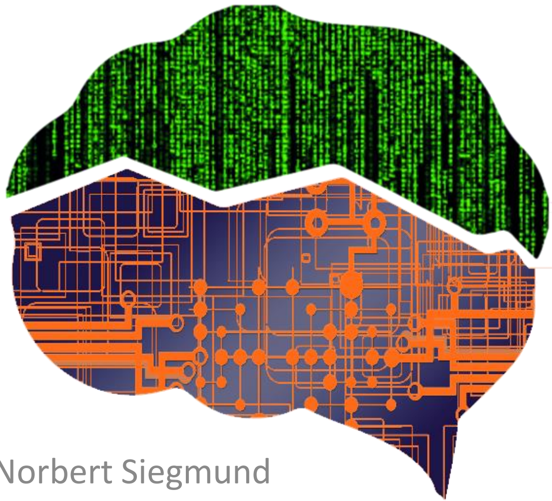
# Machine Learning for Software Engineering

## Neural Networks Overview



Prof. Dr.-Ing. Norbert Siegmund
Intelligent Software Systems

Bauhaus-Universität Weimar

# Recap I

- Curse of dimensionality
  - Distances between points in higher dimensions become equal
  - Nearly all points are on the edge of the search space
  - To get the same information in higher dimensions, we need to obtain an exponentially larger set of observations
- PCA
  - Unsupervised dimensionality-reduction technique
  - Idea: Find a small number of higher order (latent) variables that explain the same information as the many original variables and use only those for later processes
  - Approach: Find correlating variables and compute a vector that minimizes the variance when transferring the variables

# Recap II

- Feature (subset) selection
  - Supervised dimensionality-reduction technique
  - Find a subset of features/variables that explain/predict a dependent variable with minimal/specified information loss
  - Forward and backward selection
  - Brute force and iterative search
  - Different evaluation measures

# Neural Networks

# (Short) Intro to Neural Networks

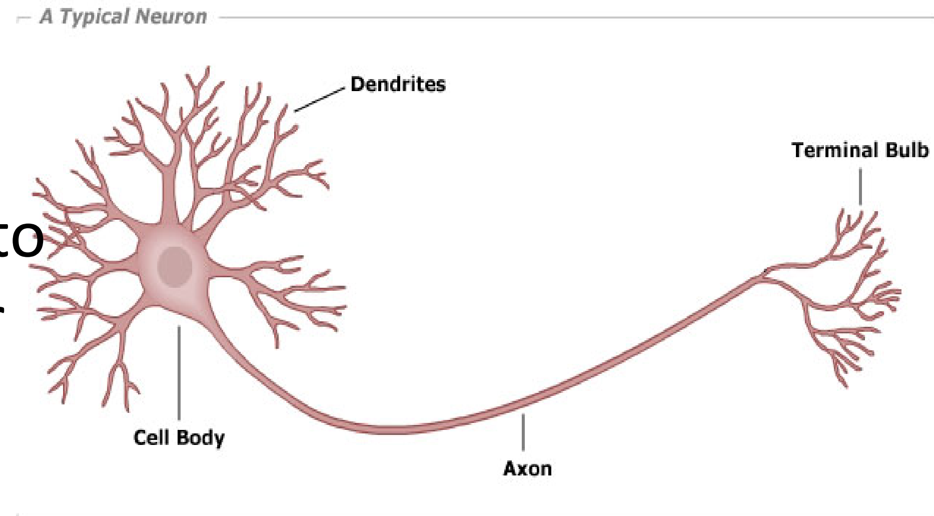
- A specific machine-learning technique
  - Unsupervised for pattern detection
  - Supervised for classification, prediction, etc.
  - Reinforcement learning (learns what actions to take to accomplish a task)

- Huge number of tasks have been accomplished
  - Speech recognition
  - Object detection and labeling
  - AI solutions for playing games (chess, GO, etc.)
  - Robot control
  - Even arts, music, etc.

# Example: ImageNet (2012 competition)

# Neuron

- Dendrites collect input from other neurons
- Axons cause electric charges to transmit information to other neurons
- Axon hillock generates outgoing spikes whenever a critical charge has been reached
- Synapses (terminal bulb) translate an electric charge into a chemical reaction

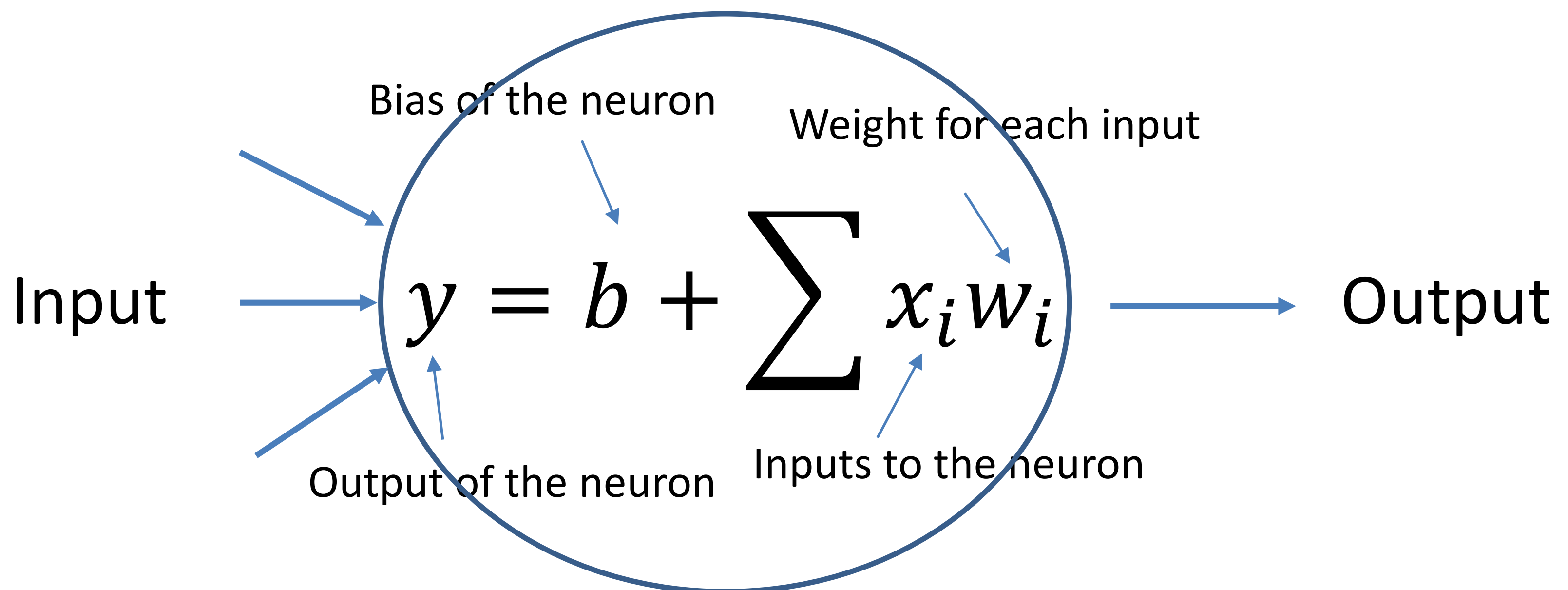


# Interesting Properties of Neurons

- Synapsis strength can change
  - Varying number of transmitting vesicles (dt. geladene Bläschen)
  - Varying number of receptor molecules
- Comparing synapses with RAM
  - Low power and very small
  - Adaptability to local signals
  - So, synapses contain weights
  - Each neuron can contain 10,000 weights!

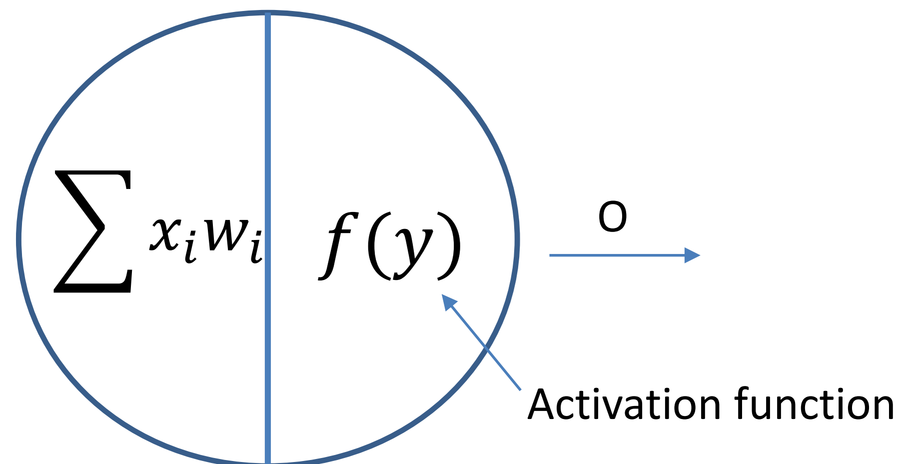
# Linear Neuron (No Activation Function)

- A neuron combines all incoming signals, usually by building the sum
- Easiest neuron just passes these signals weighted by the links and pass them forward



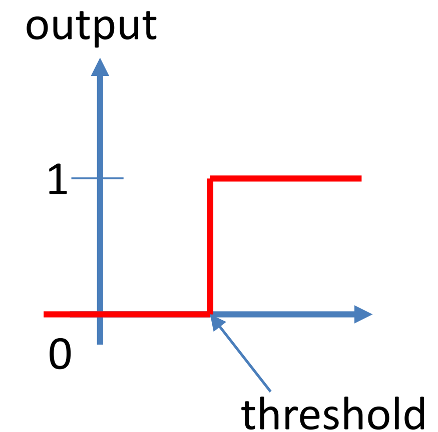
# Activation Functions = Types of Neurons

- A neuron does not just forward the combination of the inputs
- Usually, a threshold must be reached to let a neuron fire an output value
- How the threshold is determined and how large the output will be, is determined by an activation function



# Binary Threshold Neuron

- From McCulloch-Pitts (1943)
- Send an output after a defined threshold over the sum of input signals has been reached
- Can be used to combine truth functions




- Get rid of the bias

$$y = b + \sum x_i w_i$$

$$o = \begin{cases} 1 \text{ if } y \geq 0 \\ 0 \text{ otherwise} \end{cases}$$

$b$=-$\theta$

$$y = \sum x_i w_i$$

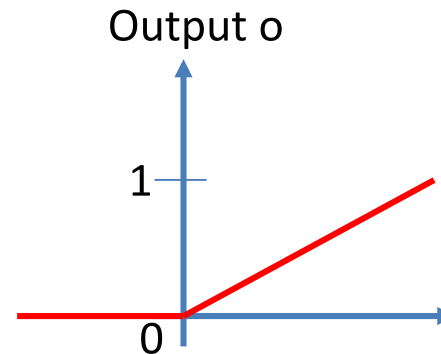
$$o = \begin{cases} 1 \text{ if } y \geq \theta \\ 0 \text{ otherwise} \end{cases}$$

# Rectified Linear Neuron

- Again, first compute the weighted linear sum over the input signals
- Then, compute a non-linear output

$$y = b + \sum x_i w_i$$

$$o = \begin{cases} y \text{ if } y \geq 0 \\ 0 \text{ otherwise} \end{cases}$$

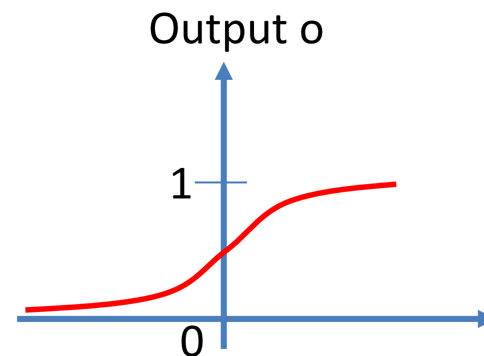


# Sigmoid Neuron

- Problems of sharp, step functions
  - Derivate cannot be computed
  - Even if it can, it is often 0
  - With 0 slope, where to go next?
- So, the smooth sigmoid function is preferred

$$y = b + \sum x_i w_i$$

$$o = \frac{1}{1 + e^{-y}}$$

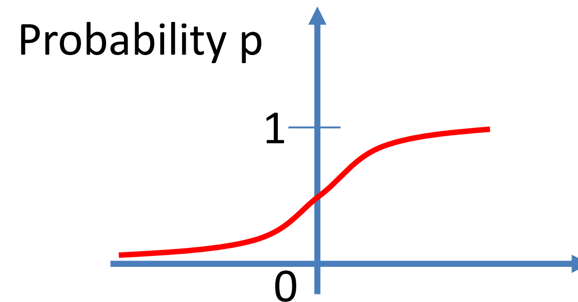


# Stochastic Binary Neuron

- Similar to sigmoid neuron, but output is treated as a probability of producing a spike

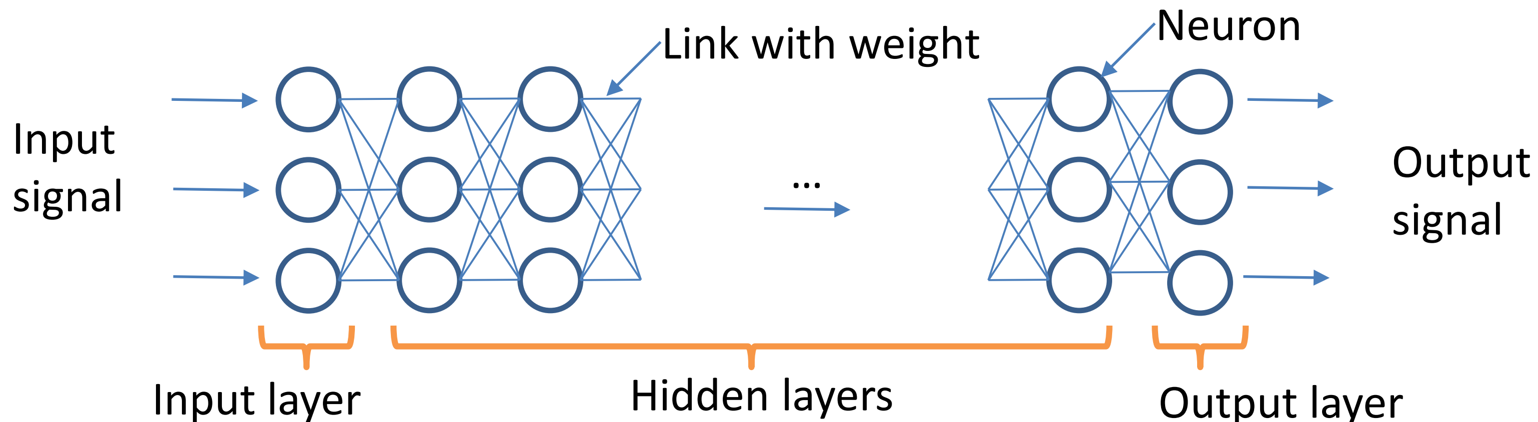
$$y = b + \sum x_i w_i$$

$$p(s = 1) = \frac{1}{1 + e^{-y}}$$



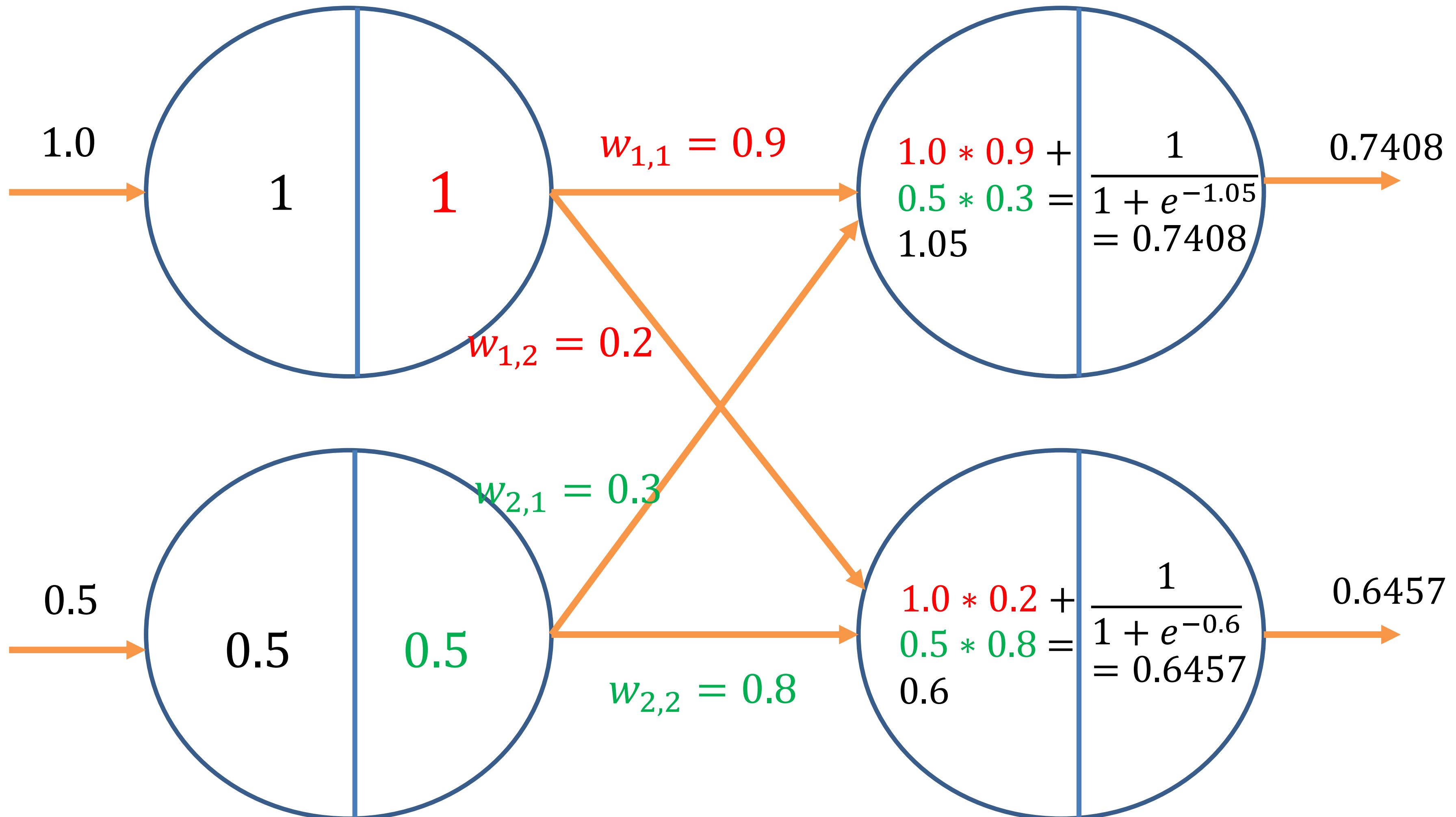
# Forward Propagation

- Process of querying the network to obtain an output
  - Based on the chosen activation function in the next layer and the learned weights for the connections, the input is transferred to produce an output for each neuron in the second layer
  - The output of the second layer is the input of the third layer and this process continues till the output layer

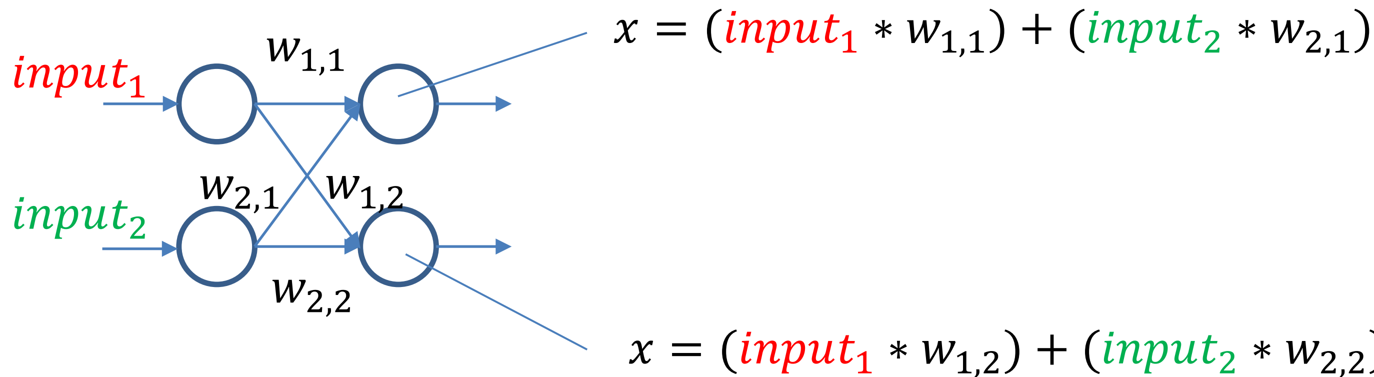


# Forward Prop Example

Input layer does not computation! Sigmoid neuron



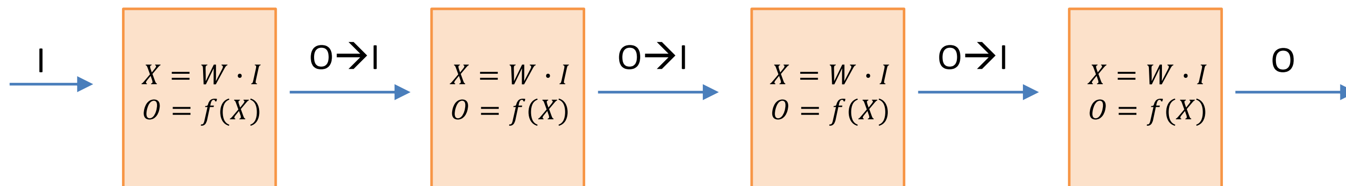
# Forward Prop Matrix Formalization



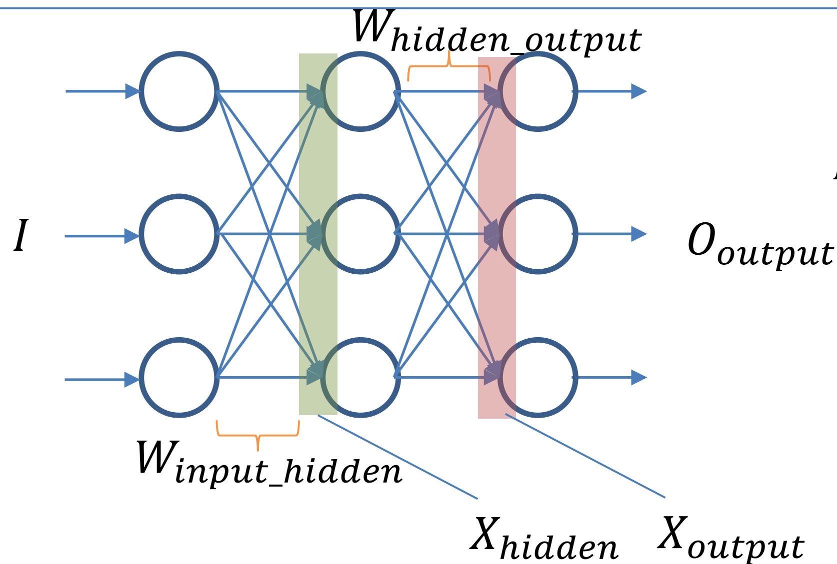

$$\begin{pmatrix} w_{1,1} & w_{2,1} \\ w_{1,2} & w_{2,2} \end{pmatrix} \begin{pmatrix} input_1 \\ input_2 \end{pmatrix} = \begin{pmatrix} input_1 * w_{1,1} + input_2 * w_{2,1} \\ input_1 * w_{1,2} + input_2 * w_{2,2} \end{pmatrix}$$

$$X = W \cdot I$$
$$O = f(X)$$

For example: $O = sigmoid(X)$



# Forward Prop Matrix Example: Get $O$




$$I = \begin{pmatrix} 0.9 \\ 0.1 \\ 0.8 \end{pmatrix} \qquad w_{input_hidden} = \begin{pmatrix} 0.9 & 0.3 & 0.4 \\ 0.2 & 0.8 & 0.2 \\ 0.1 & 0.5 & 0.6 \end{pmatrix}$$

$$w_{hidden_output} = \begin{pmatrix} 0.3 & 0.7 & 0.5 \\ 0.6 & 0.5 & 0.2 \\ 0.8 & 0.1 & 0.9 \end{pmatrix}$$

$$X_{hidden} = W_{input_hidden} \cdot I$$

$$X_{hidden} = \begin{pmatrix} 0.9 & 0.3 & 0.4 \\ 0.2 & 0.8 & 0.2 \\ 0.1 & 0.5 & 0.6 \end{pmatrix} \cdot \begin{pmatrix} 0.9 \\ 0.1 \\ 0.8 \end{pmatrix}$$

$$X_{hidden} = \begin{pmatrix} 1.16 \\ 0.42 \\ 0.62 \end{pmatrix}$$

$$O_{hidden} = sigmoid(X_{hidden})$$

$$O_{hidden} = \begin{pmatrix} 0.761 \\ 0.603 \\ 0.650 \end{pmatrix}$$

$$X_{output} = W_{hidden_output} \cdot O_{hidden}$$

$$X_{output} = \begin{pmatrix} 0.3 & 0.7 & 0.5 \\ 0.6 & 0.5 & 0.2 \\ 0.8 & 0.1 & 0.9 \end{pmatrix} \cdot \begin{pmatrix} 0.761 \\ 0.603 \\ 0.650 \end{pmatrix}$$

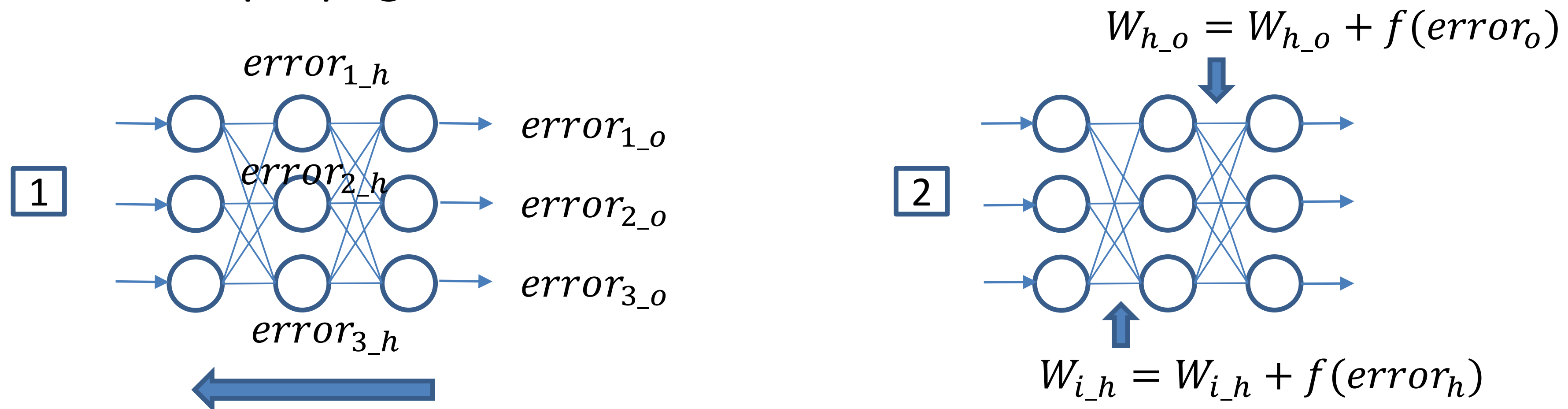
$$X_{output} = \begin{pmatrix} 0.975 \\ 0.888 \\ 1.254 \end{pmatrix}$$

$$O_{output} = sigmoid(X_{output})$$

$$O_{output} = \begin{pmatrix} 0.726 \\ 0.708 \\ 0.778 \end{pmatrix}$$

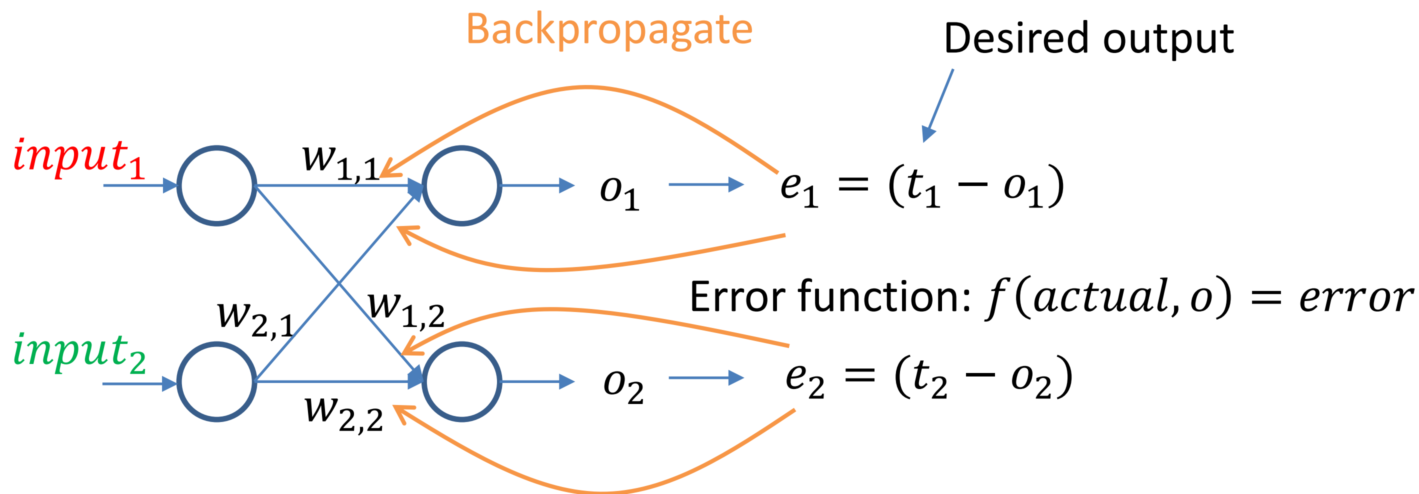
# Learning/Training Process

- Goal: Tune the weights such that the output produces a minimal number of errors
- Two steps are involved:
  1 – Propagate the error back from the output layer to all previous layers
  2 – Update the weights between all layers based on the backpropagated error



# Step 1: Backpropagation – One Layer

- Idea: Propagate the error back proportional to the link weights



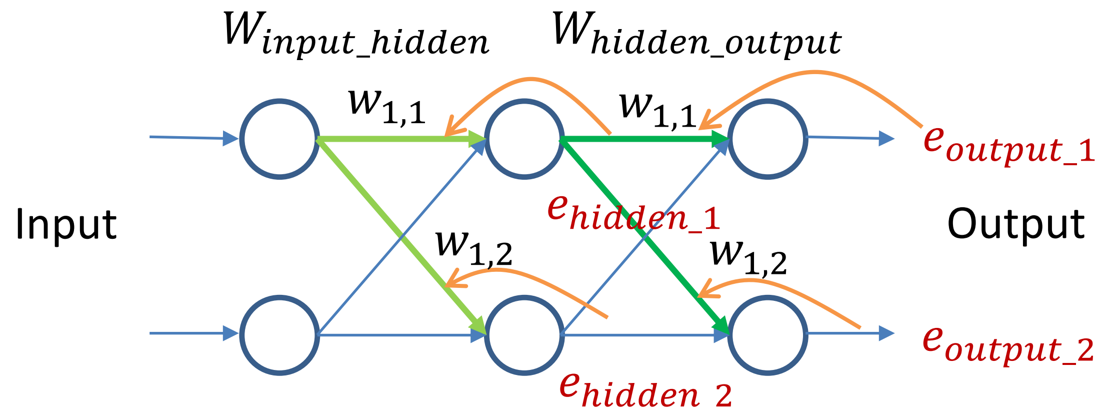

Compute fraction of error, depending on link weights:

Error fraction for $w_{1,1}$ — Error fraction for $w_{2,1}$

$$e_1 = e_1 * \frac{w_{1,1}}{w_{1,1} + w_{2,1}} + e_1 * \frac{w_{2,1}}{w_{1,1} + w_{2,1}}$$

$$e_2 = e_2 * \frac{w_{1,2}}{w_{1,2} + w_{2,2}} + e_2 * \frac{w_{2,2}}{w_{1,2} + w_{2,2}}$$

# Step 1: Backpropagation – More Layers




$$e_{hidden_1} = e_{output_1} * \frac{w_{1,1}}{w_{1,1} + w_{2,1}} + e_{output_2} * \frac{w_{1,2}}{w_{1,2} + w_{2,2}}$$

$$e_{hidden_2} = e_{output_1} * \frac{w_{2,1}}{w_{1,1} + w_{2,1}} + e_{output_2} * \frac{w_{2,2}}{w_{1,2} + w_{2,2}}$$

Matrix formulation:

$$e_{output} = \begin{pmatrix} e_1 \\ e_2 \end{pmatrix} \qquad e_{hidden} = \begin{pmatrix} \frac{w_{1,1}}{w_{1,1} + w_{2,1}} & \frac{w_{1,2}}{w_{1,2} + w_{2,2}} \\ \frac{w_{2,1}}{w_{1,1} + w_{2,1}} & \frac{w_{2,2}}{w_{1,2} + w_{2,2}} \end{pmatrix} \cdot \begin{pmatrix} e_1 \\ e_2 \end{pmatrix}$$

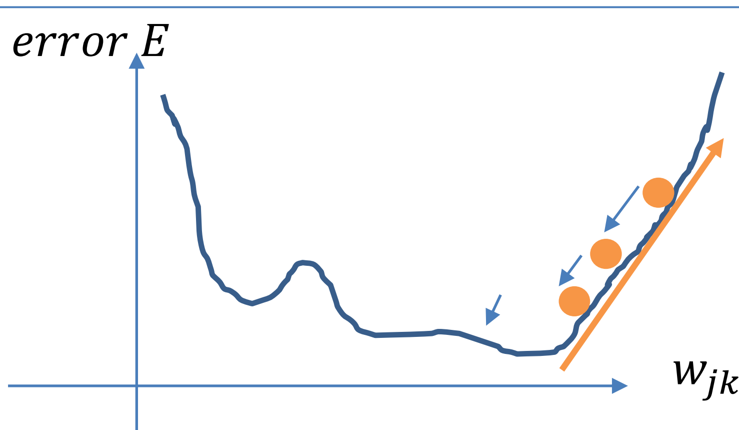
Often, the simpler form suffices: $e_{hidden} = \begin{pmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \end{pmatrix} \cdot \begin{pmatrix} e_1 \\ e_2 \end{pmatrix}$ $\quad e_{hidden} = W_{hidden}^T \cdot e_{output}$

# Step 2: Update the Weights

- Recap: Gradient descent can be used for optimization of parameters
- What are our parameters?
  - The weights of the links
- What do we want to optimize?
  - We want to minimize the error, which depends on the weights

- Gradient descent is the learning technique for neural networks
  - Iteratively update the weights in the opposite direction of the gradient (first derivative) of the error function
  - Use a learning rate to moderate the updates

# Gradient of Weight's Error




$$\frac{\partial}{\partial x} sigmoid(x) = sigmoid(x)(1- sigmoid(x))$$

Simplification: weight $w_{jk}$ depends only on the output of neuron $k$

Error function

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} \sum_n (t_n - o_n)^2 = \frac{\partial}{\partial w_{jk}} (t_k - o_k)^2$$

Output depends on the activation function and the output of the previous layer

Chain rule

Derivative

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial o_k} \cdot \frac{\partial o_k}{\partial w_{jk}} = -2(t_k - o_k) \cdot \frac{\partial o_k}{\partial w_{jk}} = -2(t_k - o_k) \cdot \frac{\partial}{\partial w_{jk}} sigmoid\left(\sum_j w_{jk} \cdot o_j\right)$$

Chain rule

$$\frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) \cdot sigmoid\left(\sum_j w_{jk} \cdot o_j\right)\left(1 - sigmoid\left(\sum_j w_{jk} \cdot o_j\right)\right) \cdot \frac{\partial}{\partial w_{jk}} \sum_j w_{jk} \cdot o_j$$

# Deriving the Derivative

$$\frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) \cdot sigmoid\left(\sum_j w_{jk} \cdot o_j\right)(1 - sigmoid\left(\sum_j w_{jk} \cdot o_j\right)) \cdot \frac{\partial}{\partial w_{jk}} \sum_j w_{jk} \cdot o_j$$

Derivative

$$\frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) \cdot sigmoid\left(\sum_j w_{jk} \cdot o_j\right)(1 - sigmoid\left(\sum_j w_{jk} \cdot o_j\right)) \cdot o_j$$

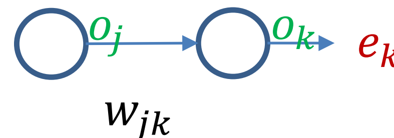
Constant is not necessary and can be omitted for simplicity

Input to node $k$

$$\frac{\partial E}{\partial w_{jk}} = -e_k \cdot sigmoid\left(\sum_j w_{jk} \cdot o_j\right)(1 - sigmoid\left(\sum_j w_{jk} \cdot o_j\right)) \cdot o_j$$

$$\frac{\partial E}{\partial w_{jk}} = -e_k \cdot o_k(1 - o_k) \cdot o_j$$

Depends on the activation function



# Step 2: Actually Update the Weights

New weight / Old weight / Learning rate

$$w_{jk} = w_{jk} - \alpha \cdot \frac{\partial E}{\partial w_{jk}}$$ ← Gradient of the error

Matrix formulation:

Sigmoid function

$$\begin{pmatrix} \Delta w_{1,1} & \cdots & \Delta w_{n,1} \\ \vdots & \ddots & \vdots \\ \Delta w_{1,m} & \cdots & \Delta w_{n,m} \end{pmatrix} = \alpha * \begin{pmatrix} e_1 * S_1(1 - S_1) \\ \vdots \\ e_n * S_n(1 - S_n) \end{pmatrix} \cdot (o_1 \quad \ldots \quad o_m)$$

Values from previous layer

Values from next layer

$$\Delta w_{jk} = \alpha * E_k * sigmoid(X_k) * (1 - sigmoid(X_k)) \cdot O_j^T$$

Hint:

$$O_k = sigmoid(X_k)$$

Further tasks:

Update weights after each training case (online), all training cases (full batch), a small sample of training cases (mini-batch)

# (Some)
# Network Architectures

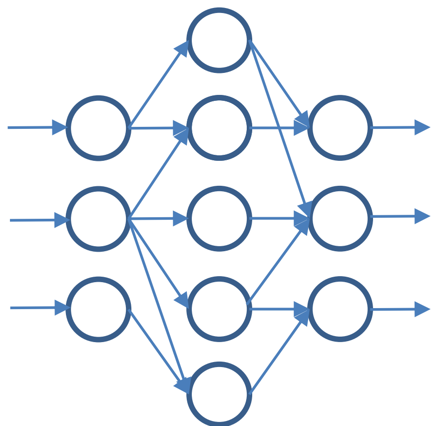
# Architectures Overview

- Architecture has a huge influence on what can be learned efficiently (and maybe at all), as in nature
- Different architectures for different tasks, such as object recognition vs. sequence prediction

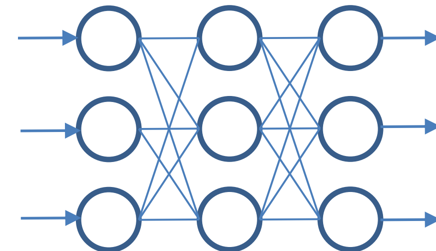
- The way how neurons are connected, how many layers exist, and how many neurons per layer defines the architecture
  - Often, the activation function is part of the architecture and might vary from layer to layer
  - Complex architectures are composed from very different types of layers

# Feed-forward Network

- Common network type
  - Just pass the input from the input layer forward through the network
  - If we have multiple hidden layer, it is called deep neural network

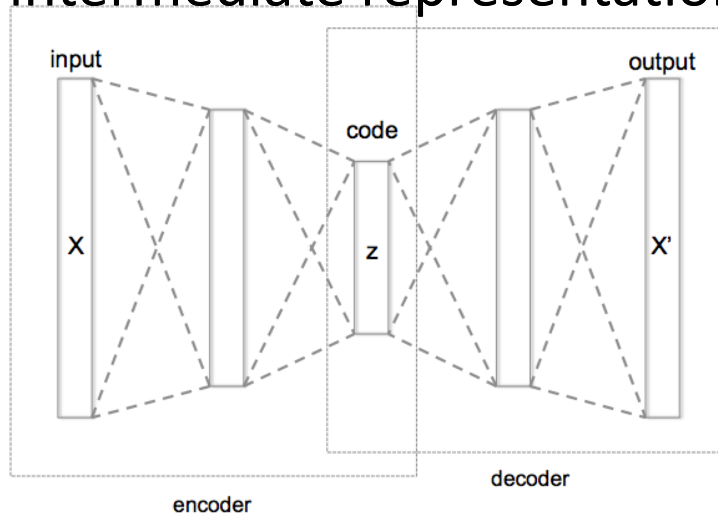


Fully-connected layers: each node in a layer has a connection to each node in the next layer



# Auto-Encoders

- Used for unsupervised learning of efficient encoding the input and generative models
  - Dimensionality reduction technique from size input to size z
  - Compression capabilities
  - The decoding part aims at reproducing the result from the compressed intermediate representation

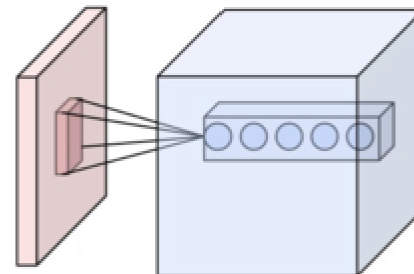


# Convolutional Neural Networks

- Supervised learning to classify and tag images
  - Inspired by biological processes, where the connectivity pattern among neurons is based on certain activation areas
  - Learns itself key (low and high level) features of images
- Usually 3 types of hidden layers:
  - Convolution layer
  - Pooling/Max pooling Fully-Connected: non-linearity activation functions (e.g., tanh, ReLU, sigmoid)

# Convolution Layer

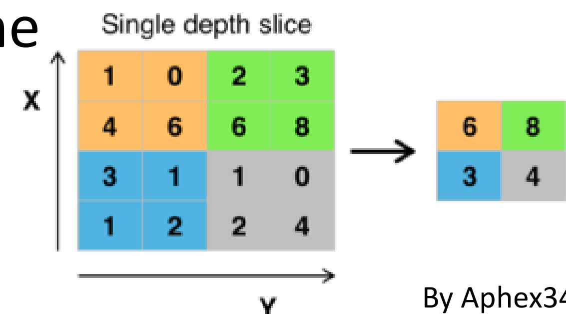
- Neurons of the convolutional layer extract spatial features
  - Each neuron process data only for a predefined area (e.g., 3x3 tile) of the image
  - Filter the picture for certain low-level properties, such as edges, etc.
- Idea: Later layers combine spotted edges and build higher order features (e.g., a house has spotted edges in a certain geometrical order)



By Aphex34

# Pooling Layer

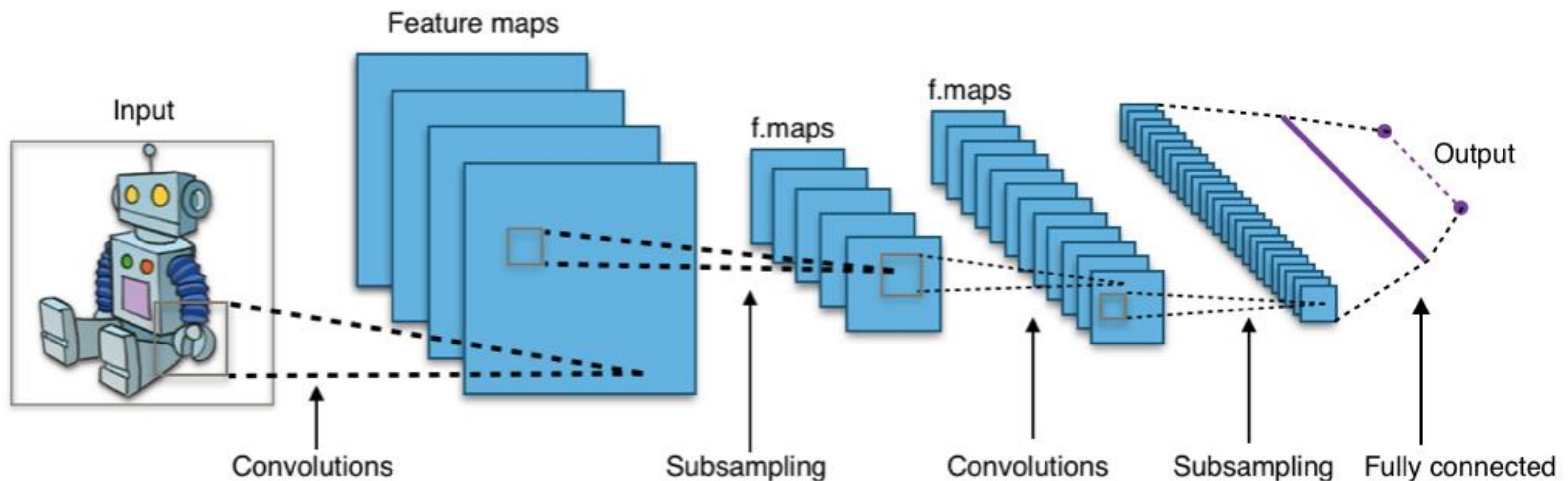
- Sub-(or down)sample spatial feature maps
  - Combine the output of neuron clusters from previous layers into a single neuron
  - Max pooling uses the maximum value from a cluster's neurons from the previous layer)
- Idea: Exact location of a feature is less important than is broad location relative to other features
  - Reduces the size of the spatial representation
  - Avoids overfitting, improves learning time




By Aphex34

# ReLU / Fully-Connected Layer

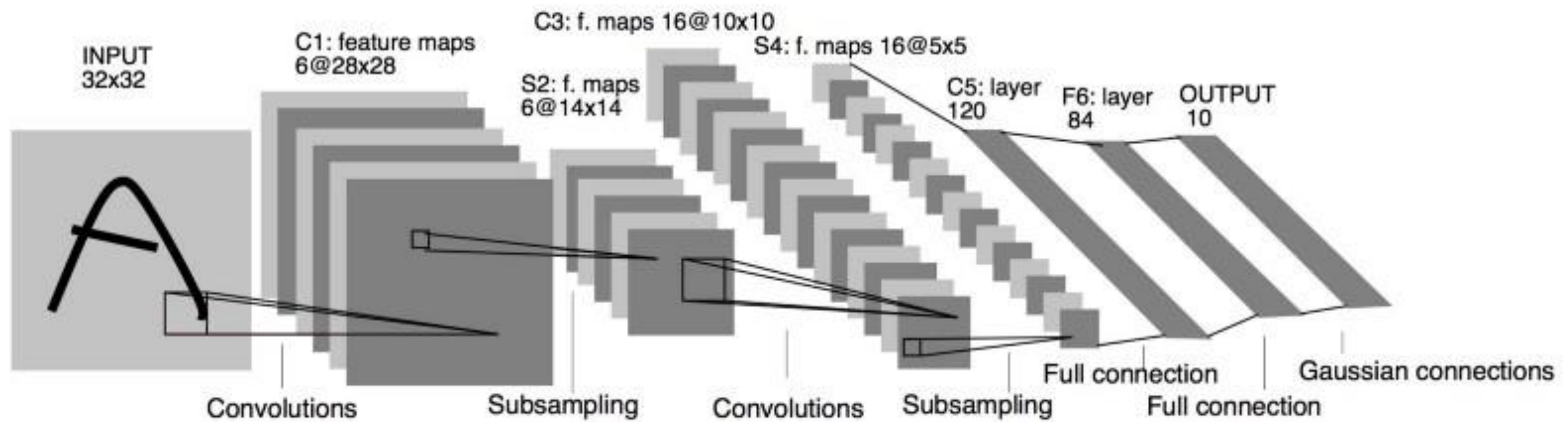
- Recap ReLU: f(x) = max(0,x)
- Other non-linear activation functions are possible as well (e.g., tanh, sigmoid)



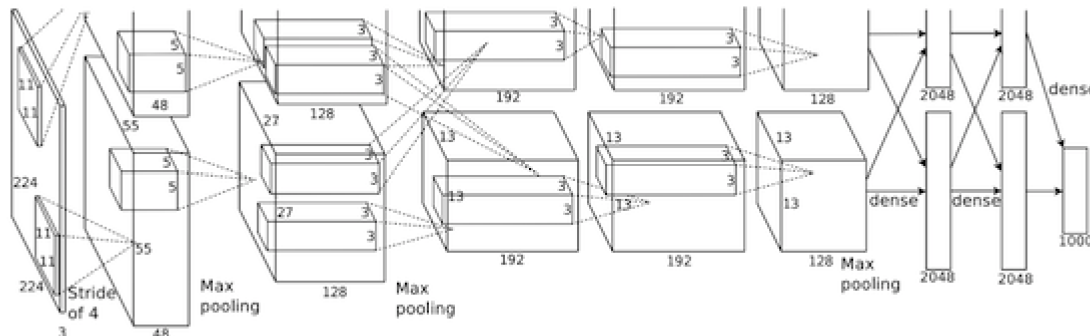

By Aphex34

# Examples of CovNets

LeNet5 from Yann LeCun in 1994: One of the very first convolutional neural networks




AlexNet from Alex Krizhevsky in 2012: Won the ImageNet competition with huge margin

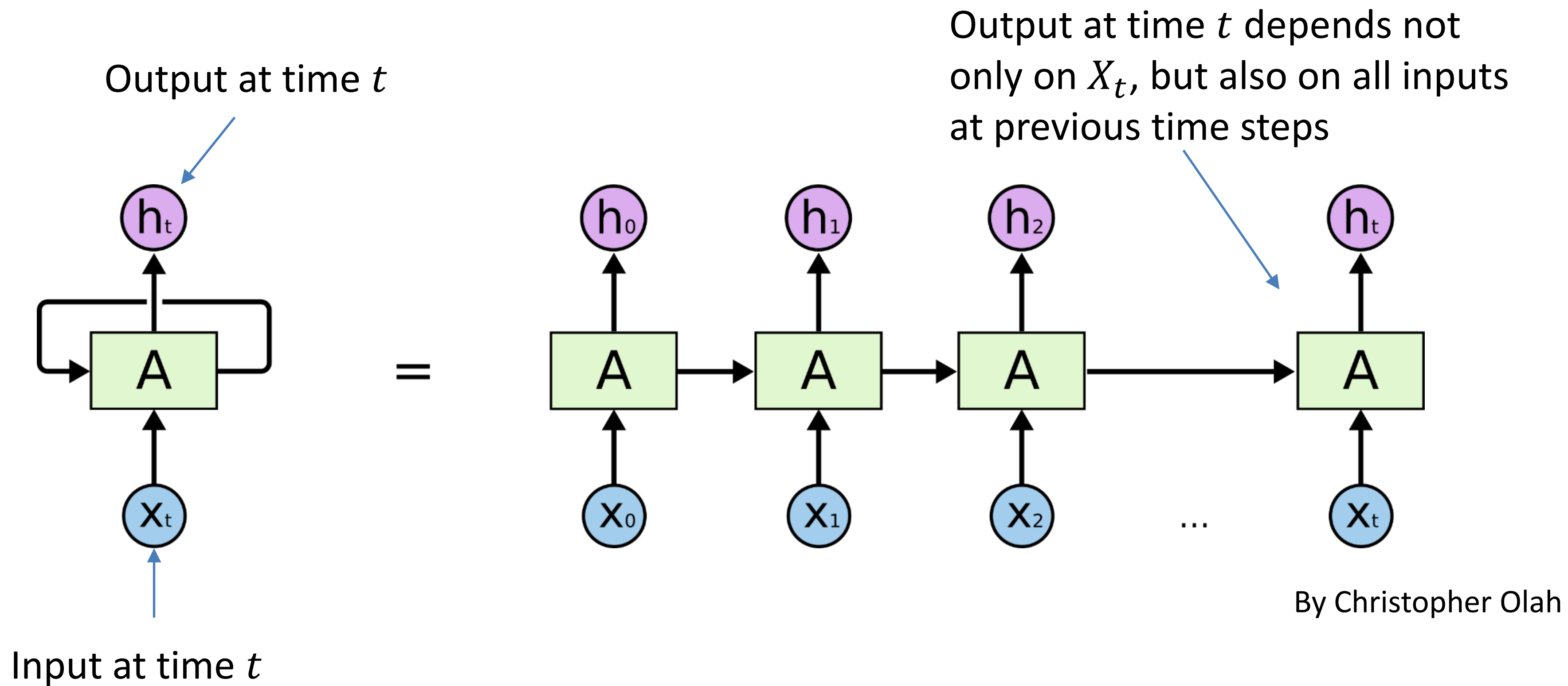


# Recurrent Neural Networks (RNNs)

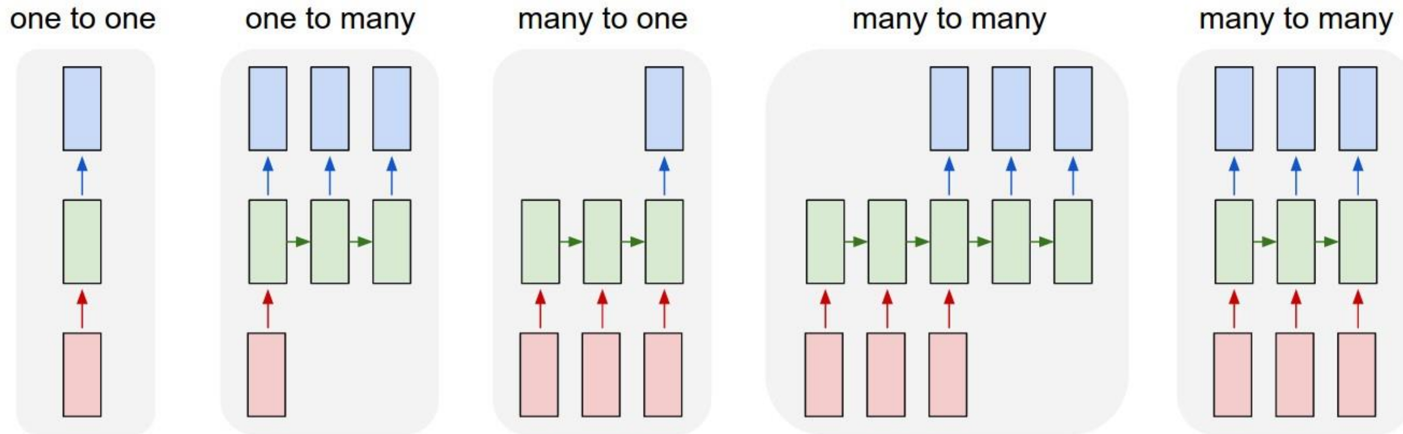
- Supervised learning used for learning sequences, temporal behavior and predictions (dev. by John Hopfield 1982)
  - Connections from a directed cycle
  - Have an internal memory (with limited capacity)
  - Can process arbitrary sequences of inputs

- Many different variations
  - Recursive, LSTM, Hopfield, fully recurrent, encoder-decoder
- Training for RNNs is often done using genetic algorithms
  - Mean squared error is the fitness function

# Structure of an RNN

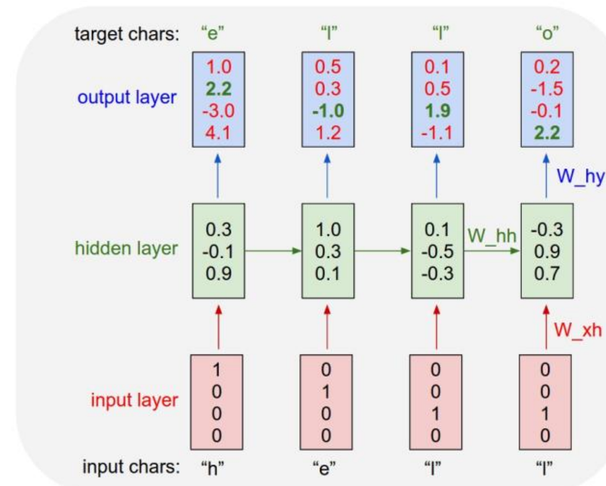


# Different Structures Possible




By Andrej Karpathy

No fixed-size input nor fixed-size output
No fixed amount of computation steps

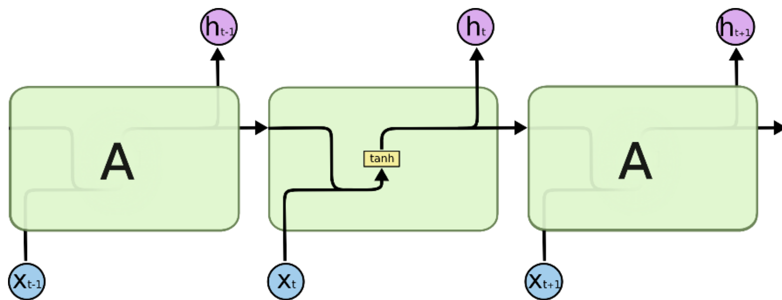



Great blog post (a must read):http://karpathy.github.io/2015/05/21/rnn-effectiveness/

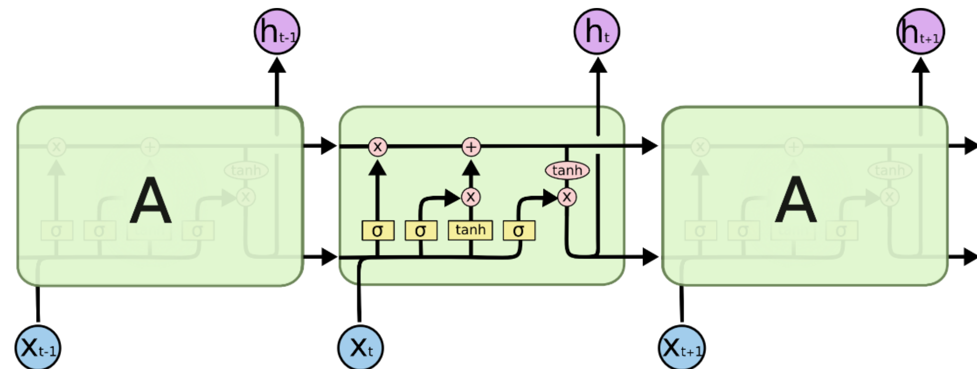
# Long Short-Term Memory (LSTM)

- Problem in RNNs: Long-term dependencies (context gets lost the more time steps we are away from relevant information)
- LSTMs were introduced by Hochreiter & Schmidhuber (1997) to remember information for long periods of time

RNN

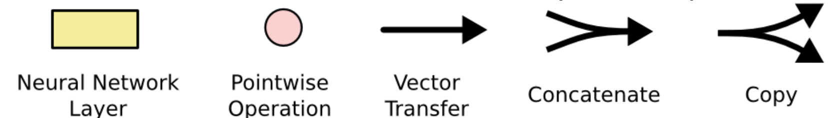



LSTM



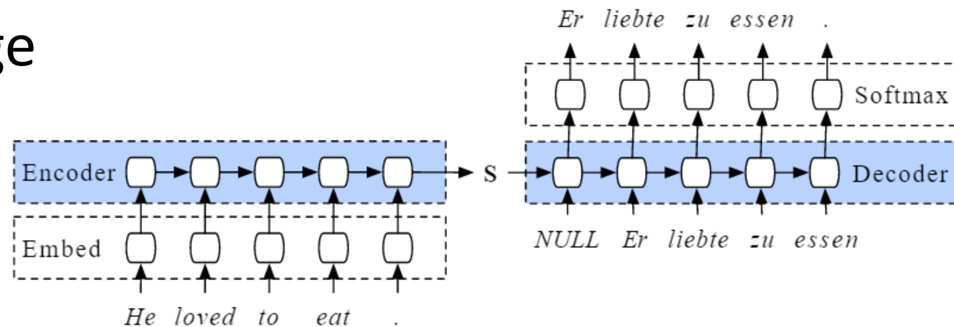

By Christopher Olah

Key idea: Ease information flow through cells



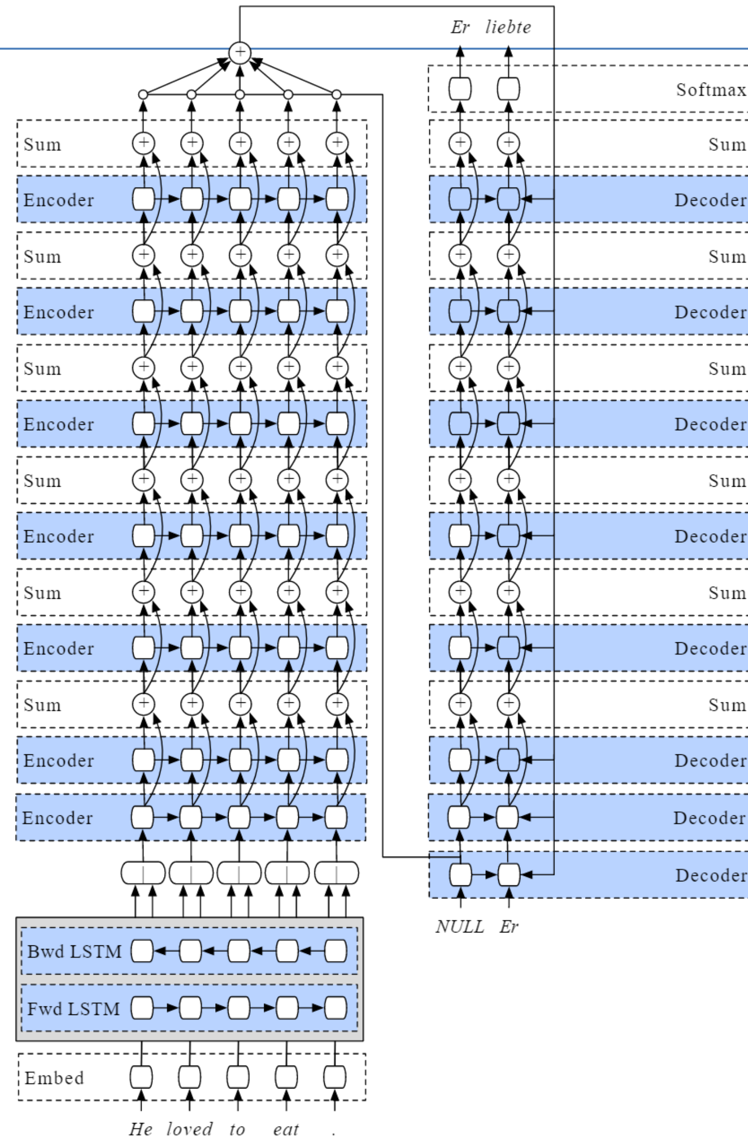
# Encoder-Decoder

- Often used for language translation
- Consists of two components: encoder and decoder
  - RNN (LSTM) to encode a language resulting in hidden state s
  - Decode (LSTM) uses the state s and decodes it to a sentence in another language



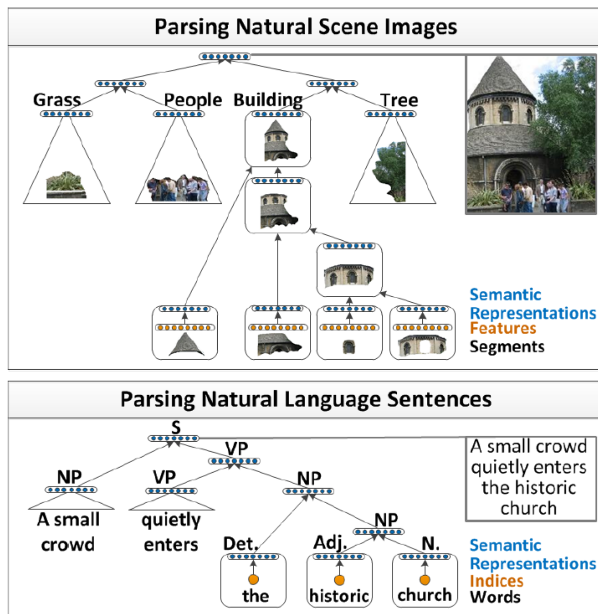

- Drawbacks:
  - Limited memory (the whole sentence must be represented in the single state s)
  - Very deep network architecture == hard to train

# Google Neural Machine Translation

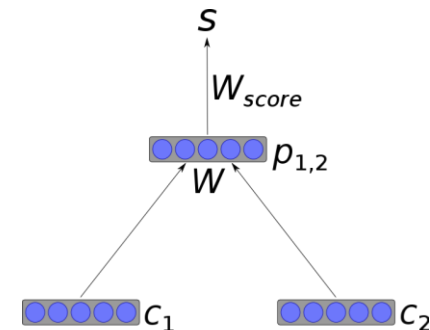


# Recursive Neural Networks (rNN)

- Unsupervised and supervised learning to produce structured prediction over input sequences
  - Traverses a given structure in topological order
  - Can learn tree structures, distributed representations of structures



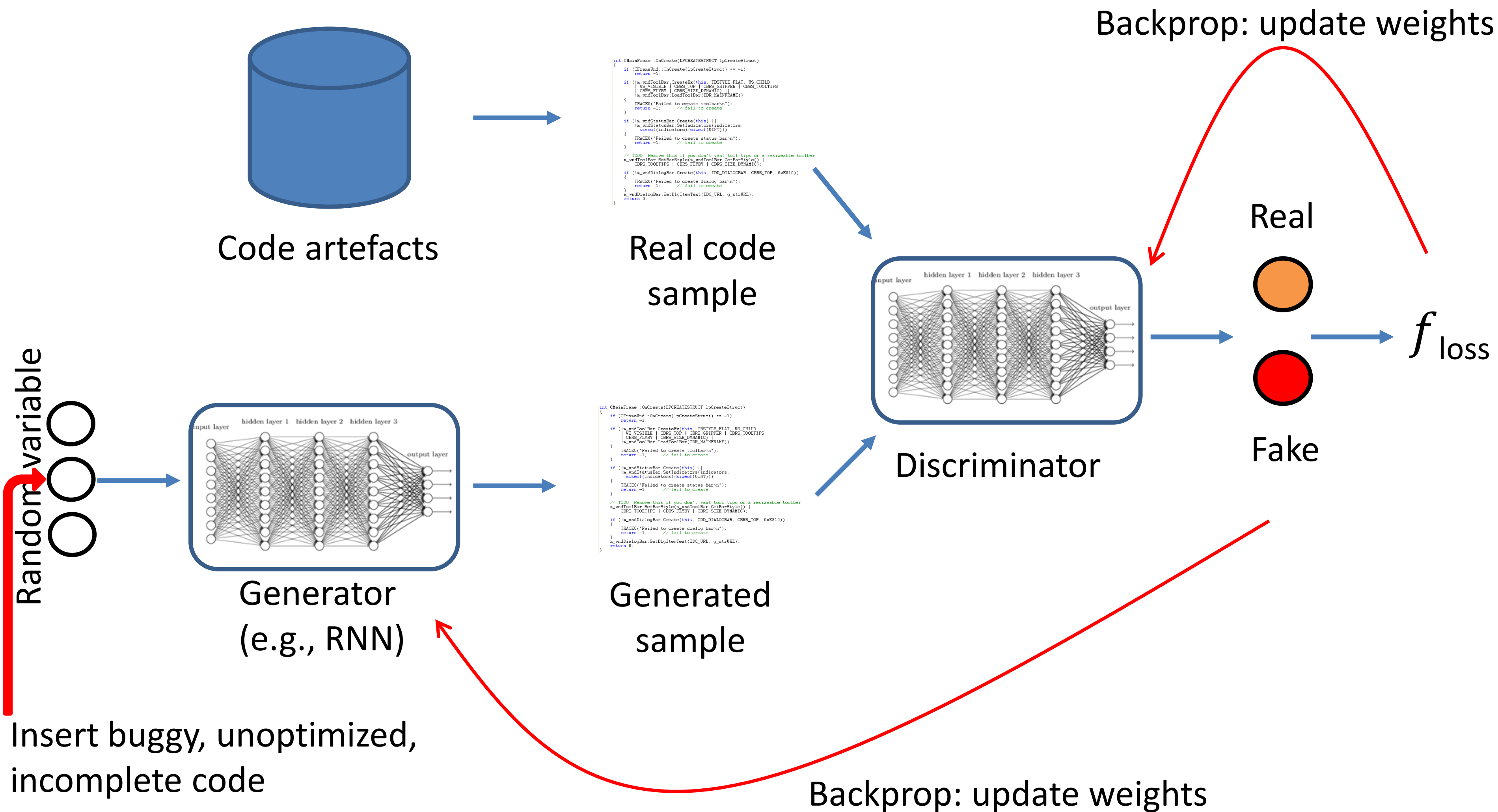

Socher et al.: Parsing Natural Scenes and Natural Language with Recursive Neural Networks




By Daniel Hershcovich

# Where to go next?

# Generative Adversarial Networks



# Optimizing the Learning

- Stochastic gradient descent
- Mini batches
- Weight initialization
- Momentum
- Separate, adaptive learning rate per connection
- And others, such as rmsprop (sliding window of average gradient)

# Take Home Message:

# Questions and Summary Next Lecture!

- We have made it! ☺
- Prepare for the exam!
- Do your project! ☺

# Literature

- Make Your Own Neural Network (English) by Tariq Rashid
- http://www.deeplearningbook.org/