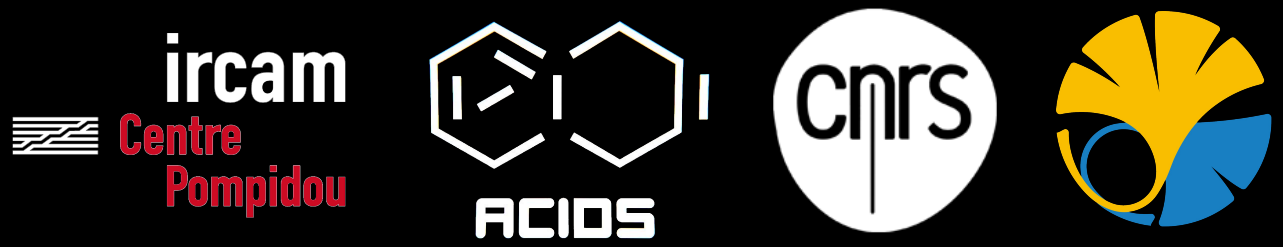


Neural networks

Creative Machine Learning - Course 02

Pr. Philippe Esling
esling@ircam.fr

Brief history of neural networks

Originally neuroscience

Modeling the structure and function of the human brain
Interconnected neurons that transmit and process information.

Engineering side

An attempt to mimic « intelligence » in biology
So why modeling biological neurons ?...
Well it is the only « intelligent » thing we know ☺️

A quite rocky history

First mathematical model of neuron by McCulloch & Pitts (1943)
Went into disappearance until backpropagation algorithm (1986)
Sparked back the interest ~1990

Goals

1. **Approximate complex functions**
   Can learn to approximate any non-linear functions **(universal approximator theorem)**.
2. **Automatically discover relevant features**
   Learn to extract representation from raw data, eliminates manual feature engineering.
3. **Adapt and generalize to new data**
   Generalize learned knowledge to handle previously unseen examples.

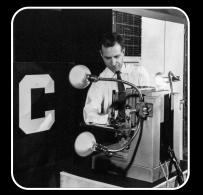
Current trend

Currents popularity of neural networks in deep learning
- Advances in hardware (e.g., GPUs),
- Availability of large-scale datasets
- Improved training algorithms.

Brief history of AI

1943 - Neuron

First model by McCulloch & Pitts (purely theoretical)



This lesson

1957 - Perceptron

Actual **learning machine** built by Frank Rosenblatt

Learns character recognition analogically

This lesson

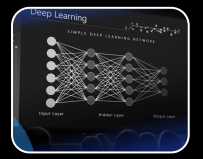
1986 - Backpropagation

First to learn neural networks efficiently (*G. Hinton*)

Lesson #3

1989 - *Convolutional NN*

Mimicking the vision system in cats (*Y. LeCun*)



Lesson #4

2012 - *Deep learning*

Layerwise training to have deeper architectures

Swoop all state-of-art in classification competitions

Lesson #6

2015 - *Generative model*

First wave of interest in generating data

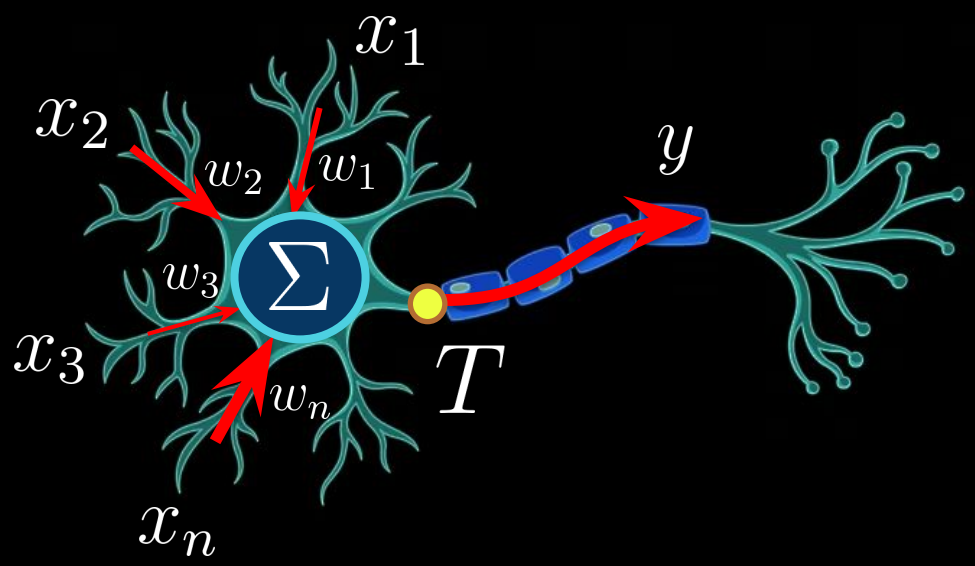
Led to current model craze (VAEs, GANs, Diffusion)

2012 onwards Deep learning era

Artificial neuron

Neuron model (McCulloch & Pitts - 1943)

Basic building block of neural networks



$$\mathbb{I}_{>T}\left(\sum_{i=1}^{n} w_i.x_i\right) \qquad \mathbb{I}_{>T}(x) = \begin{cases} \mathbf{0} \text{ if } x \leq T \\ \mathbf{1} \text{ if } x > T \end{cases}$$

So what does a single neuron mathematically do ?

Understanding a single neuron

$$\mathbb{I}_{>T}\left(\sum_{i=1}^{n} w_i.x_i\right) \longrightarrow y = \left(\sum_{i=1}^{n} w_i.x_i + T\right)$$

So what is the **interpretation** of this equation?

Say we only have a **single weight** ?

$$y = w_1.x_1 + T$$

When $\mathbf{x} \in \mathbb{R}^n$ a neuron is a N-dimensional hyperplane

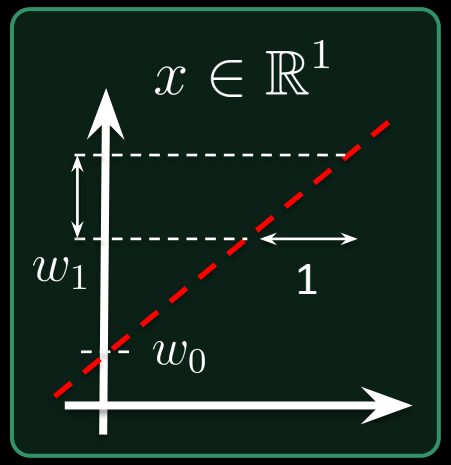
$$y = w_1.x_1 + w_2.x_2 + T$$

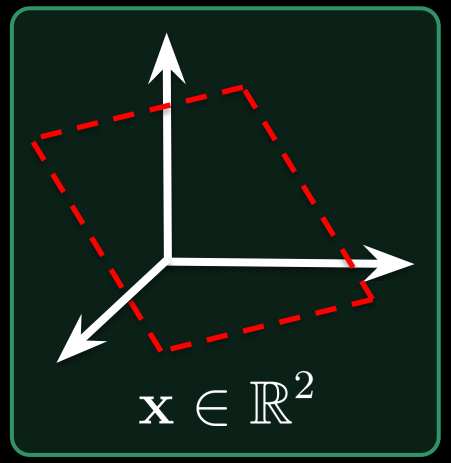
So essentially a space division but also function approximation

as the sum will give an output whatever X comes in

Rings any bell ?

Geometrical insight





Simplified neuron is linear regression

Neuron in a simplified form

Model the relationship between **one** output (target) and one or more inputs (features).

$$\underbrace{\mathbf{x} \in \mathbb{R}^n}_{\textit{Input (feature)}} \quad \mathcal{X} \xrightarrow{f_\theta} \mathcal{Y} \quad \underbrace{y \in \mathbb{R}}_{\textit{Output (target)}}$$

Akin to linear regression

$$y = \left(\sum_{i=1}^{n} w_i . x_i + T \right)$$

Similar loss functions

$$\mathcal{L}_{MSE}\left(\bar{\mathbf{y}}, \theta\right) = \sum_{i=1}^{n} \left| y_i - \bar{y}_i \right|^2$$

One key difference to keep in mind

$$\mathbb{I}_{>T}\left(\sum_{i=1}^{n} w_i . x_i \right)$$

Neurons as space division

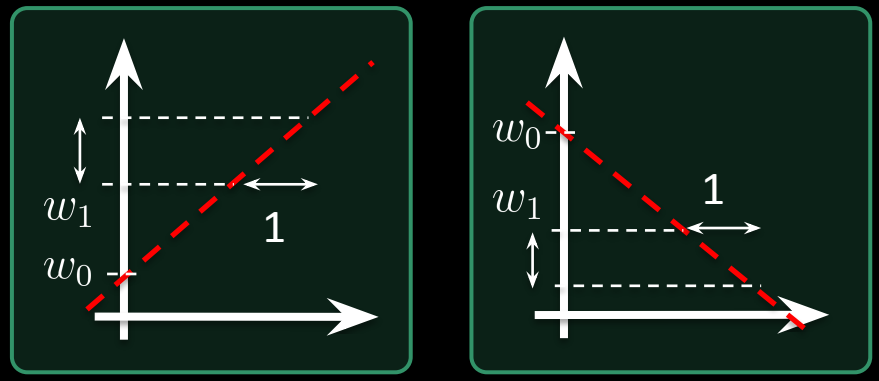
Elegant interpretation

- Neurons simply learn to divide space
- Neurons allow to perform **classification**
- But also **function approximation**

For now, only *linearly separable problems*

Neuron learning

Tweaking the neuron weights



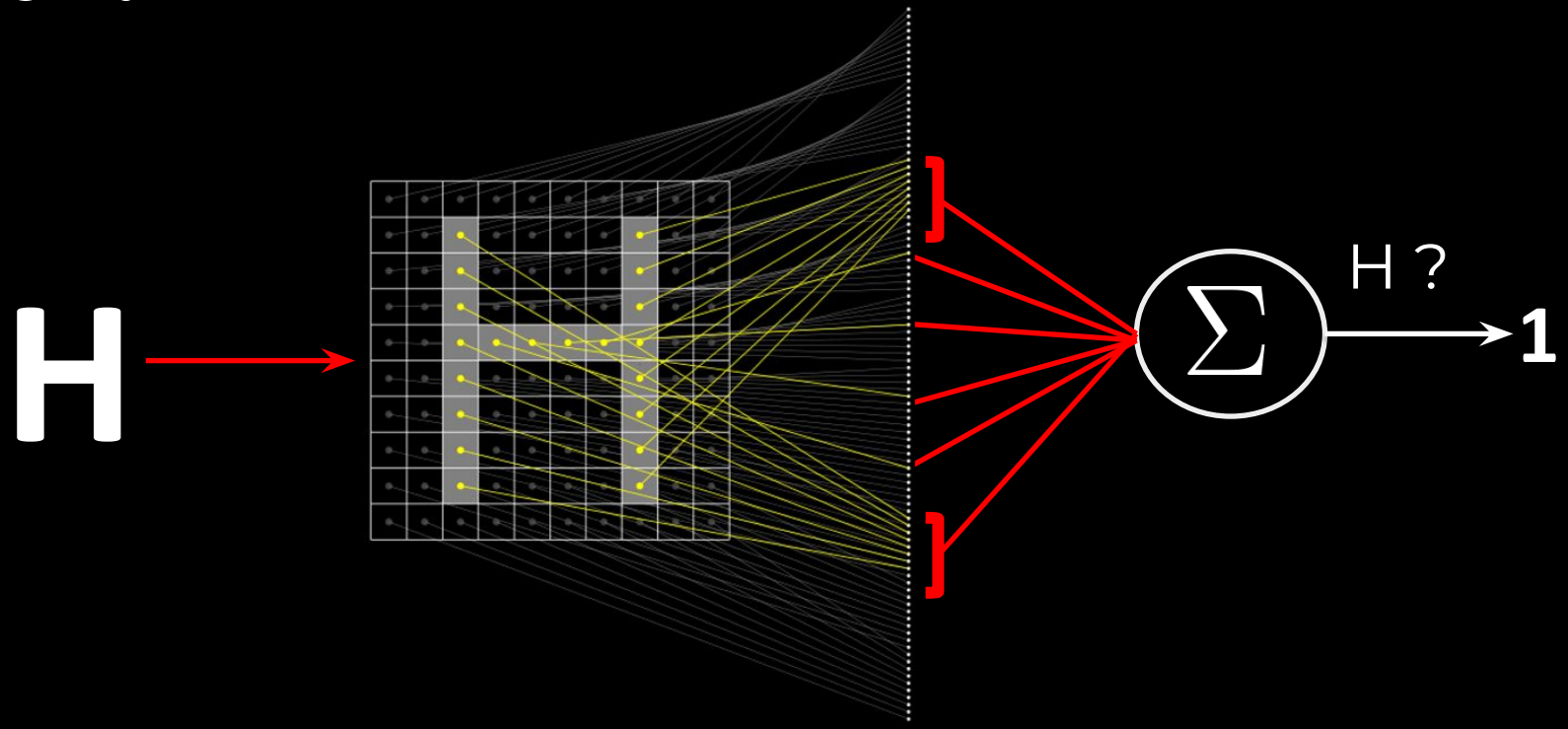
Ability to perform any space division

Yet only **linear separation**

Neurons as feature detectors

Alternative interpretation of neural networks

Of course, slightly idealized case



Interpretation more adapted to *convolutional* networks

Also interesting for allowing *interpretability*

We will see that over the next course

Recalling gradient descent

Applied to our neuron

1 - Initialize parameters

Start from random point $\mathbf{w} \sim \mathcal{N}(\mu_0, .01)$

2 - Compute partial derivatives

Derive the loss based on all parameters

$$\nabla \mathcal{L}(\mathbf{w}) = \left[\frac{\partial \mathcal{L}}{\partial w_1}, \frac{\partial \mathcal{L}}{\partial w_2}, ..., \frac{\partial \mathcal{L}}{\partial w_n} \right]$$

3 - Iterative updates

In each iteration update the weights

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \nabla \mathcal{L}(\mathbf{w}^t)$$

Two key problems

#1 - Threshold

#2 - Discontinuous

$$\mathbb{I}_{>T} \left(\sum_{i=1}^{n} w_i . x_i \right)$$

From neuroscience to optimization

Problem #1 There is a threshold T for each neuron

Solution: *We consider T as a virtual weight called bias (b)*

Problem #2 Gradient descent requires smooth (derivable) function

$$\mathbb{I}_{>0}(x) \in C^0$$

Solution: *Replace with a smooth function*

Activation function $\phi(x) \in C^1$

$$\mathbb{I}_{>T}\left(\sum_{i=1}^{n} w_i.x_i\right) \longrightarrow \mathbb{I}_{>0}\left(\sum_{i=1}^{n} w_i.x_i + b\right) \longrightarrow \phi\left(\sum_{i=1}^{n} w_i.x_i + b\right)$$

Activation functions

Sigmoid

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad \sigma(x) \in (0, 1)$$

Smooth and differentiable but suffers from vanishing gradient

Hyperbolic tangent

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \quad tanh(x) \in (-1, 1)$$

Also vanishing gradient problem but wider output range

Rectified Linear Unit (ReLU)

$$\mathrm{ReLU}(x) = \max(0, x) \quad \mathrm{ReLU}(x) \in [0, \infty)$$

Computationally efficient and solves the vanishing gradient

Already *"deep learning era"*, more to come

Applying gradient descent

Model | Our model for a single neuron computes

$$\bar{\mathbf{y}} = f_\theta(\mathbf{x}) = \phi\left(\sum_{i=1}^{n} w_i.x_i + b\right)$$

Loss function | Still rely on MSE loss function $\mathcal{L}(\mathbf{y}, \bar{\mathbf{y}}) = \sum_i (y_i - \bar{y}_i)^2$

Optimizing our single neuron

Need to compute gradients of a quite complicated (non-linear) function

We need gradients $\frac{\delta \mathcal{L}}{\delta \theta_i}$ of the loss given $\theta = \{w_i, b\}$

Hence, we should perform derivation of

$$\mathcal{L}_{MSE}(\bar{\mathbf{y}}, \theta) = \sum_{i=1}^{n} \left| y_i - \phi\left(\sum_{i=1}^{n} w_i.x_i + b\right)\right|^2$$

Backpropagation

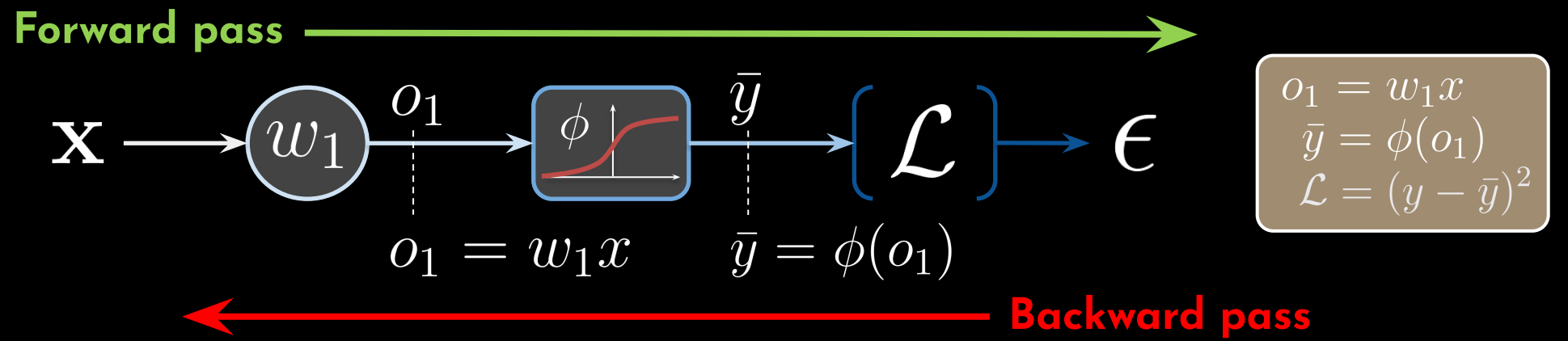
Efficiently rely on the **chain rule of calculus**

Backpropagation compute gradients of complex functions

The algorithm is based on the **chain rule of calculus**.

$$\begin{array}{l} u = g(x) \\ y = f(u) \\ y = f(g(x)) \end{array} \qquad \frac{\partial y}{\partial x} = \frac{\partial y}{\partial u} \cdot \frac{\partial u}{\partial x}$$

Observing the chain rule (*backpropagation*) in the simplest setting



$$\begin{array}{l} o_1 = w_1 x \\ \bar{y} = \phi(o_1) \\ \mathcal{L} = (y - \bar{y})^2 \end{array}$$

$$\frac{\delta \mathcal{L}}{\delta w_1} = \frac{\delta \mathcal{L}}{\delta \bar{y}} \cdot \frac{\delta \bar{y}}{\delta w_1} \cdot \frac{\delta \bar{y}}{\delta o_1} \cdot \frac{\delta o_1}{\delta w_1} \cdot x$$

$2(y - \bar{y})$

$\frac{\partial \phi(x)}{\partial x}$

Final “composed” gradient

$$\frac{\delta \mathcal{L}}{\delta w_1} = 2(y - \bar{y})\phi'(x)x$$

Implementing backpropagation

How to code your own artificial neuron

1. Initialize w_i to small random values
2. While !(stop condition)
   1. Present patterns $\mathbf{X} = \{\mathbf{x}_1, \cdots, \mathbf{x}_n\}$ to neuron
   2. **Propagation** (forward): Evaluate output $\bar{y}$ and error ϵ
   3. **Back-propagation**: Update the weights based on

Stop conditions
- Iterations number
- $\mathcal{L} \geq \epsilon$
- $\delta\mathcal{L} \geq \tau$

$$w_i^{t+1} = w_i^t + \eta . \frac{\delta \mathcal{L}}{\delta w_i}$$

learn rate ([0,1])

$$\frac{\delta \mathcal{L}}{\delta w_i} = 2(y - \bar{y})\phi^{'}(x_i)x_i$$

$$w_i^{t+1} = w_i^t + \eta . 2(y - \bar{y})\phi^{'}(x_i)x_i$$

- We can do this for each example x_i separately (*Stochastic* GD)
- However, gradient is « noisy », so we rather rely on *batches* of data

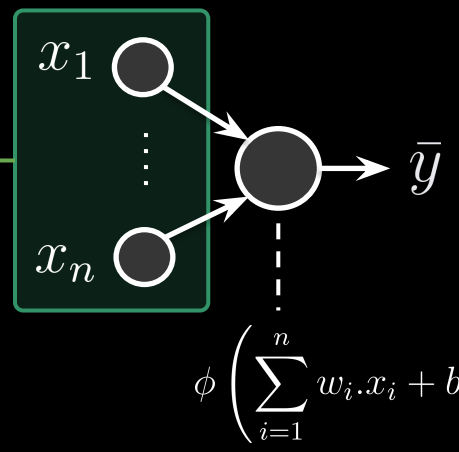
Extends quite simply to multi-layer networks

Multi-Layer Perceptron (MLP)

Feedforward neural network are composed of multiple **layers** of our **neurons**

Structure of a Multi-Layer Perceptron (MLP)

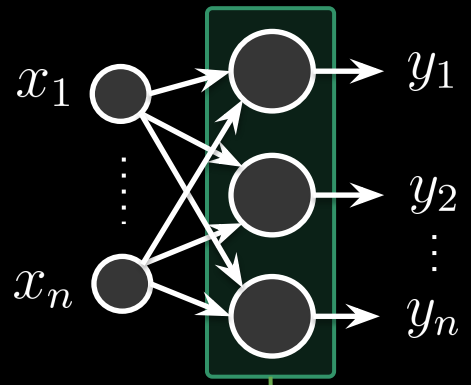
Single neuron



Artificial neuron
as seen previously

Input layer: $\mathbf{x} = [x_1, x_2, \ldots, x_n]$
corresponds to input features

*Single **layer***

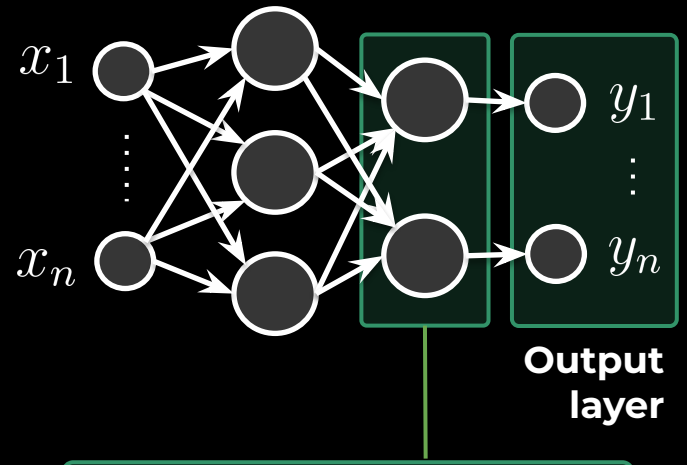


Hidden layer
set of neurons applied to input.

$$o_m^l = \phi\left(\mathbf{w}_m^l * \mathbf{x} + b_m^l\right)$$

Expressed in **matrix notation**

Multi-layer** perceptron **(MLP)



Output layer

Multiple layers
Input of each layer is output of the previous layer

$$\mathbf{o}^l = \phi\left(\mathbf{W}^l * \mathbf{o}^{l-1} + \mathbf{b}^l\right)$$

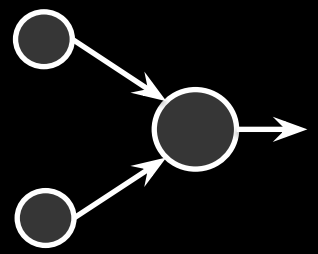
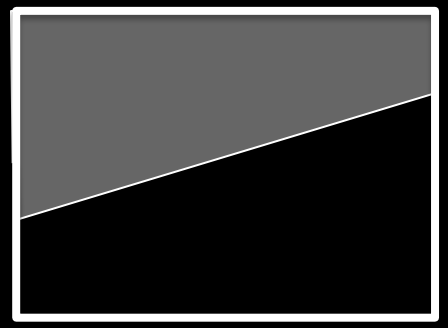
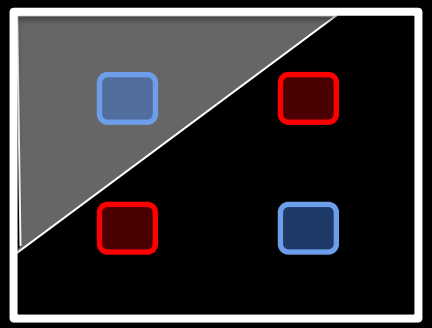
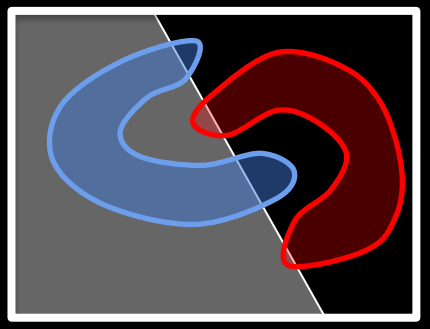
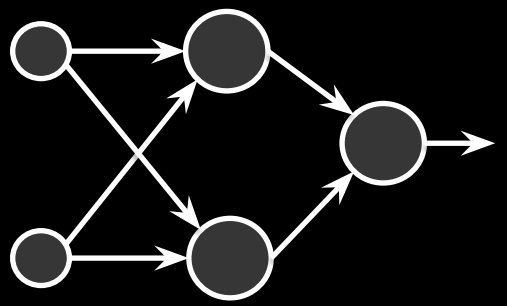
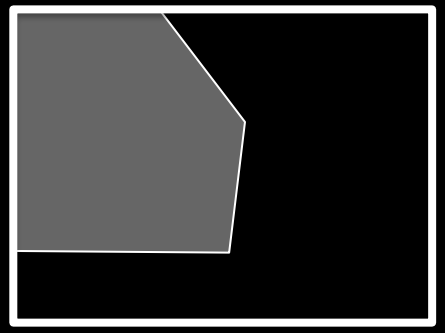
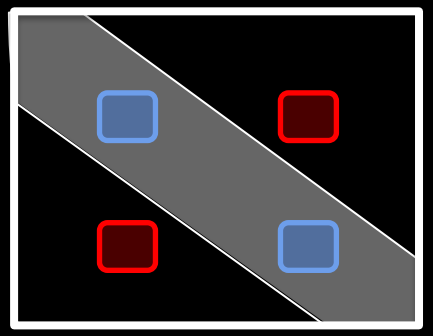
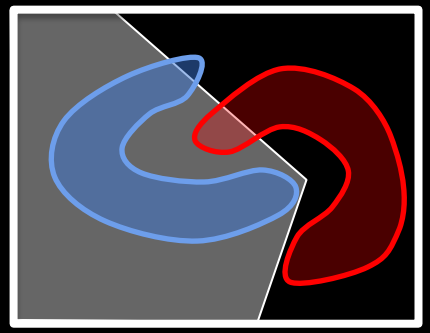
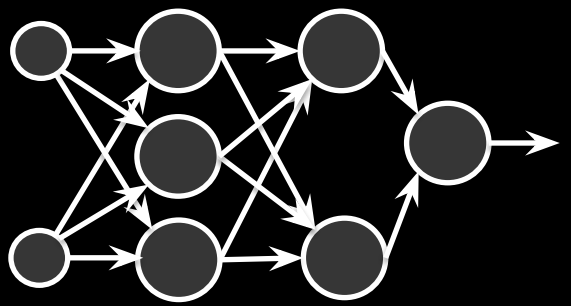
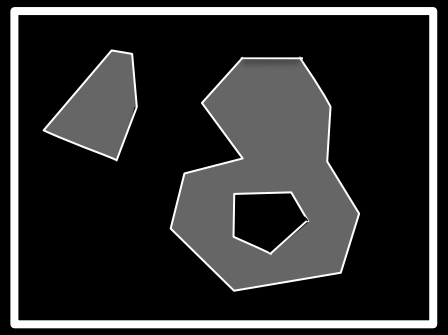
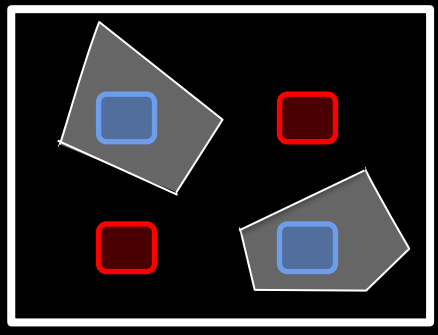
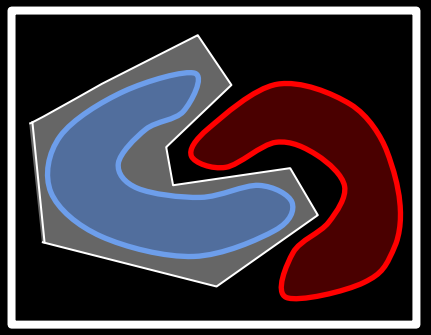
Called **forward propagation**

Multi-Layer Perceptron (MLP)

Feedforward neural network are composed of multiple **layers** of our **neurons**

ANIMATION SHOWING MULTIPLE LAYERS
(Starting from a single neuron)

Multiple layers

	Architecture	Space region	XOR Problem	General shapes
Single layer				
Two layers				
Three layers				

Importance of non-linearities

Problem | Multi-layer **linear** networks do not learn *more complex* functions.
Composition of linear functions is still a linear function.

Suppose we have two linear transforms $\mathbf{A} : \mathbb{R}^n \to \mathbb{R}^m \quad \mathbf{B} : \mathbb{R}^m \to \mathbb{R}^p$

If we define the composition of these two transforms $\mathbf{C} = \mathbf{B}(\mathbf{A}(\mathbf{x}))$

Then it is also a linear transform

$$\begin{aligned}\mathbf{C}(\alpha\mathbf{x} + \beta\mathbf{y}) &= \mathbf{B}(\mathbf{A}(\alpha\mathbf{x} + \beta\mathbf{y})) \\ &= \mathbf{B}(\alpha\mathbf{A}(\mathbf{x}) + \beta\mathbf{A}(\mathbf{y})) \\ &= \alpha\mathbf{B}(\mathbf{A}(\mathbf{x})) + \beta\mathbf{B}(\mathbf{A}(\mathbf{y})) \\ &= \alpha\mathbf{C}(\mathbf{x}) + \beta\mathbf{C}(\mathbf{y})\end{aligned}$$

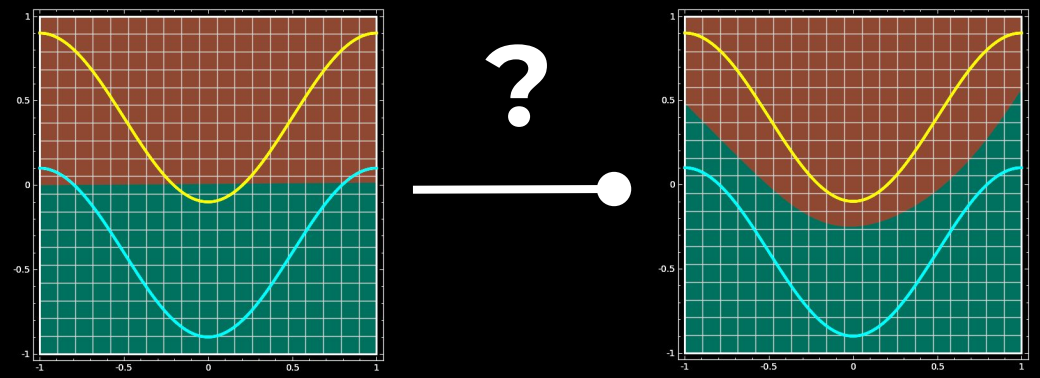
Therefore, the composition of two linear transforms is still a linear transform.

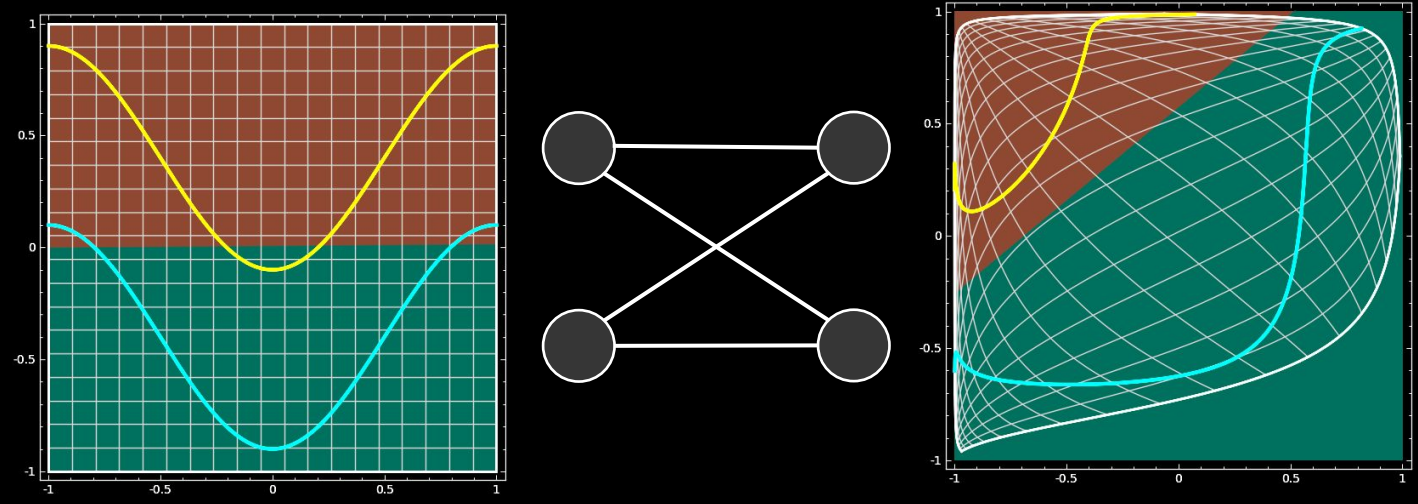
Therefore, **N linear layers are equivalent to a single linear layer**.

Non-linear activation functions introduce non-linearity into the model
Allowing the network to learn more complex patterns.

Networks as space transform

Extending our geometrical insight to **non-linearities** ?





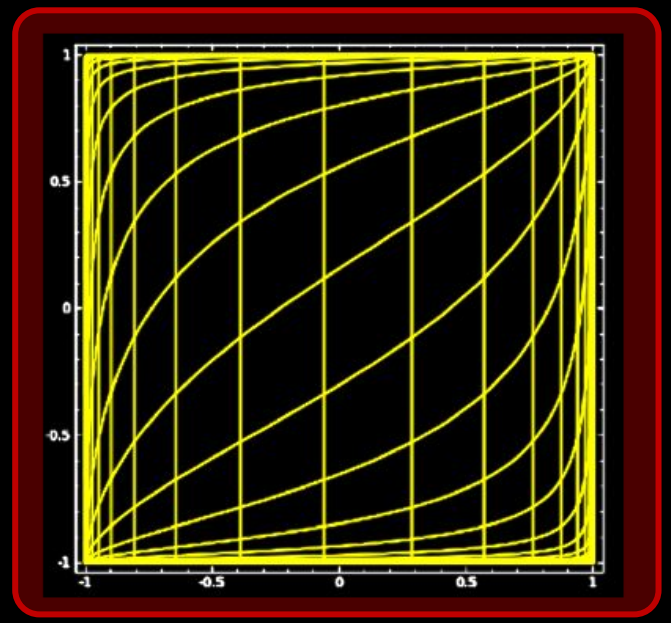
$$\mathbf{o}^l \in \mathbb{R}^2 \quad \mathbf{o}^{l-1} \in \mathbb{R}^2$$

Interpreting our previous equation

$$\mathbf{o}^l = \phi\left(\mathbf{W}^l * \mathbf{o}^{l-1} + \mathbf{b}^l_m\right)$$

Affine transform

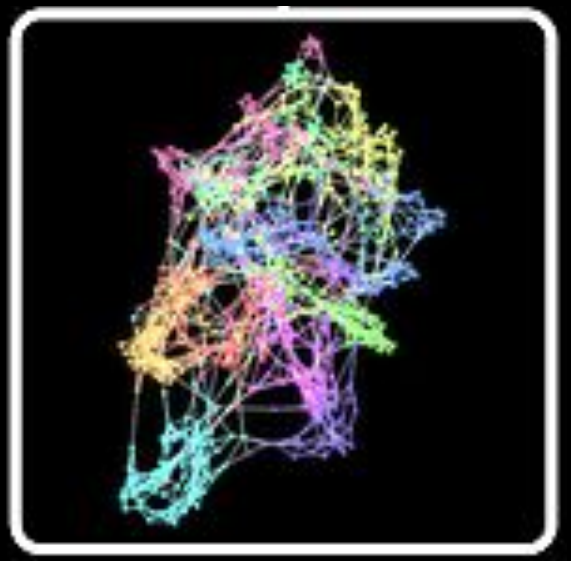
Non-linearity

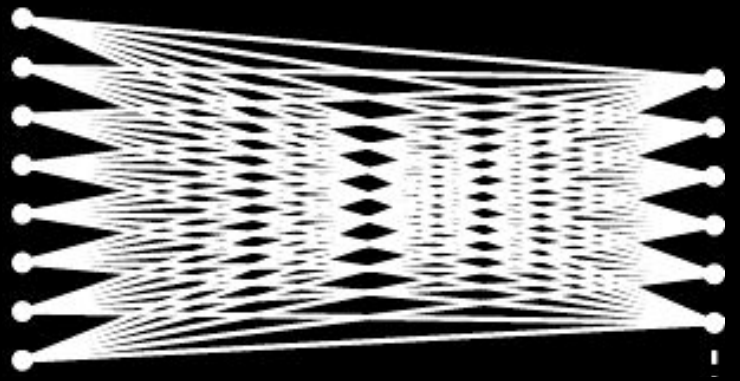


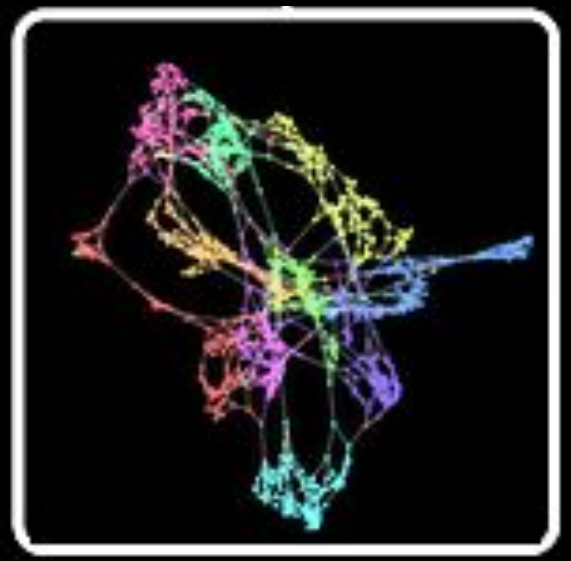
Networks as space transform

This idea extends even to very complex spaces

$$f_\phi(\mathbf{x})$$







Our goal is to find these mathematical unfolding
In order to provide **new ways of representing the data**

Networks as **representation learning**

Universal approximation theorem

An intriguing result on the expressive power of networks

We know that neural networks are powerful function approximator

Theoretical results show that they even are *universal approximators*

Universal approximation theorem

> For *any continuous function* defined on the unit hypercube, there exists a neural network that can approximate this function *up to an arbitrary level of accuracy* on the entire domain of the function

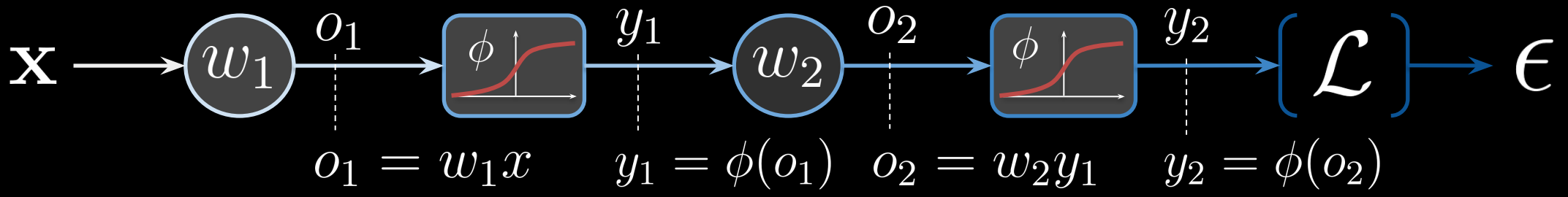
So neural networks can approximate **any function**

Although we know that such a network exists for any function

- No guarantees that we know how to learn it
- No knowledge on the required width (capacity)
- Not necessarily representable in a dataset

Backpropagation

Deriving the simplest multi-layer setting



As usual, we need the effect of weight modification on errors

So we need the derivatives $\Delta \bar{\mathbf{w}} = \left(\frac{\delta \mathcal{L}}{\delta w_1}, \frac{\delta \mathcal{L}}{\delta w_2} \right)$

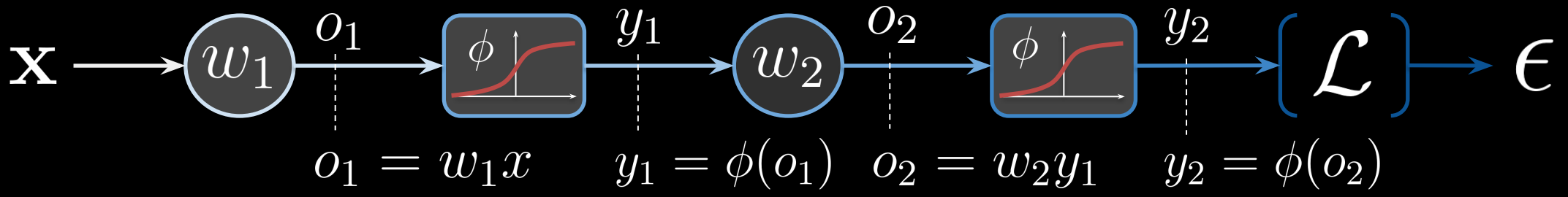
We rely on *sigmoid* activation $\phi(x) = \frac{1}{1+e^{-x}} \longrightarrow \frac{\partial \phi(\mathbf{x})}{\partial \mathbf{x}} = \phi(\mathbf{x})\left(1 - \phi(\mathbf{x})\right)$

Still we would need to derive a quite complicated equation

$$\bar{y} = y_2 = \phi(w_2 \phi(w_1 x))$$

Backpropagation

Deriving the simplest multi-layer setting



$$\frac{\delta \mathcal{L}}{\delta w_2} = \frac{\delta \mathcal{L}}{\delta y_2} \cdot \frac{\delta y_2}{\delta w_2} \quad \frac{\delta y_2}{\delta o_2} \cdot \frac{\delta o_2}{\delta w_2} \cdot y_1$$

$2\,(y - y_2)$ $y_2\,(1 - y_2)$

$$\frac{\delta \mathcal{L}}{\delta w_1} = \frac{\delta \mathcal{L}}{\delta y_2} \cdot \frac{\delta y_2}{\delta w_1} \quad \frac{\delta y_2}{\delta o_2} \cdot \frac{\delta o_2}{\delta w_1} \quad \frac{\delta o_2}{\delta y_1} \cdot \frac{\delta y_1}{\delta w_1} \quad \frac{\delta y_1}{\delta o_1} \cdot \frac{\delta o_1}{\delta w_1} \cdot x$$

$2\,(y - y_2)$ $y_2\,(1 - y_2)$ w_2 $y_1\,(1 - y_1)$

Total amount of work is a linear function of the depth of the network

Multi–Layer Perceptron (MLP)

How to code a Multi-Layer Perceptron (MLP)

1. Initialize w_i^l to small random values
2. Randomly choose a batch of inputs from the training set
    1. Present patterns $\mathbf{X} = \{\mathbf{x}_1, \cdots, \mathbf{x}_n\}$ to the network
    2. **Forward propagation**: Evaluate its output $\bar{y}$ and error ϵ
    3. Compute the δ_i^L in the *output (last) layer* $(y_i^L = \hat{y})$ $l \in [1, \cdots, L]$

$$\delta_i^L = \phi^{'}\left(o_i^L\right)\left[\delta_i^u - y_i^L\right]$$

derivative of activation — **Net input to ith unit in Lth layer**

    4. **Back-propagation** by computing deltas for preceding layers

$$\delta_i^l = \phi'\left(h_i^l\right)\sum_j w_{ij}^{l+1}\delta_j^{l+1} \quad l \in [1 \ldots L-1]$$

    5. Update weights in the corresponding layers $\Delta\bar{\mathbf{w}} = \left(\frac{\delta\mathcal{L}_\mathcal{D}}{\delta w_1}, \ldots, \frac{\delta\mathcal{L}_\mathcal{D}}{\delta w_n}\right)$

Optimization

Gradient descent can be slow and may get stuck in local minima.
Several techniques are used to improve convergence and escape local minima

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \nabla \mathcal{L}_i(\mathbf{w}_t)$$

Mini-batch GD

Use a **random subset (mini-batch)** of the dataset.
Trade-off for balancing speed and variance

Momentum

Add a momentum term to the weight update
Help escape local minima and speed up convergence.

$$\nabla \mathcal{L}_i(\mathbf{w}_t) + m * \nabla \mathcal{L}_i(\mathbf{w}_{t-1})$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{\eta}{\sqrt{\sum_{k=1}^{\tau} \nabla \mathcal{L}_i(\mathbf{w}_{t-k})}} \nabla \mathcal{L}_i(\mathbf{w}_t)$$

Adaptive Gradient (AdaGrad)

Adapt the learning rate for each weight individually
Based on the historical gradients

Adaptive Moment Estimation (ADAM)

Combines benefits of momentum and adaptive learning rates.
Maintains separate first and second moment estimates for each weight.

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \mathbf{m}_t \left(\frac{\alpha}{\sqrt{\mathbf{v}_t} + \epsilon} \right)$$

Overfitting

Problem

The network learns the training data too well

- Starts to *memorize* all of the training examples
- Does not **generalize** to unseen examples

Leads to poor performance on unseen example (our major goal)

Potential solutions

Several techniques to prevent *overfitting* and improve **generalization**:

- **Early stopping**: Stop training when validation loss starts to increase
- **Regularization**: Penalize large weights or complex models
  - Add an L1 or l2 penalty to the loss to encourage sparsity
- **Correct initialization**: Provide a more efficient starting point for optimization
- **Dropout**: Randomly dropout neurons to prevent *co-adaptation*
- **Data augmentation**: Create new training samples by applying transforms to the data

Cross-validation used to estimate generalization and tune hyperparameters

Important to balance between preventing overfitting and allowing the model to fit

Regularization issues

Regularization used to prevent overfitting by penalizing complex models

Incentive to find a simpler solution (usually through ***additive penalty***)

Weights penalty Encourages sparsity in the weight vector

L1 (lasso)

Add a penalty on the **weights magnitude**

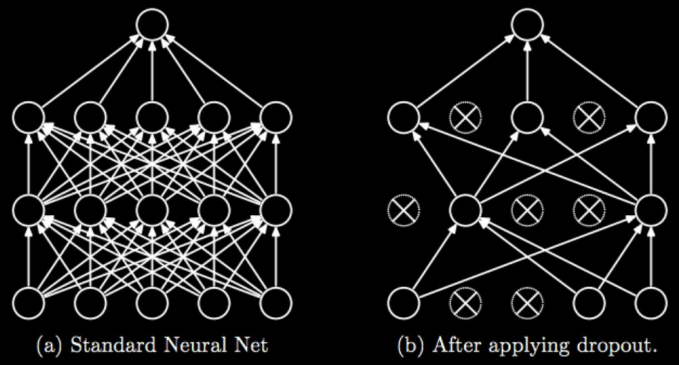
$$\mathcal{L}_{L_1}(\bar{\mathbf{y}}, \theta) = \mathcal{L}(\bar{\mathbf{y}}, \theta) + \lambda \sum_{i=1}^{n} |w_i| \quad \lambda \sum_{l=1}^{L} \sum_{i=1}^{n^{(l)}} \sum_{j=1}^{n^{(l-1)}} |w_{ij}^{(l)}|$$

L2 (ridge)

Extension to squared penalty

$$\mathcal{L}_{L_2}(\bar{\mathbf{y}}, \theta) = \mathcal{L}(\bar{\mathbf{y}}, \theta) + \frac{\lambda}{2} \sum_{i=1}^{n} w_i^2$$

Elastic net Combines L1 and L2 regularization techniques with two hyperparameters



(a) Standard Neural Net (b) After applying dropout.

Dropout

While training, randomly set neurons to zero with given probability

Prevents the network from relying on any single neuron

Encourages redundancy, leading to a more robust model.

Already "deep learning era", more to come

Initialization

Important to ensure convergence and prevent vanishing or exploding gradients.

Worst case | *Zero Initialization*: Initialize all weights and biases to zero.
Problem: Symmetric neurons cause the network to fail to learn

Typical case | *Random Initialization*: Initialize weights randomly from Gaussian distribution
Problem: Large initial values = exploding gradients, small values = vanishing gradients.

Better strategies

Xavier $\mathbf{w} \sim \mathcal{U}(-\frac{1}{\sqrt{n_l}}, \frac{1}{\sqrt{n_l}})$

$\mathbf{w} \sim \mathcal{N}(0, \frac{2}{n_l})$ ***He***

Xavier and **He** Initialization:

- Use Gaussian distribution with variance depending on number of neurons.
- Problem: Assumes that the distribution of input features is symmetric and Gaussian.

Orthogonal Initialization:
Initialize weights to orthogonal matrix keeps gradient norm during backpropagation.
Very useful for *convolutional kernels* (see next course)

Kaiming Initialization:
Variant of He Initialization for rectified linear units (ReLU) activation function.

Hyperparameters

Problem

Large variety of hyperparameters that can be tuned to improve performance.

Hyperparameter tuning is time-consuming and computationally expensive.

Common hyperparameters

- **Learning rate**. Step size of gradient update during training.
- **Number of layers**. Depth of the network
- **Number of neurons**. Width of the network
- **Activation function**. Non-linearity of the network
- **Batch size**. Number of samples used in each iteration
- **Dropout rate**. Probability of dropping out neurons
- **Regularization**. Penalty for large weights

Best practices

- Using a validation set to evaluate model during training.
- Coarse-to-fine and *random* search strategy (Bayesian optimization)
- Keeping track of all experiments and results to identify trends and patterns.
- Using visualization tools to understand hyperparameters effects

Alternative learning rules

So far we have only used the **error-correction** learning rule

We use the error signal as a learning signal (learning only if errors are made)

Other rules can be implemented

Hebbian

- Synapse strength increases if neurons on both sides are activated
- Learning is done locally with synchronicity (only for two connected neurons)
- This amounts to learn the maximal variance

Boltzmann

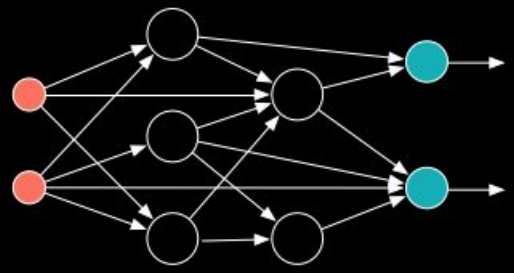
- Define symmetric networks where each neuron is a stochastic unit
- Output based on Boltzmann distribution of statistical mechanics
- Two modes (clamped or free-running) for each neuron

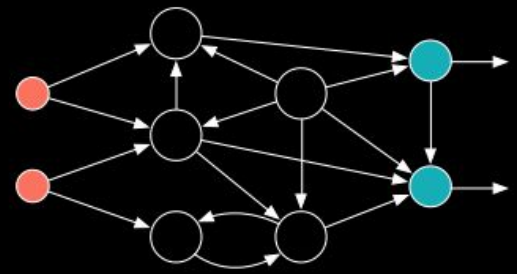
Competitive

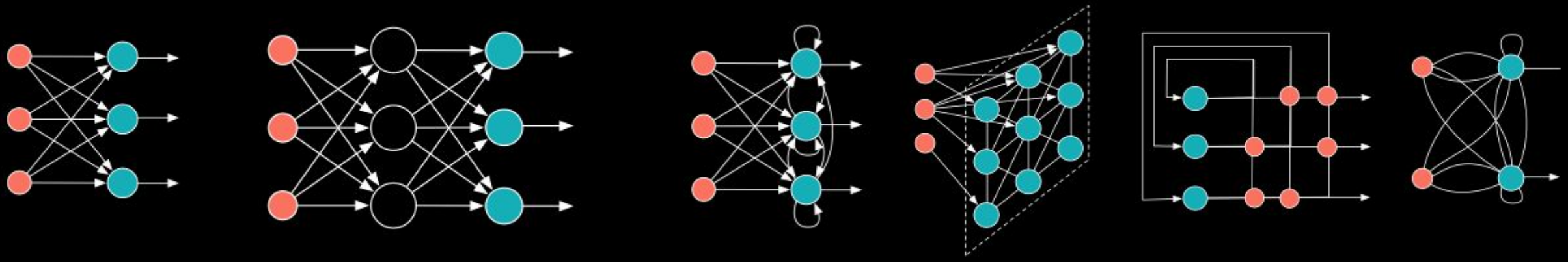
- Neurons compete with only one output active at any given time
- Complete mesh of connection with inhibitory weights and self-feedback
- Creates an implicit tradeoff between plasticity and stability

Architecture zoo

Large variety in network architectures (**upcoming courses**)

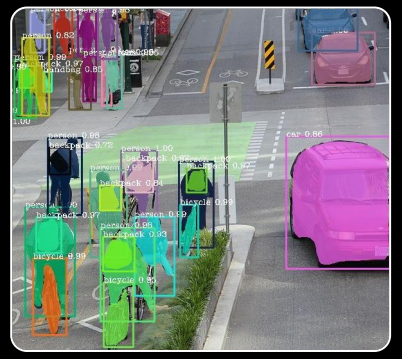






Applications

Neural networks have a wide range of applications across various domains

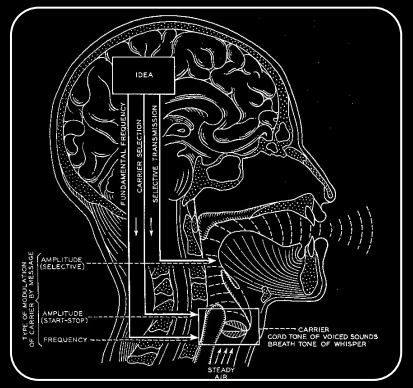


Computer vision

Object detection, image classification, and segmentation.
Examples: AlexNet, VGG, ResNet.

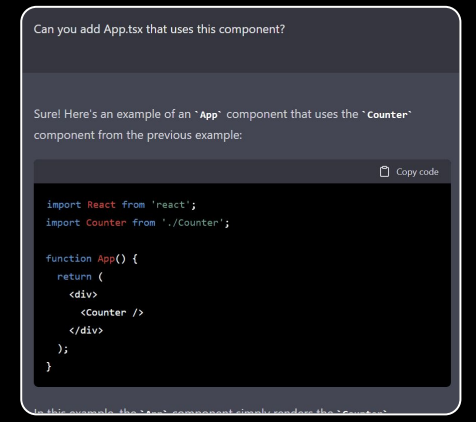
Natural language processing

Text classification, translation, and sentiment analysis.
Examples: LSTM, Transformer, BERT.



Speech synthesis

Automatic Recognition (ASR), Text-to-Speech (TTS).
Examples: DeepSpeech, Tacotron, WaveNet.



Reinforcement learning

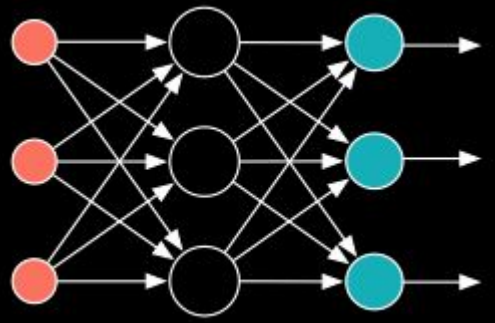
Control systems, game playing, and robotics.
Examples: Deep-Q Network (DQN), A3C, PPO

Generative models

Generate new data samples or perform data imputation.

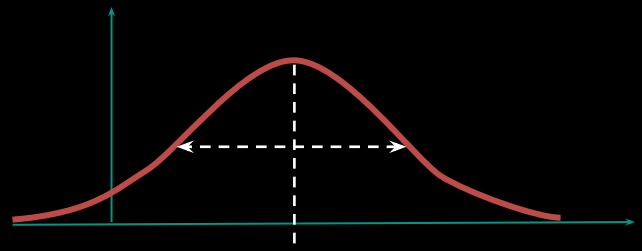
$$\mu_{i}$$
$$\sigma_{i}$$

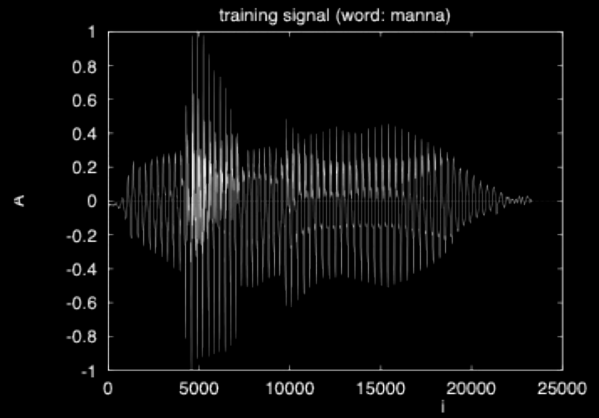
Radial Basis Function (RBF)

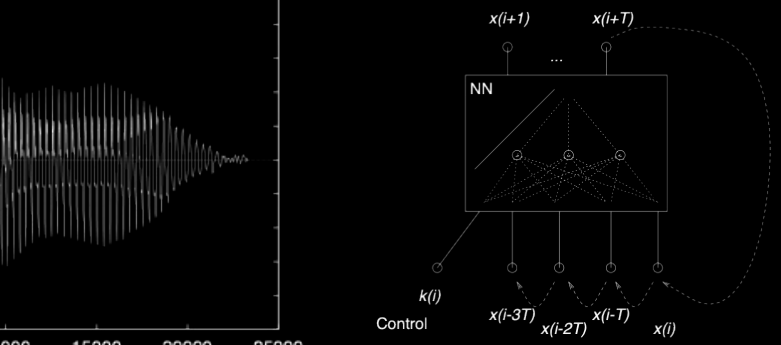


Radial Basis Function (RBF) Network

- Special case of feed-forward network
- Follows the multilayer perceptron structure
- Usually 2 layers architecture
- Activation function is a gaussian kernel
- Both and must be learned
- 2-step hybrid learning strategy
  1. Unsupervised clustering for and
  2. Supervised Least Mean Squares (LMS)



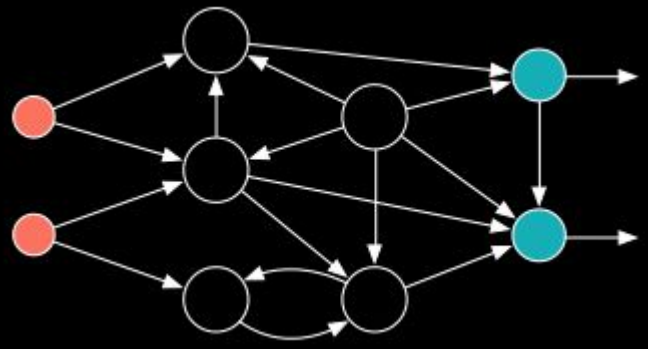


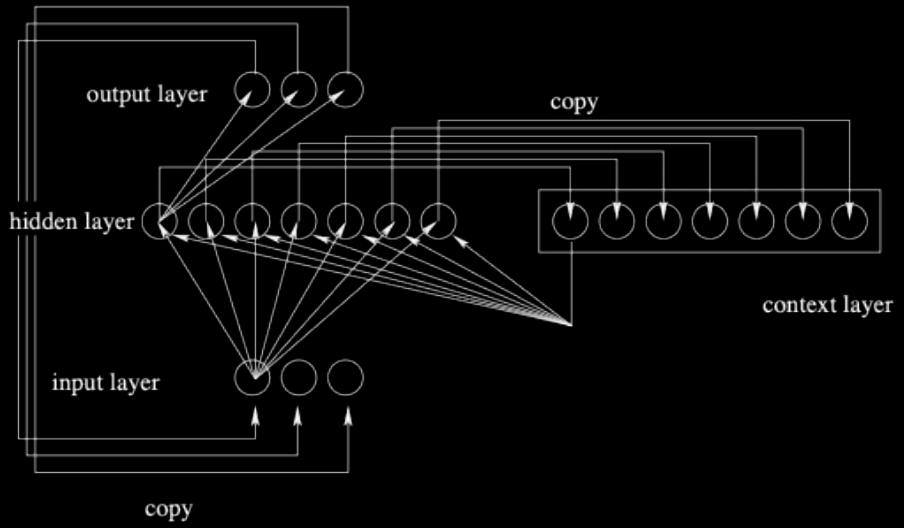


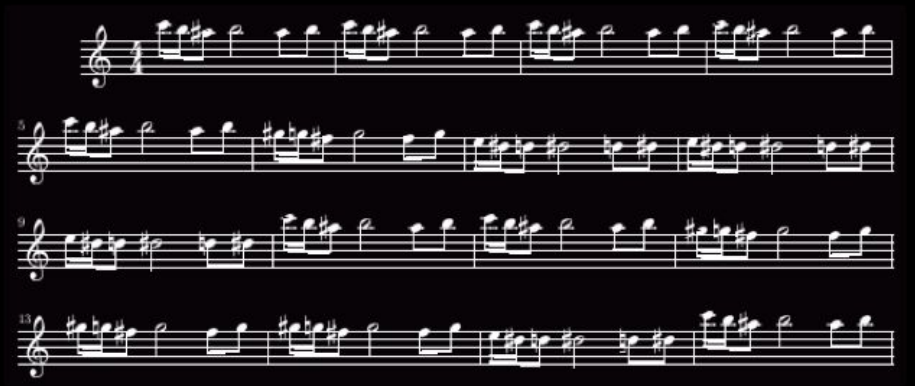
Recurrent / feedback networks

Recurrent/Feedback Network

- Connections can go backwards
- Self-connections are also allowed
- Allows retro-action but can also model
  - Time dependency notions
  - Different network states



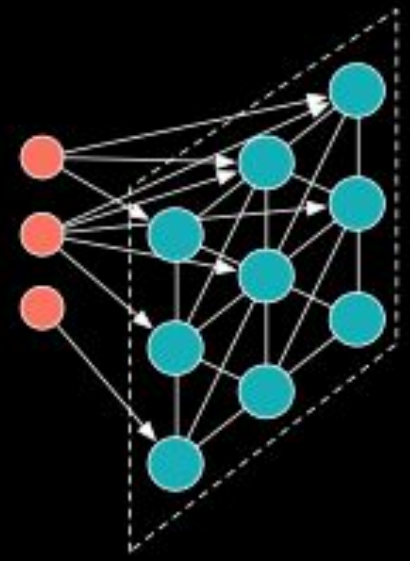


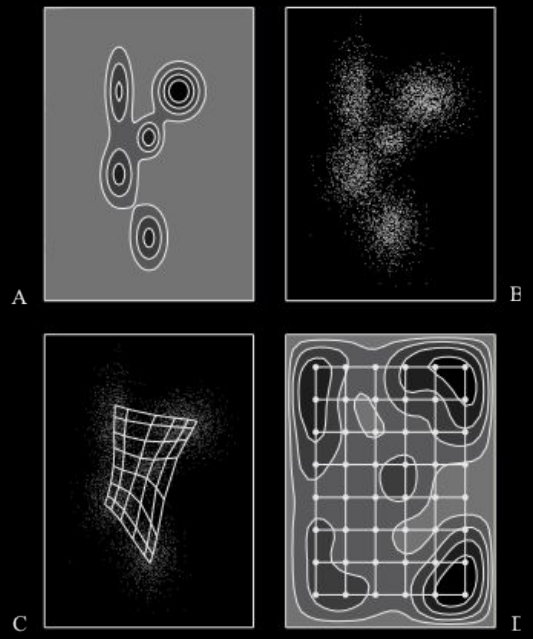


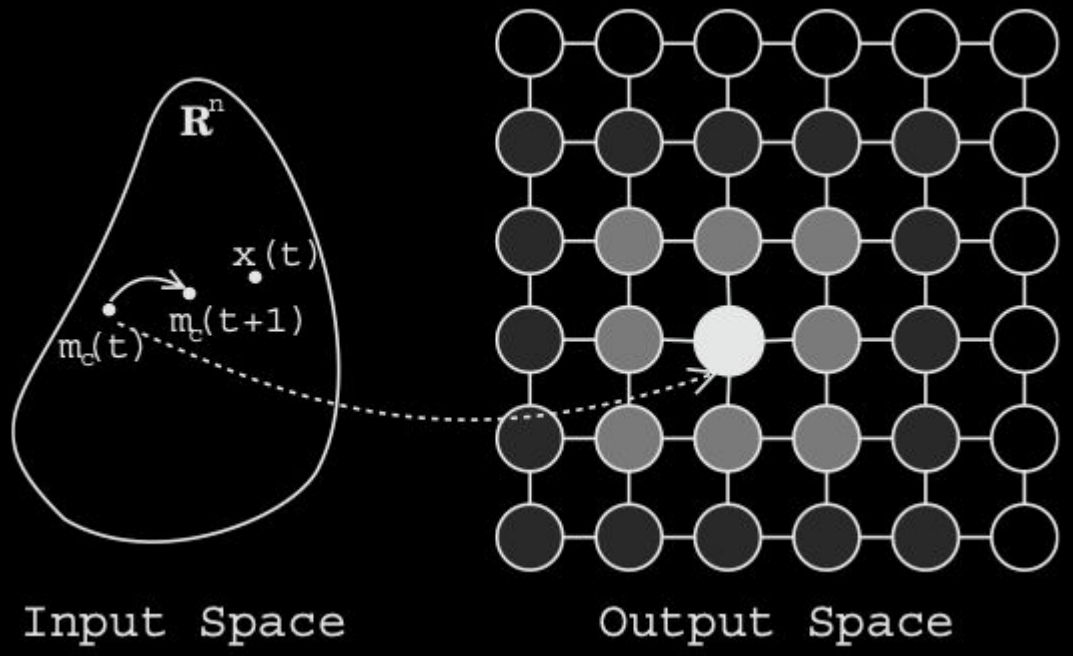
Self-Organizing Maps (SOM)

Kohonen's Self-Organizing Maps (SOM)

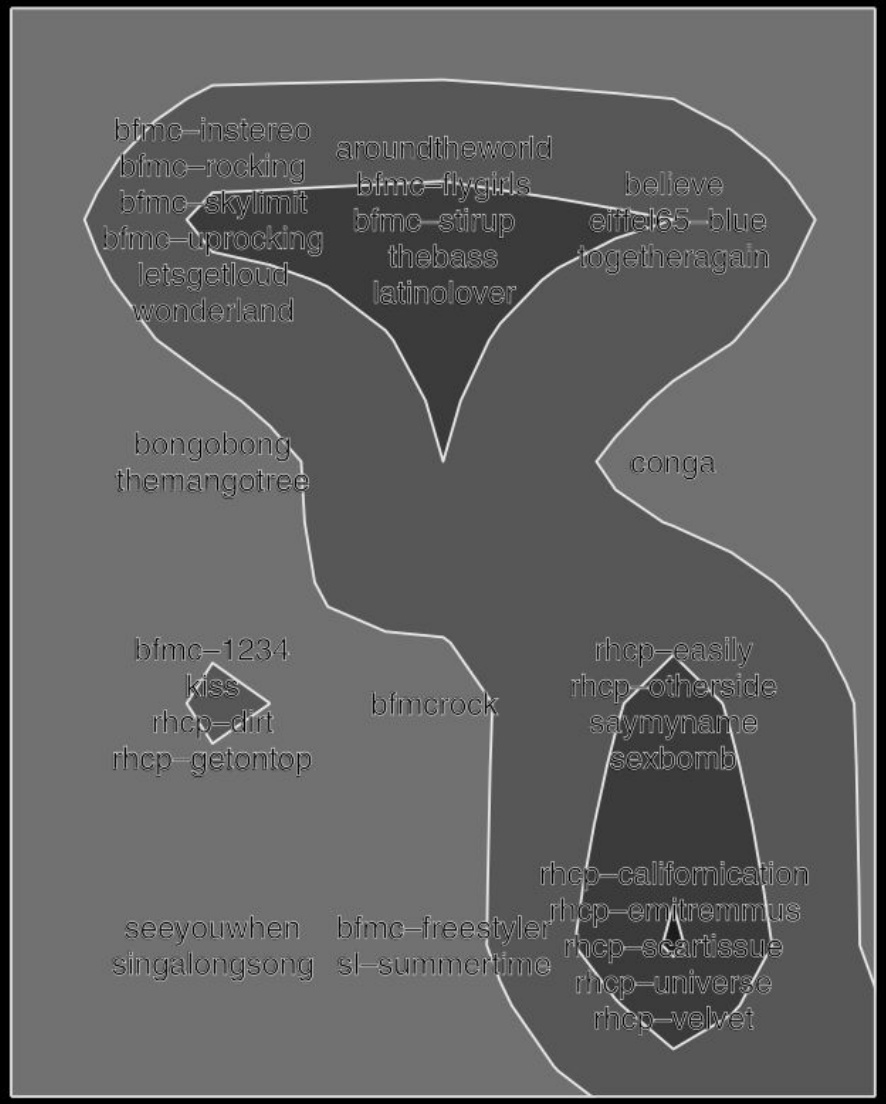
- Each neuron connected to all neurons
- But distance between neurons is kept
- And this distance influences the network
- Models relations of spatial neighborhood
- Local neurons influence more than global
- Embeds a property of topology preservation

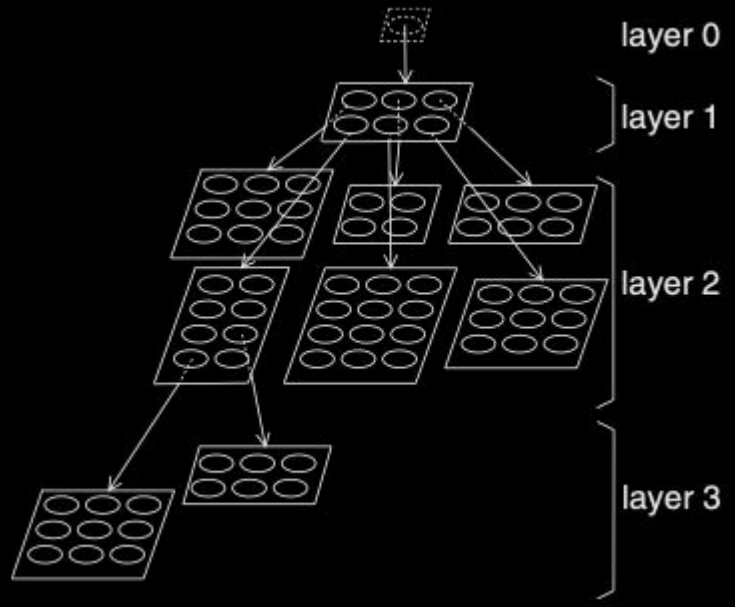


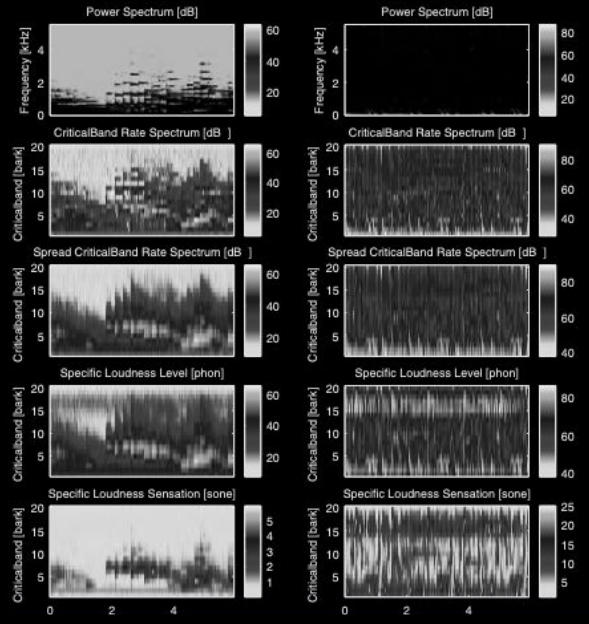




SOM Examples







Adaptive Resonance Theory

Adaptive Resonance Theory (ART) models

- Form of competitive network
- Not all outputs neuron are used (only if necessary)
- Difference between commited and uncommited
- Only update a category if prototypes simlar
- The extent of similarity is controled by vigilance P