1} - m_t$$

- m: moment, α: learning rate, δ: partial derivative in corresponding layer, d: output of neurons of previous layer, w: weights
  γ: dampening factor (e.g. 0.9)
- Nesterov-Moments
  - gradient is computed after weights have been updated in current iteration
  - basically, a correction term to account for changes in gradient

Technische Hochschule Rosenheim

Optimization Method – Adaptive Learning Rate

- selecting the ”correct” learning rate is
  - critical – high impact on result
  - difficult
- slightly less so when using moments
  - but at the cost of an additional parameter

- adaptive learning rate:
  - determine learning rate separately for each weight update (batch) based on a global learning rate parameter

Technische Hochschule Rosenheim

Optimization Method – Adaptive Learning Rate

- AdaGrad (2011)
  - changes learning rate proportional to square root of past squared gradients
  - works well for some deep networks, but not for others
- RMSProp (2012)
  - modified AdaGrad, working better with neural networks
  - uses gradient accumulation with exponential dampening
  - there is a variant in combination with Nesterov-Moments
  - widely used for neural networks
- Adam (2014)
  - combination of RMSProp with moments
  - uses also second order moments
- in general, the training result is highly dependent on parameter settings

Technische Hochschule Rosenheim

Regularization – Dropout

- Regularization: avoid extreme values for weights
- one possibility: Dropout
  - in each iteration: set output of some randomly selected neurons to zero (as if they had been removed)
  - typical value: 0.5 (= 50% probability for removal)

- Advantage:
  - forces network to learn redundant representations of classes
  - prohibits that different neurons concentrate on the same features

- behaves like training a large set of network models that share parameters (Bagging)

Objective Function / Loss Function

- Mean Squared Error (MSE):
  see Error-Backpropagation

- Cross Entropy (or Log Loss)
  - for comparing two probability densities
  - Computation (for a single sample): $-\sum_{i=0}^{m-1} y_i \ln p_i$
    m: #classes,
    y_i: desired output (typically 0 or 1),
    p_i: actual output

- Binary Cross Entropy
  - Cross Entropy for two classes
  - Computation simplifies to: $-(y \ln p + (1-y) \ln (1-p))$

Activation/Loss Function – Classification

- Two classes
  - use a single neuron in output layer
  - activation output layer: sigmoid (logistic)
  - activation hidden layers: ReLU
  - loss function: binary cross entropy

- Multiple disjoint classes
  - use one neuron per class (1-out-of-n coding, one-hot) in output layer
  - activation output layer: Softmax
  - activation hidden layers: ReLU
  - loss function: cross entropy

- Multiple non-disjoint classes
  - use one neuron per class in output layer
  - activation output layer: sigmoid (logistic)
  - activation hidden layers: ReLU
  - loss function: binary cross entropy (summed over all output neurons)

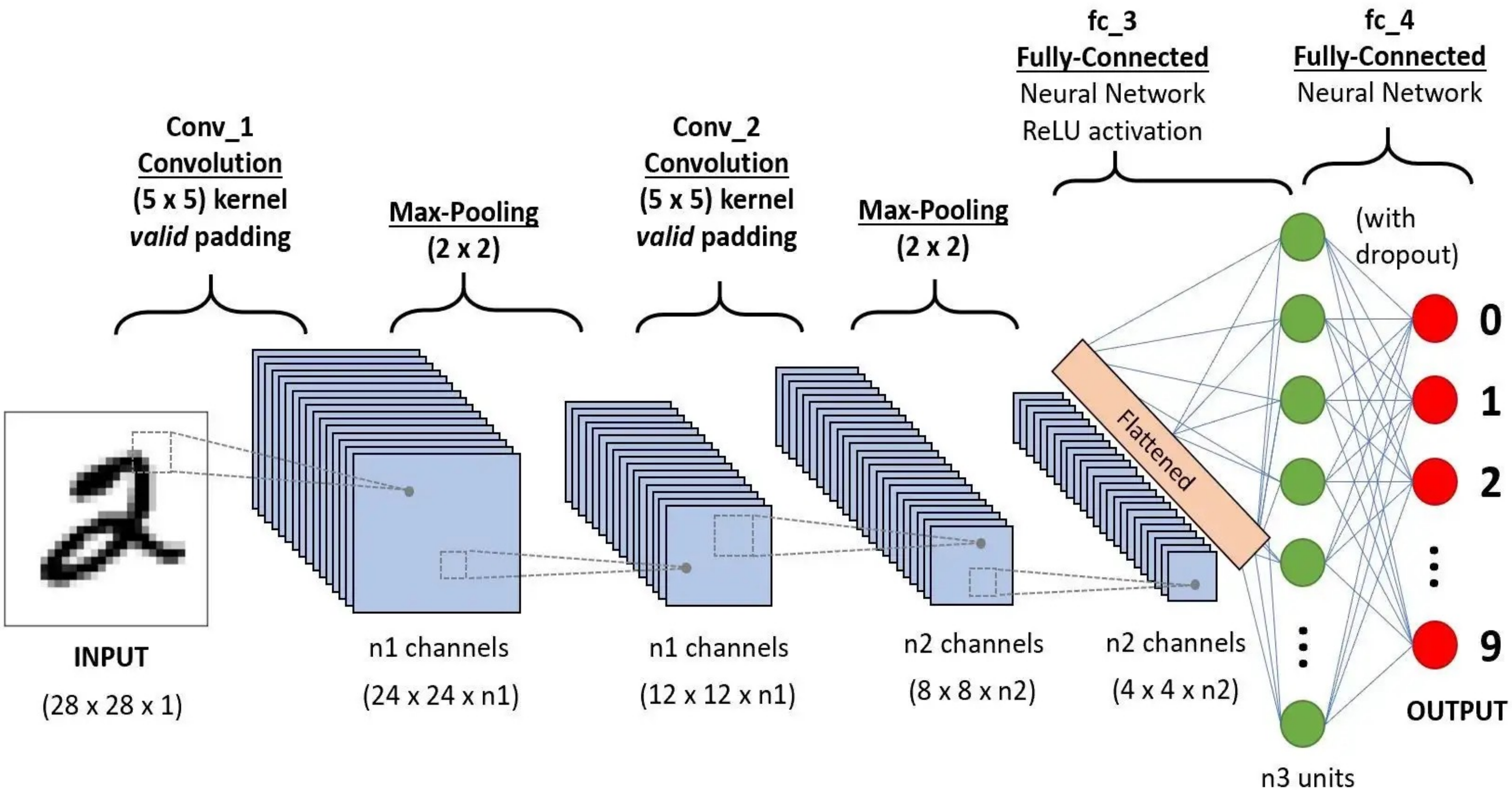
Activation/Loss Function – Regression

- instead of discrete classes: compute function value

- use a single neuron in output layer
- activation output layer: linear
  (sum of weights passes through unchanged)
- activation hidden layers: ReLU
- loss function: mean squared error

Convolutional Neural Network (CNN)

- deep feed-forward networks
- not fully connected
- integrates convolution operation (filter) in network

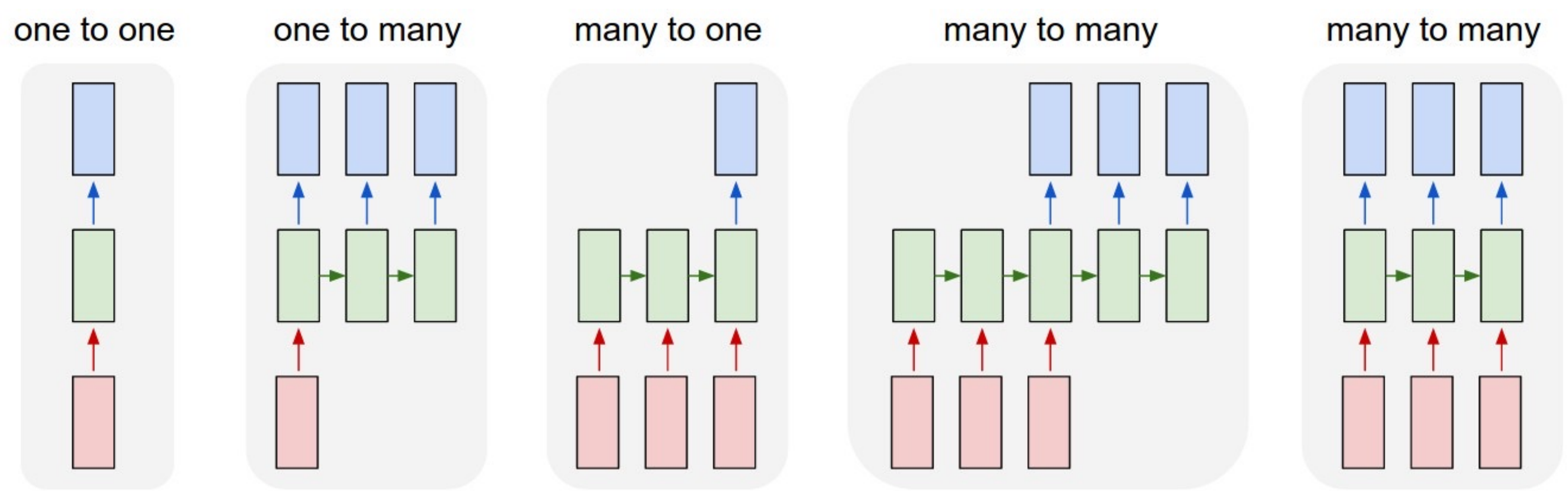


Source

Technische Hochschule Rosenheim

Recurrent Neural Network (RNN)

- for sequential data
- maintains internal state
- widely used: Long Short-Term Memory (LSTM) networks



Source

Technische Hochschule Rosenheim

Neural Network APIs

- scikit-learn contains a neural network API
  - should not be used
    (not intended for large scale applications, no GPU support)

- PyTorch
  - Facebook neural network API (open source)
  - https://pytorch.org/

- Tensorflow
  - Google Python API for neural networks (open source)
  - https://www.tensorflow.org/