Most common kinds of neural networks

Culloch & Pitts neuron

- 1940s
- Binary-state elements with threshold s

$$y = \Theta(\sum_{i=1}^{k} w_i x_i - s)$$

$$\Theta(x) = \begin{cases} 1 & \text{if } x \in \mathcal{R}_0^+ \\ 0 & \text{if } x \in \mathcal{R}^- \end{cases}$$

- It can express any logical function
- Not yet a proper artificial neural network - does not include adaptive dynamics.

Hebbian rule

- Any two neurons that are repeatedly active at the same time will tend to become 'associated'.
- Change of weight of the connection between two neurons is proportional to the correlation of their activities.

$$\Delta w_i = \epsilon y x_i, i = 1, ..., k$$

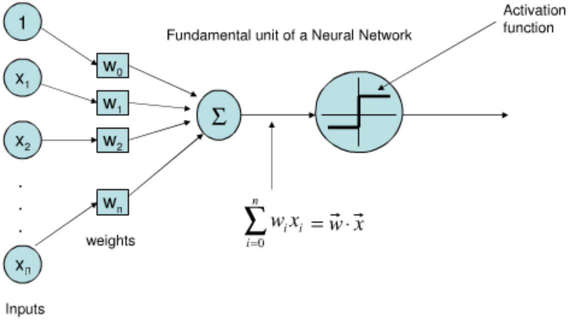
  - input signals $x = (x_1, ..., x_k)$,
  - output signal y,
  - learning rate ε, possibly dependent on x (then denoted ε_x)

Perceptron

- Rosenblatt - 1958

$$y_r = \Theta(\sum_{i=1}^{k} w_i x_i)$$

- Threshold from Culloch & Pitts neuron can be expressed with $-w_1$ for $x_0 = 1$



Perceptron learning

- Learning is performed in epochs.
- In each epoch:
  - A vector (learning sample) x_r , $r \in \{1, ..., n\}$ is introduced to the perceptron and it reacts with output y_r.
  - Weigths $w = (w_1, ..., w_k)$ are adjusted unless y_r fulfills:

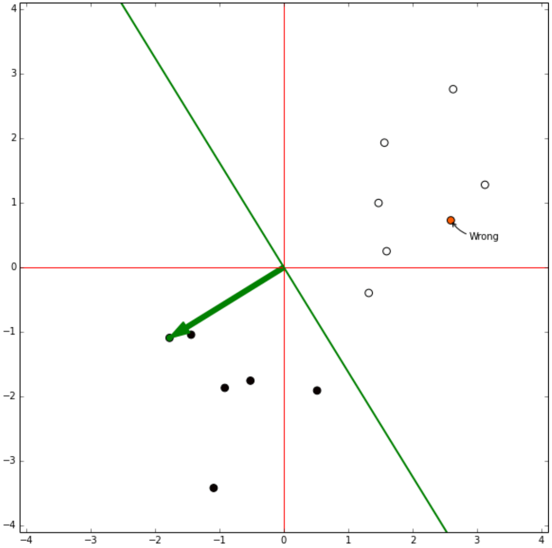
$$y_r = \begin{cases} 1 & \text{if sample } r \text{ is class of } C_r \\ 0 & \text{if sample } r \text{ is not class of } C_r \end{cases}$$

  - weight w_i is changed by $\Delta w_{(i,r)} = \varepsilon_x(\delta(r,s) - y_r)x_i$

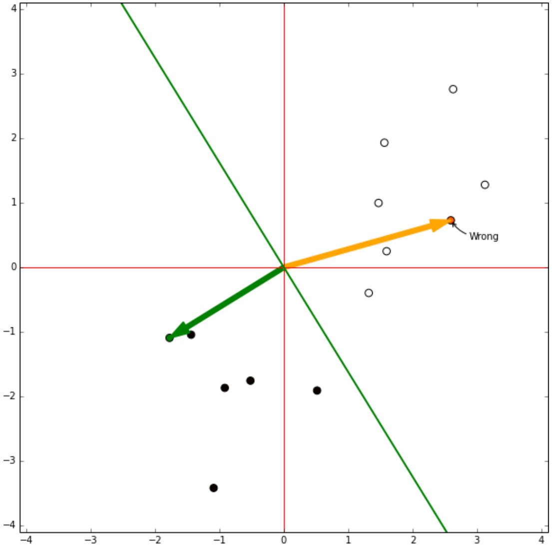
$$\delta(r,s) = \begin{cases} 1 & |r, s = 1, ..., n, r = s \\ 0 & |r, s = 1, ..., n, r \neq s. \end{cases}$$

- The solution exists if the classes are *linearly separable*.

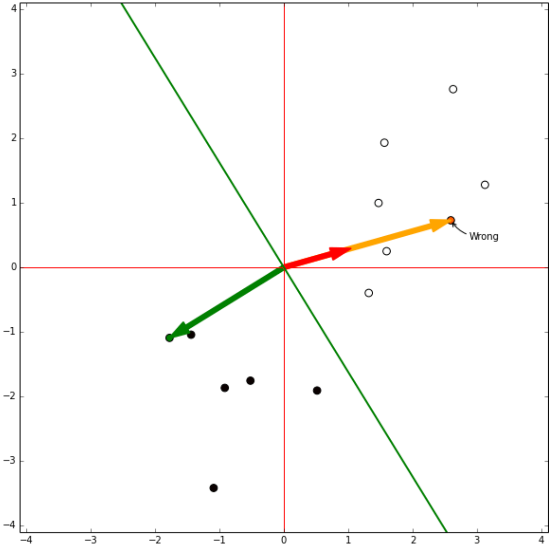
Perceptron learning animation



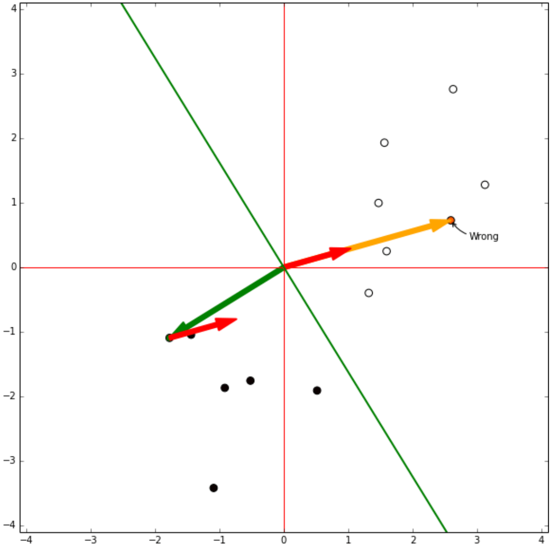
Perceptron learning animation



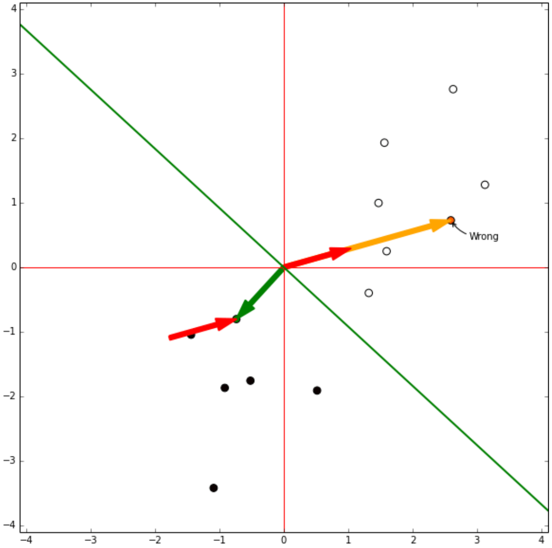
Perceptron learning animation



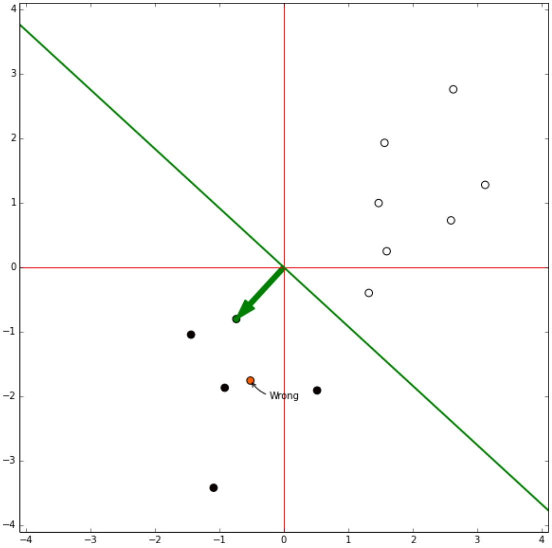
Perceptron learning animation



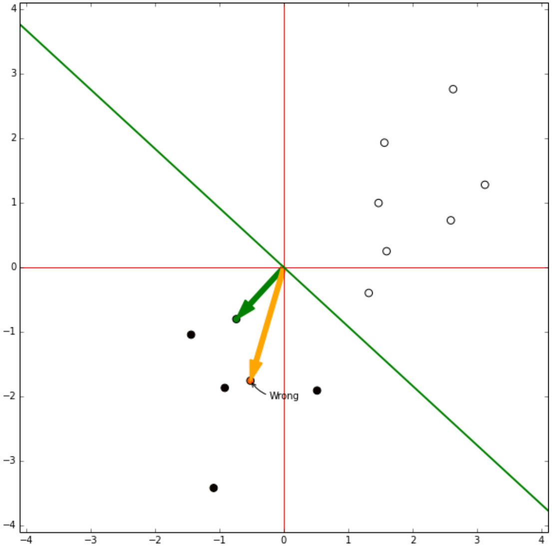
Perceptron learning animation



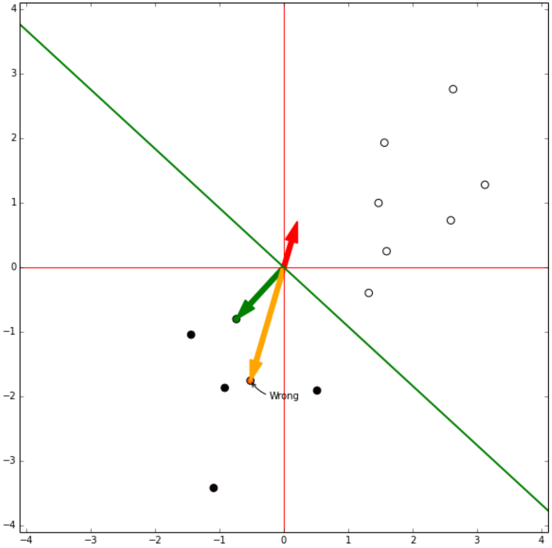
Perceptron learning animation



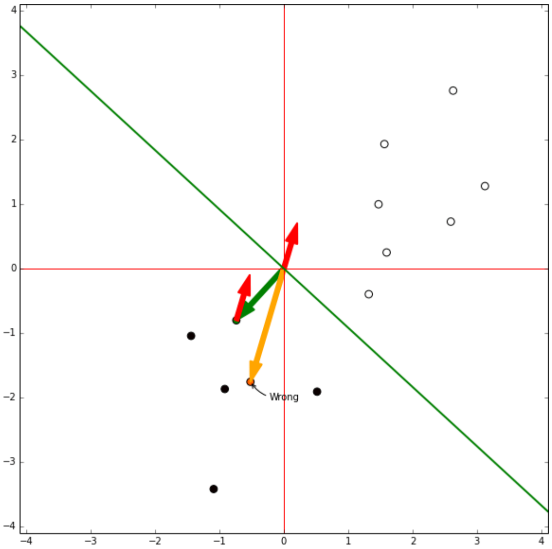
Perceptron learning animation



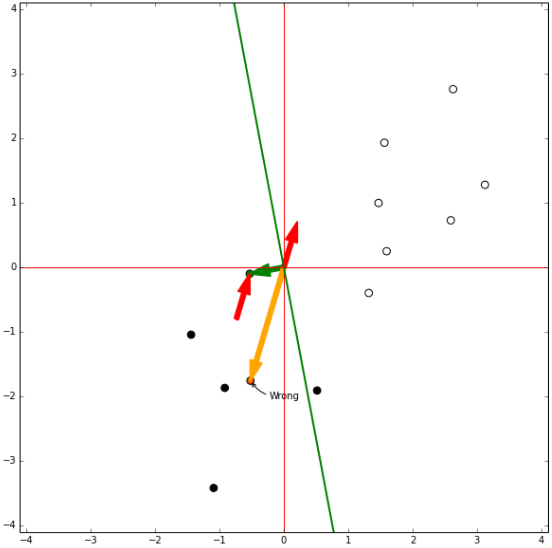
Perceptron learning animation



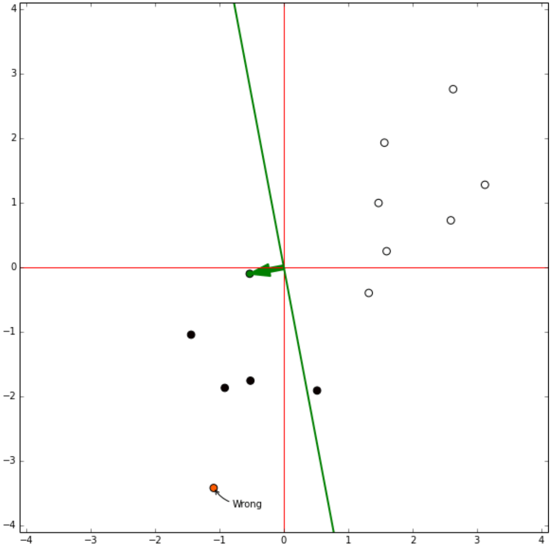
Perceptron learning animation



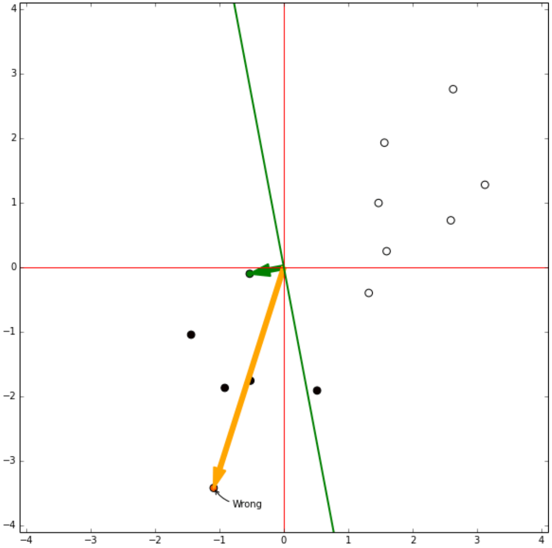
Perceptron learning animation



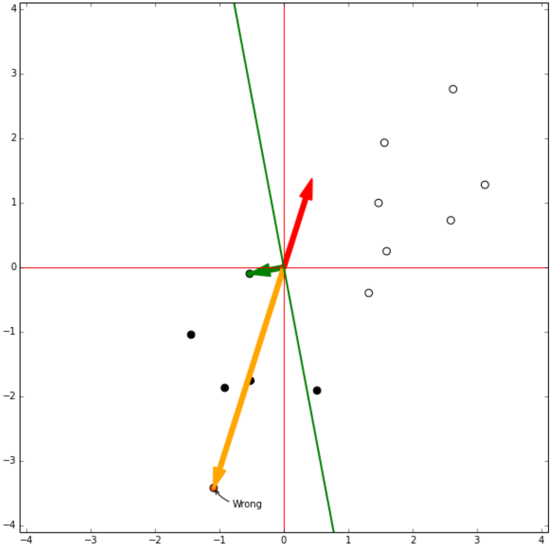
Perceptron learning animation



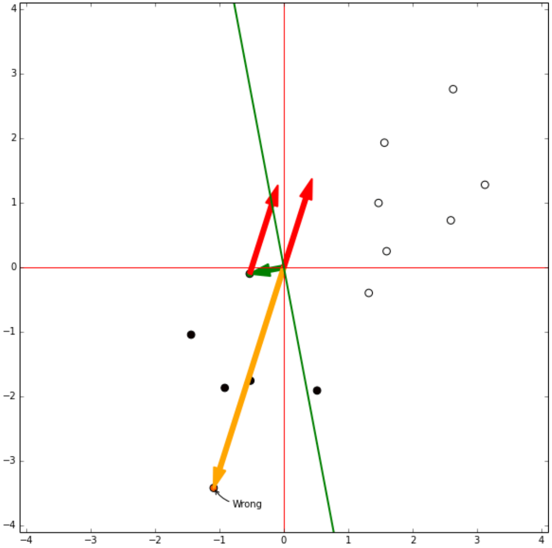
Perceptron learning animation



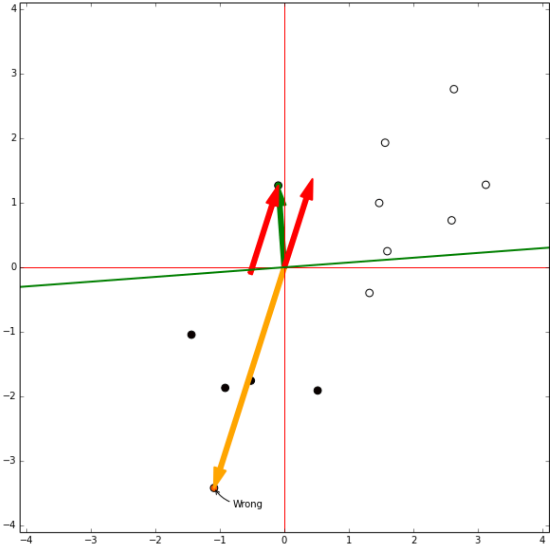
Perceptron learning animation



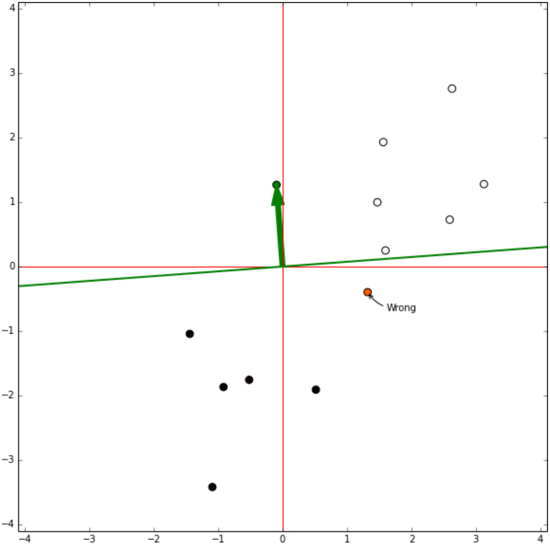
Perceptron learning animation



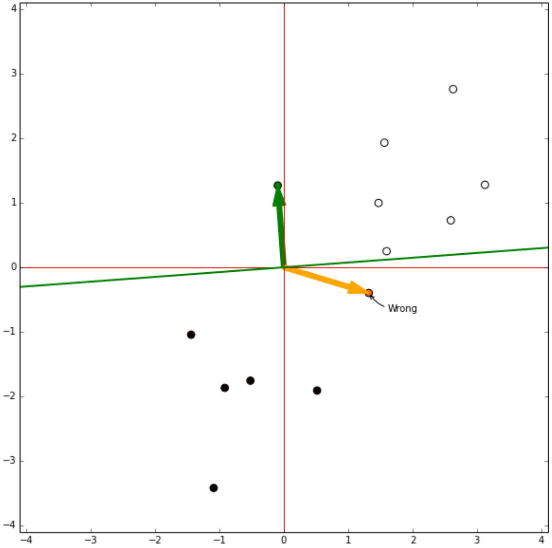
Perceptron learning animation



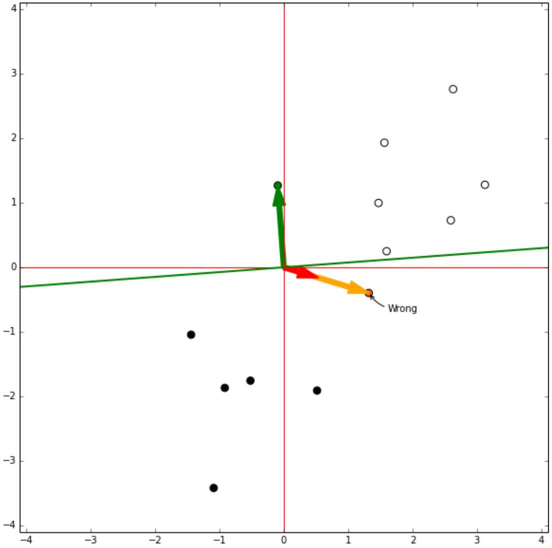
Perceptron learning animation



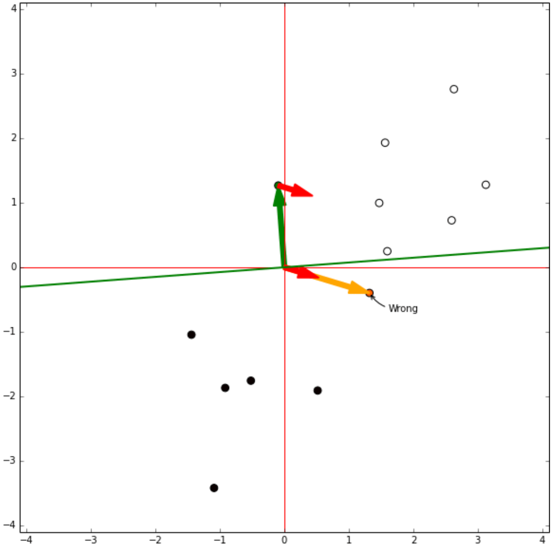
Perceptron learning animation



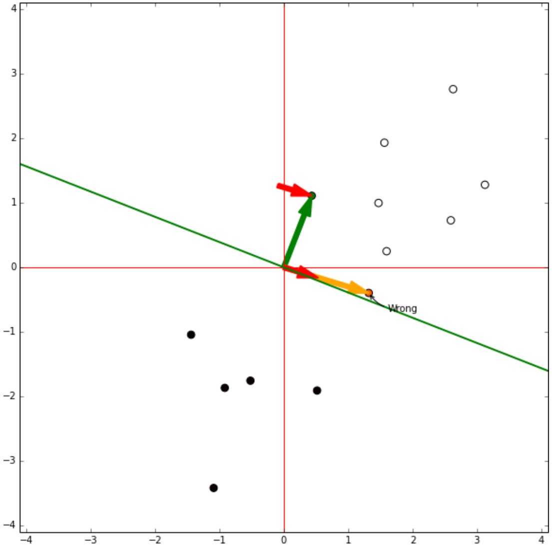
Perceptron learning animation



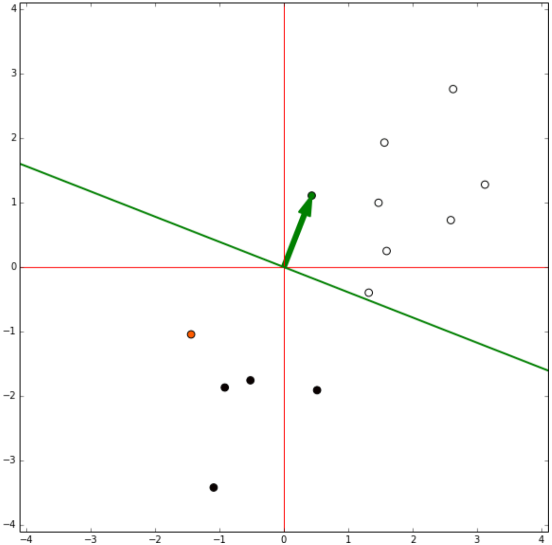
Perceptron learning animation



Perceptron learning animation



Perceptron learning animation



Perceptron learning convergence

Perceptron Convergence Theorem I.

Assume set of learning samples $X \subset \mathcal{R}^k$ for which there exists system of weights $(w_i^*)_{i=1,\dots k}$ leading to their correct classification into two linearly separable classes. Let X have the following properties:

1. $(\exists M \in \mathcal{R}^+)(\forall x \in X) \quad 0 < \sum_{i=1}^{k} x_i^2 < M$
2. $(\exists \delta \in \mathcal{R}^+)(\forall x \in X)(\forall r \in \{1, \dots, n\}) x \in C_r \Rightarrow \sum_{i=1}^{k} w_i^* x_i > \delta \quad \& \quad x \notin C_r \Rightarrow \sum_{i=1}^{k} w *_i x_i < -\delta$

Perceptron learning convergence

Perceptron Convergence Theorem II.

Then the learning algorithm for which ε_x is given by the formula

$$\varepsilon_x = \frac{1}{\sqrt{\sum_{i=1}^{k} x_i^2}}$$

finds the system of weights w_i^* for any initial setting of weights w_i and any finite set of learning samples X in a finite number of iterations.

Associative memory - motivation

- Aristotle observed that human memory connects items that are:
  - Similar
  - Contrary
  - Occur in close proximity (spatial)
  - Occur in close succession (temporal)
- AM idea comes from the Hebbian rule
  - *Cells that fire together wire together.*

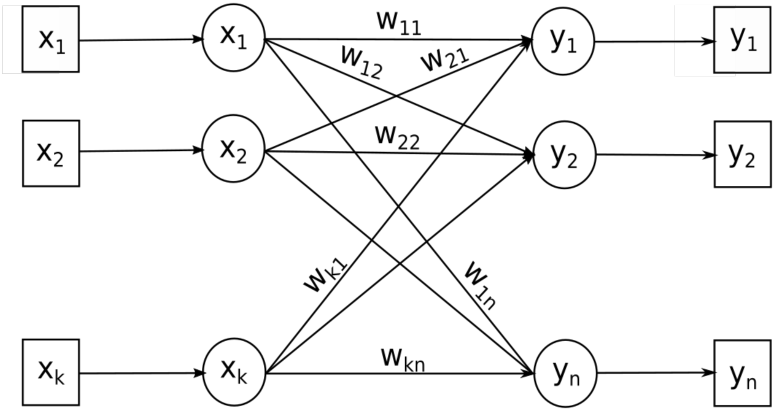
Associative memory

- Layer of units defined by:

$$y = \Theta(\sum_{i=1}^{k} w_i x_i - s)$$

- Information that should be stored is entered through pairs of binary vectors (x, y)
- $x = (x_1, ..., x_k)$ - input pattern, $y = (y_1, ..., y_n)$ - output pattern
- To obtain a satisfactory behaviour of the network, we require $k >> n$.

Associative memory



Associative memory - training

- Set all weights w_i to 0
- For each pair $(x^{(j)}, y^{(j)})$ from a training set of p training samples:
  - change $w_{i,r}$ to 1 if $x_i = y_r = 1$
- After p pairs were introduced:

$$(\forall i \in \{1, ..., k\})(\forall r \in \{1, ..., n\}) w_{i,r} = \max_{j=1,\dots,p} x_i^{(j)} y_r^{(j)}$$

Associative memory - choosing a threshold

- The threshold s is usually chosen $s = l - \frac{1}{2}$, where l is the number of "1" in input patterns.
- It can happen that the output $y_q, q = \{1, ..., n\}$ is 1 even if $y_q^{(i)}$ was 0 0 for $x^{(i)}$ at the input.
- With $s = l - \frac{1}{2}$, the network is intolerant to errors
- With lowering s, we achieve better tolerance, but a wrong $y_q = 1$ occurs more frequently.

Linear Associative memory

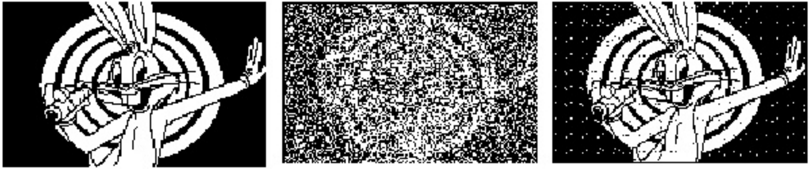
- Absence of non-linear activation function
- Units are simplified:

$$y_r = \sum_{i=1}^{k} w_{i,r} x_i$$

$$y = Wx$$

- Superposition principle
- $x^{(j)} \in \mathcal{R}, y^{(j)} \in \mathcal{R}^n$
- Real-valued inputs might be very useful (e.g. colours of a picture)

Auto Associative memory



Original　Degraded　Reconstruction

Linear Associative memory - learning weights

- Optimizing weights W^* to minimize loss function γ

$$\sum_{j=1}^{p} \gamma(y^{(j)}, W^* x^{(j)}) = \min_{W \in \mathcal{R}^{k,n}} \sum_{j=1}^{p} \gamma(y^{(j)}, W x^{(j)})$$

- for the common loss function least squares this leads to quadratic optimization

$$E(W^*) = \min_{W \in \mathcal{R}^{k,n}} E(W), \text{where}$$

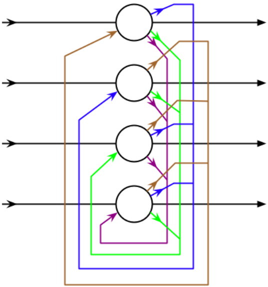
$$E(W) = \sum_{j=1}^{p} \sum_{r=1}^{n} \left(y_r^{(j)} - \sum_{i=1}^{k} w_{i,r} x_i^j\right)^2 | W \in \mathcal{R}^{k,n}$$

Hopfield network

- The output signal of each neuron is sent to the input of other neurons.

$$z_i(t) = 2\Theta\left(\sum_{j=1}^{k} w_{(j,i)} z_j(t-1)\right) - 1, w_{i,i} = 1$$

- At each time $t \in \mathcal{N}$, exactly one neuron $i \in \{1, ..., k\}$ is changing its activity value (asynchronous behavior).



Hopfield network - steady state

- Hopfield network can be studied in terms of interacting particles known from statistical physics.
- Energy function:

$$H(z) = -\frac{1}{2} \sum_{j,i=1}^{k} w_{(i,j)} z_j z_i | z \in \{-1, 1\}^k$$

- From the function $H(z)$ we can see if the network is in *steady state* (local minimum)
- Every Hopfield network will get into steady state after few iterations.

Hopfield network - weights settings

- Common setting for independent training samples:

$$w_{(i,j)} = \frac{1}{k} \sum_{\nu=1}^{p} x_i^{(\nu)} y_j^{(\nu)}$$

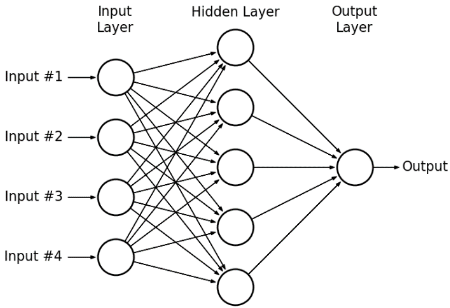
- Works well for $p << k$.

Hopfield network - summary

- Important for theoretical study of recurrent Neural nets properties
- Does not work well if input vectors are correlated
- Vector $z(0)$ is not invariant to simple transformations (shift, rotation, size change)

Multilayer perceptron

- Topology organized in layers
- Neurons within a layer are not connected
- Signals are transferred only from input neurons to output neurons (feed-forward neural network)



Multilayer perceptron - backpropagation algorithm I

- We are trying to find a system of weights $w^* \in \mathcal{R}^{|\mathcal{I}\times\mathcal{H}\cup\mathcal{H}\times\mathcal{O}|}$ minimizing

$$E(w) = \sum_{j=1}^{p} \gamma(y^{(j)}, F_w(x^{(j)}))$$

- The most commonly used lost function is the *sum of squares (SSE)*, typically multiplied by $\frac{1}{2}$:

$$E(w) = \frac{1}{2}\sum_{j=1}^{p} ||y^{(j)} - F_w(x^{(j)})||^2 = \frac{1}{2}\sum_{j=1}^{p}\sum_{i=1}^{|\mathcal{O}|}(y_i^{(j)} - (F_w(x^{(j)}))_i)^2$$

Multilayer perceptron - backpropagation algorithm II.

- The minimum of the function E is found iteratively:
  $w_{(u,v)} = w_{(u,v)} - \alpha \Delta w_{(u,v)}$, where
  $$\Delta w_{(u,v)} = \frac{\partial E}{\partial w_{(u,v)}}(w)$$
- The direction of weight change is opposite to the direction of the gradient of E (the steepest descent of E)

Multilayer perceptron - backpropagation algorithm III.

- Assume the SSE loss function and any differentiable activation function f (logistic, arctan).
- For links $(u, v) \in \mathcal{H} \times \mathcal{O}$:

$$\frac{\partial E}{\partial w_{(u,v)}}(w) = -\sum_{j=1}^{p}(y_v^{(j)} - z_v^{(j)})f'(\sum_{h\in\mathcal{H}} w_{(h,v)}z_h^{(j)} + \Theta_v)z_u^{(j)}$$

- For links $(u, v) \in \mathcal{I} \times \mathcal{H}$:

$$\frac{\partial E}{\partial w_{(u,v)}} = -\sum_{j=1}^{p}\sum_{o\in\mathcal{O}}(y_o^{(j)} - z_o^{(j)})f'(\sum_{h\in\mathcal{H}} w_{(h,o)}z_h^{(j)} + \Theta_o)w_{(v,o)}\frac{\partial z_v^{(j)}}{\partial w_{(u,v)}}(w)$$

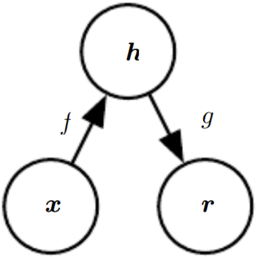
$$= -\sum_{j=1}^{p}\sum_{o\in\mathcal{O}}(y_o^{(j)} - z_o^{(j)})f'(\sum_{h\in\mathcal{H}} w_{(h,o)}z_h^{(j)} + \Theta_o)f'(\sum_{i\in\mathcal{I}} w_{(i,v)}x_i^{(j)} + \Theta_v)w_{(v,o)}x_u^{(j)}$$

Multilayer perceptron - backpropagation algorithm IV.

- This algorithm often leads to a local minimum instead of a global minimum
- The function E has $|\mathcal{H}|(|\mathcal{I}| + |\mathcal{O}|)$ variables and it is very complicated with many local minima.
- To overcome this issue, there are many approaches that help us to get out of local minimum by changing α (cyclic learning rate, learning rate annealing, ...)

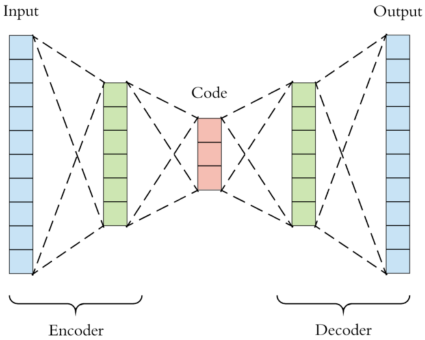
Autoencoder I.

- Autoencoder is is trained to attempt to copy its input to its output.
- Hidden layer h that describes a *code* used to represent the input.
- Consists of two parts:
  - encoder $h = f(x)$
  - decoder $r = g(h)$
- The net aims to learn $g(f(x)) = x$ as precisely as possible.



Autoencoder II.

- Autoencoder may be thought of as a special case of feedforward network
- It is typically trained using minibatch back-propagation.
- Typically used in unsupervised way.



Undercomplete autoencoder

- We hope that training the autoencoder will result in h taking on useful properties.
- $\Rightarrow$ Constrain h to have a smaller dimension than input x.
- With nonlinear encoder and decoder functions it can learn a more powerful nonlinear generalization of PCA.

- If the encoder and decoder are allowed too much capacity, the autoencoder can learn to perform the copying task without extracting useful information
- Similar situation can happen with *overcomplete autoencoders* in which the hidden code has dimension greater than the input.
- Solution is to use **regularization**

PCA vs autoencoder

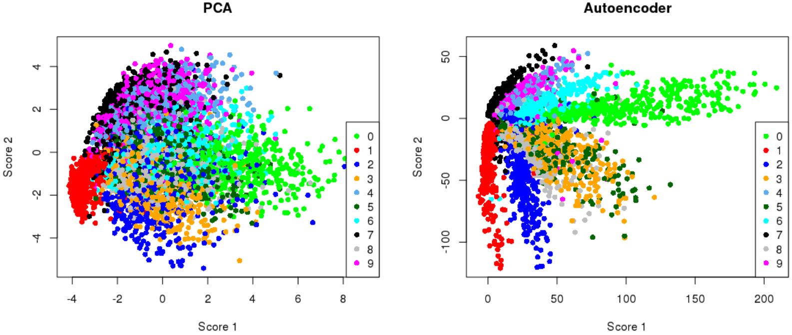


Figure: Dimensionality reduction of the MNIST dataset.

Hinton, Geoffrey E., and Ruslan R. Salakhutdinov. "Reducing the dimensionality of data with neural networks." science 313.5786 (2006): 504-507.

Autoencoder regularization

- Use a loss function that encourages the model to have other properties besides the ability to copy its input to its output.
- Regularization techniques:
  - sparsity of the representation,
  - small derivatives of the representation,
  - robustness to noise or to missing inputs.
- A regularized autoencoder can be nonlinear and overcomplete but still learn something useful about the data distribution.

Sparse autoencoder

- An autoencoder whose training criterion involves a sparsity penalty $\Omega(h)$ on the code layer h, in addition to the reconstruction error:

$$L\big(x, g(f(x))\big) + \Omega(h),$$

  where $g(h)$ is the decoder output and $h = f(x)$ is the encoder output.
- For example:

$$\Omega(h) = \lambda \sum_i |h_i|,$$

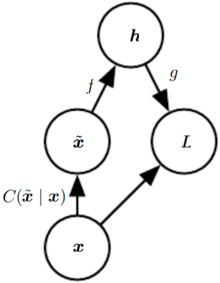
  where λ is a hyperparameter.

Denoising autoencoder I.

- Rather than adding a penalty Ω to the cost function, change the reconstruction error term of the cost function.
- A denoising autoencoder (DAE) minimizes

$$L(x, g(f(\tilde{x}))),$$

where $\tilde{x}$ is a copy of x that has been corrupted by some form of noise.
- Denoising training forces f and g to implicitly learn the structure of $p_{\text{data}}(x)$



Denoising autoencoder II.

- A corruption process $C(\tilde{x}|x)$ represents a conditional distribution over corrupted samples $\tilde{x}$ given a training sample x.
- The autoencoder learns a reconstruction distribution $p_{\text{reconstruct}}(x|\tilde{x})$ estimated from training pairs $(x, \tilde{x})$ as follows:
  1. Sample a training example x from the training data.
  2. Sample a corrupted version $\tilde{x}$ from $C(\tilde{x}|x)$
  3. Use $(x, \tilde{x})$ as a training example for estimating the autoencoder reconstruction distribution $p_{\text{reconstruct}}(x|\tilde{x}) = p_{\text{decoder}}(x|h)$ with h the output of encoder $f(\tilde{x})$ and p_{decoder} defined by a decoder $g(h)$.

Contractive autoencoder

- Another strategy for regularizing an autoencoder is to use a penalty Ω, as in sparse autoencoders,

$$L\big(x, g(f(x))\big) + \Omega(h, x),$$

with Ω that penalizes derivatives:

$$\Omega(h, x) = \lambda \sum_i \|\nabla_x h_i\|^2 .$$

- This forces the model to learn a function that does not change much when x changes slightly.

Convolutional neural network (CNN)

- Specialized kind of neural network for processing data that has a known grid-like topology.
- E.g. time-series data (1D grid of values), image data (2D grid of pixels).
- CNNs are simply neural networks that use convolution in place of matrix multiplication in at least one of their layers.

Convolution I.

- One dimensional convolution:

$$s(t) = (x * w)(t) = \sum_{-\infty}^{\infty} x(a)w(t-a),$$

where x is input, w denotes a kernel and the output s is sometimes also called feature map.

- Convolution for two-dimensional input X requires a 2D kernel K:

$$S(i,j) = (X * K)(i,j) = \sum_m \sum_n X(m,n)K(i-m, j-n)$$

or

$$S(i,j) = (K * X)(i,j) = \sum_m \sum_n X(i-m, j-n)K(m,n).$$

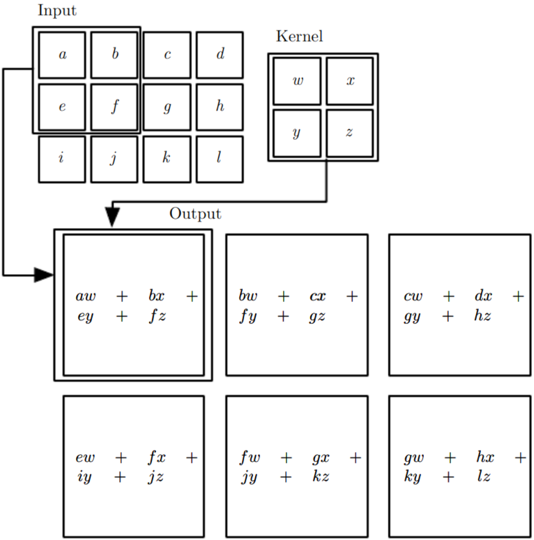
Convolution II.

- The commutative property of convolution arises because of kernel flip.
  - The index into the input increases, but the index into the kernel decreases.
- In practice, **cross-correlation** is used instead, which is the same as convolution but without flipping the kernel:

$$S(i,j) = (K * X)(i,j) = \sum_m \sum_n X(i+m, j+n) K(m,n).$$

- Many machine learning libraries implement cross-correlation but call it convolution.

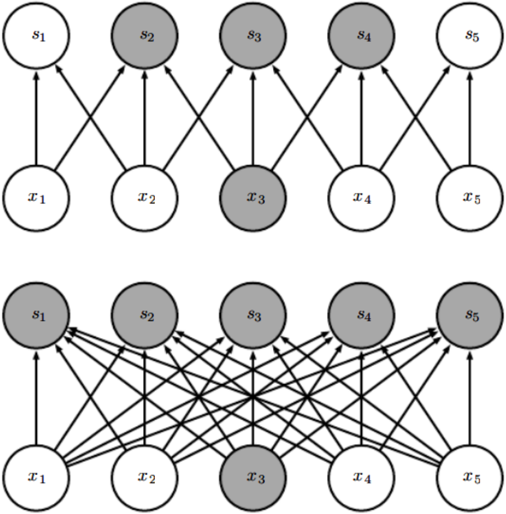
Cross-correlation



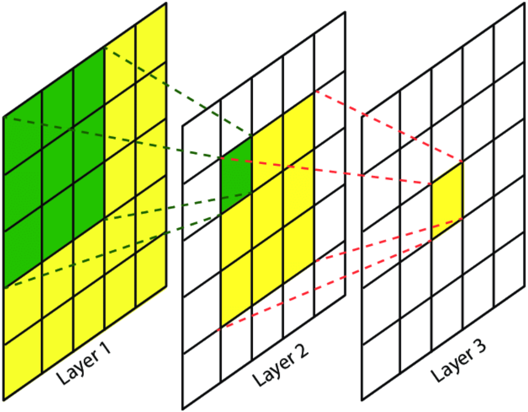
CNNs motivation I.

- Sparse interactions
  - Reduces the memory requirements.
  - Improves statistical efficiency.
  - Requires fewer operations.

CNN interactions



CNN receptive field



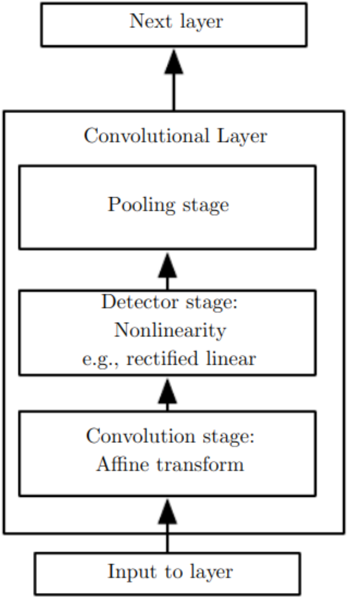
CNNs motivation II.

- Parameter sharing
  - The same parameter is used for more than one function in a model.
  - Efficient in memory requirements.
- Equivariance to translation
  - If the input changes, the output changes in the same way.
  - If we move the object in the input, its representation will move the same amount in the output.
  - Convolution is not naturally equivariant to some other transformations, such as changes in the scale or rotation of an image. Other mechanisms are necessary for handling these kinds of transformations.

Convolutional layer

- Each convolutional layer usually consists of three stages:
  - Convolution stage
    - It performs several convolutions in parallel to produce aset of linear activations.
  - Detector stage
    - Each linear activation is run through a nonlinear activation function (e.g. rectified linear activation function).
  - Pooling stage
    - Replaces the output of the net at a certain location with a summary statistic of the nearby outputs (e.g. max pooling).
    - Makes the representation approximately invariant to small translations of the input.
    - Improves the statistical efficiency and the computational efficiency and reduces memory requirements.

Convolutional layer stages



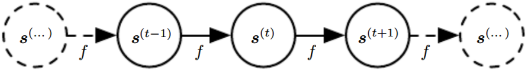
Recurrent neural network (RNN)

- Processing sequence of values $x^{(1)}, \ldots, x^{(N)}$
- RNNs can process sequences of variable length.
  - A network trained on short sequence is able to predict long sequence and vice versa.
- Going from multilayer networks to RNNs $\rightarrow$ parameters sharing.

Unfolding computational graph I.

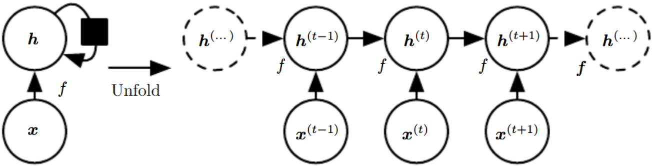
- Classical form of a dynamic system:

$$s^{(t)} = f(s^{(t-1)}; \theta)$$



- Simple recurrent neural network:

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}, \theta)$$

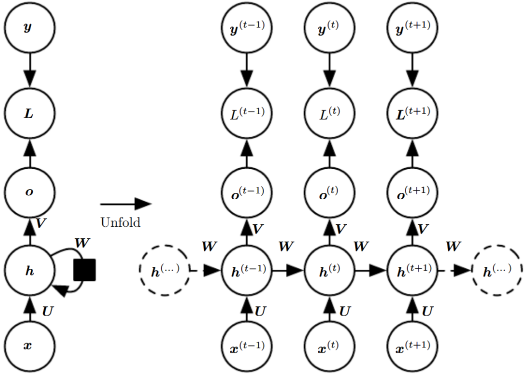


Unfolding computational graph II.

- Typical RNN adds additional output layers.
- $h^{(t)}$ is a kind of lossy summary of the task relevant aspects of the past sequence inputs up to time t
- The topologies of RNNs differ in their ability to hold information from the past.
- The unfolding process has two major advantages:
  - Regardless of the sequence length, the learned model always has the same input size.
  - It is possible to use the same activation function f with the same parameters at every time step.

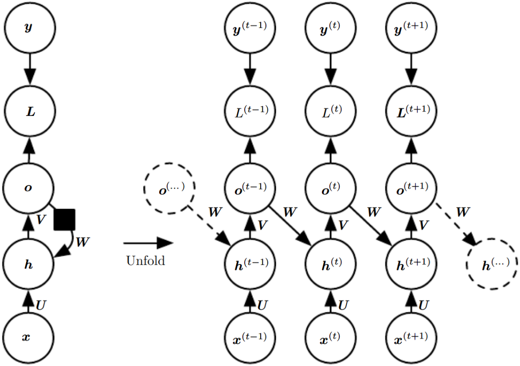
RNN examples I.

- RNNs differ in the unfolded graph topology.
- Examples:
  - Networks that produce an output at each time step and have recurrent connections between hidden units.



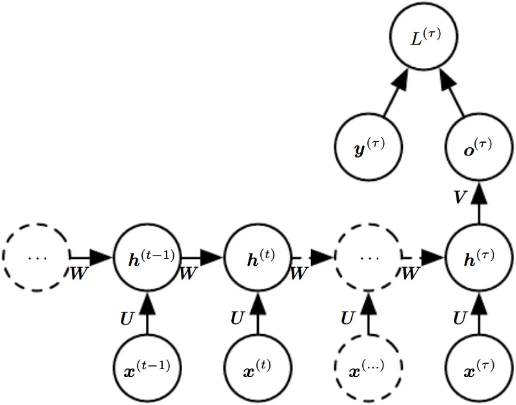
RNN examples II.

- RNNs differ in the unfolded graph topology.
- Examples:
  - Networks that produce an output at each time step and have recurrent connections only from the output at one time step to the hidden units at the next time step.



RNN examples III.

- RNNs differ in the unfolded graph topology.
- Examples:
  - Network with recurrent connections between hidden units that read an entire sequence and then produce a signle output.



Recurrent neural networks - Forward propagation

$$
\begin{aligned}
a^{(t)} =& b + Wh^{(t-1)} + Ux^{(t)}, \\
h^{(t)} =& \tanh(a^{(t)}), \\
o^{(t)} =& c + Vh^{(t)}, \\
\hat{y}^{(t)} =& \mathrm{softmax}(o^{t})
\end{aligned}
$$

- b and c are biases
- U, V and W are weight matrices (input-to-hidden, hiden-to-output and hidden to hidden).

Recurrent neural network - Loss

- Total loss is sum of the losses over all time steps:

$$L\big(\{x^{(1)},\dots,x^{(\tau)}\},\{y^{(1)},\dots,y^{(\tau)}\} = \sum_t L^{(t)}\big)$$
$$= -\sum_t \log p_{\text{model}}\big(y^{(t)}|\{x^{(1)},\dots,x^{(t)}\}\big)$$

- Computing the gradient of this loss function is expensive .
  - Forward pass through unrolled graph followed by backward propagation pass.
  - The runtime $O(\tau)$ can not be reduced by parallelization.
  - States computed in forward pass have to be stored. $\rightarrow$ memory cost is $O(\tau)$.

Recurrent neural network - Back Propagation

- Algorithm: Back propagation trough time (BPTT)
- The network is unrolled and traditional back propagation is applied.

The Challenge of Long-Term Dependencies

- Simple recurrent neural network recurrence relation:

$$h^{(t)} = Wh^{(t-1)}$$

might be simplified to:

$$h^{(t)} = W^t h^{(0)}.$$

If W admits an eigendecomposition of the form:

$$W = Q\Lambda Q^T,$$

with orthogonal Q, the recurrence may be simplified to:

$$h^{(t)} = Q\Lambda^t Q^T h^{(0)}.$$

- Eigenvalues with magnitude less than one decays to zero and eigenvalues with magnitude greater than one explodes.

Long-term dependencies

- The gradient of a long-term interaction has exponentially smaller magnitude than the gradient of a short-term interaction.
- It might take a very long time to learn long-term dependencies,because the signal about these dependencies will tend to be hidden by the smallest fluctuations arising from short-term dependencies
- Learning long dependencies in traditional RNN via SGD is almost impossible for sequences of only length 10 or 20.

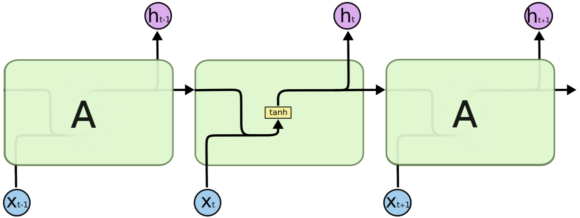
Long-term dependencies - solutions

- Design that operates at multiple time scales:
  - The part of the model that operate at fine-grained time scales can handle small details
  - The part of the model that operate at coarse-grained time scales can transfer information from the distant past.
- Add skip connections trough time.
- Have units with linear self-connections with the weight near one (similar to running average). Such hidden units are called "Leaky units".

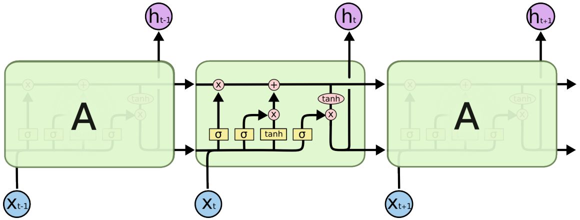
Long Short Term Memory (LSTM)

- Gated RNN.
- Similar to leaky units but the connection weights may change at each time step instead of using a manually chosen constant.
- Can *accumulate* information and *forget* old states.
- Instead of manually deciding when to forget the state, the network learns it by itself.

Vanilla RNN vs LSTM RNN



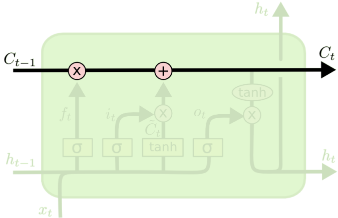
(a) Vanilla RNN cell



(b) LSTM RNN cell

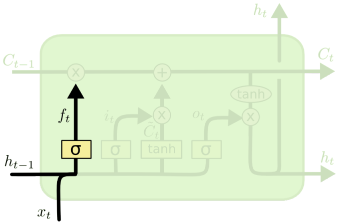
LSTM cell in detail I.

- Cell state stores internal information that is used in output gate.
- It is regulated by forget and input gates.



LSTM cell in detail II.

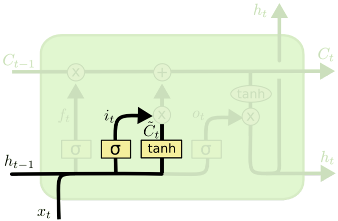
- Forget gate is a sigmoid layer that decides what information will be removed from the cell state.



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

LSTM cell in detail III.

- Input gate is a sigmoid layer that decides which values will be updated.
- Another tanh layer creates a vector of new candidate values that could be added to the cell state.

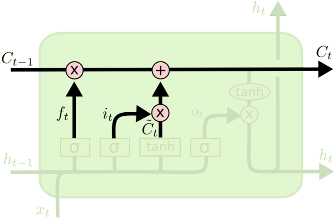


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

LSTM cell in detail IV.

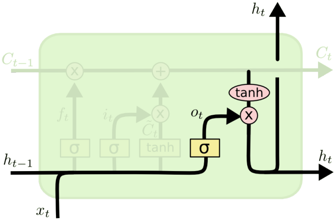
- The old cell state $C_{(t-1)}$ is updated.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

LSTM cell in detail V.

- The output (hidden state) combines the tanh of the cell state and a sigmoid layer called output gate.



$$o_t = \sigma\left(W_o \ [h_{t-1}, x_t] \ + \ b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$