Neural Network Theory and Applications：Lecture 7
神经网络理论及应用：第七讲

Bao-Liang Lu（吕宝粮）
Dept. of Computer Science & Engineering
Shanghai Jiao Tong University
Office: 3-431 Dianxin Bld, Minhang
Tel: 3420-5422
bllu@sjtu.edu.cn

4月18日，25日论文阅读要求

- 每人精读至少一篇论文并作PPT；
- 4月17日12：00之前提交PPT；
- 挑选18位同学上台介绍；每人15分钟介绍，5分钟讨论；
- 4月18日15：00之前通知入选者；
- PPT制作和上台介绍计入课堂表现分数
- 不允许直接把论文原作者的PPT直接拿来用，要加上自己的见解和评论
- 如果有同学找到与自己研究方向相关的有趣论文，并希望介绍的话，请把论文发给助教，需要得到主教的认可。

Paper list (1)

CVPR

- A hierarchical generative model for eye image synthesis and eye gaze estimation
- Boosting Self Supervised Learning Via Knowledge Transfer
- Disentangled Person Image Generation
- HetConv-Heterogeneous Kernel-Based Convolutions for Deep CNNs
- Hydranets Specialized Dynamic Architectures For Efficient Inference
- Hyperparameter Optimization For Tracking With Continuous Deep Q-learning
- Multimodal Explanations Justifying Decisions And Pointing To The Evidence
- NAG: network for adversary generation
- Partial Transfer Learning with selective adversatial networks
- SobolevFusion: 3D Reconstruction scenes undergoing free non rigid motion

ICLR

- CausalGAN: Learning Causal Implicit Generative Models with Adversarial Training
- Spectral Normalization for Generative Adversarial Networks
- Maskgan: Better Text Generation
- Self-ensembling for Visual Domain Adaptation

TCYB

- Generalized Hidden-Mapping Transductive Transfer Learning for Recognition of Epileptic Electroencephalogram Signals
- SCH-GAN Semi-Supervised Cross-Modal Hashing by Generative Adversarial Network

Paper list (2)

NIPS

- Multiple Source Domain Adaptation with Adversarial Training of Neural Networks
- Disconnected Manifold Learning for Generative Adversarial Network
- Generalizing to Unseen Domains via Adversarial Data Augmentation
- Adversarial Text Generation via Feature-Mover's Distance
- Conditional Adversarial Domain Adaptation
- PacGAN: The power of two samples in generative adversarial networks
- Transfer Learning with Neural AutoML
- Assessing the Scalability of Biologically-Motivated Deep Learning Algorithms and Architectures

ICML

- Tempered Adversarial Networks
- Improved Training of Generative Adversarial Networks Using Representative Features
- Theoretical Analysis of Image-to-Image Translation with Adversarial Learning
- Adversarial Learning with Local Coordinate Coding
- Synthesizing Programs for Images using Reinforced Adversarial Learning
- Composite Functional Gradient Learning of Generative Adversarial Models
- GAIN: Missing Data Imputation using Generative Adversarial Nets
- RadialGAN: Leveraging multiple datasets to improve target-specific predictive models using Generative Adversarial Networks

Paper list (3)

ICCV

- MUTAN: Multimodal Tucker Fusion for Visual Question Answering
- Long Short-Term Memory Kalman Filters: Recurrent Neural Estimators for Pose Regularization
- A Multimodal Deep Regression Bayesian Network for Affective Video Content Analyses
- Cross-Modal Deep Variational Hashing
- GANs for Biological Image Synthesis
- Generative Adversarial Networks Conditioned by Brain Signals
- Encoder Based Lifelong Learning
- Learning in an Uncertain World: Representing Ambiguity Through Multiple Hypotheses
- Multimodal Gaussian Process Latent Variable Models with Harmonization
- Supplementary Meta-Learning: Towards a Dynamic Model For Deep Neural Networks

EMNLP

- Adversarial Domain Adaptation for Duplicate Question Detection
- Adversarial Transfer Learning for Chinese Named Entity Recognition with Self-Attention Mechanism
- Improving Multi-label Emotion Classification via Sentiment Classification with Dual Attention Transfer Network
- Out-of-domain Detection based on Generative Adversarial Network

Paper list (4)

- ACL
  - A Helping Hand-Transfer Learning for Deep Sentiment Analysis
  - DSGAN: Generative Adversarial Training for Distant Supervision Relation Extraction
  - Transfer Learning for Context-Aware Question Matching in Information-seeking Conversations in E-commerce
  - Zero-Shot Transfer Learning for Event Extraction
- ICASSP
  - Deep Transfer Learning For EEG-based Brain Computer Interface
  - Language And Noise Transfer In Speech Enhancement Generative Adversarial Network
- AAAI
  - Deep Asymmetric Transfer Network for Unbalanced Domain Adaptation
  - Incorporating GAN for Negative Sampling in Knowledge Representation Learning
  - Multi-Adversarial Domain Adaptation
  - Unsupervised Domain Adaptation with Distribution Matching Machines
  - Wasserstein Distance Guided Representation Learning for Domain Adaptation
- TNNLS
  - Discriminative Transfer Learning Using Similarities and Dissimilarities
  - Inverting the Generator of a Generative Adversarial Network

Paper list (5)

- IJCAI
  - CR-GAN Learning Complete Representations for Multi-view Generation
  - DehazeGAN: When Image Dehazing Meets Differential Programming
  - MIXGAN-Learning Concepts from Different Domains for Mixture Generation
  - Unpaired Multi-Domain Image Generation via Regularized Conditional GANs
  - Unsupervised Cross-Modality Domain Adaptation of ConvNets for Biomedical Image Segmentations with Adversarial Loss
- IEEE-PAMI
  - Joint Semantic and Latent Attribute Modelling for Cross-Class Transfer Learning
  - On the Effectiveness of Least Squares Generative Adversarial Networks
  - StackGAN++ Realistic Image Synthesis with Stacked Generative Adversarial Networks
- TAC
  - Heterogeneous Knowledge Transfer in Video Emotion Recognition, Attribution and Summarization
- TBE
  - Transfer Learning A Riemannian Geometry Framework With Applications to Brain-Computer Interfaces
  - Transfer Learning from Simulations on a Reference Anatomy for ECGI in Personalised Cardiac Resynchronization Therapy

Paper list (6)

- AutoML
  - ASAP architecture search, anneal and prune
  - Automatic Convolutional Neural Architecture Search for Image Classification Under Different Scenes
  - Deep Evolutionary Networks with Expedited Genetic Algorithm for Medical Image Denoising
  - Designing neural networks through neuroevolution
  - Differentiable Neural Network Architecture Search
  - Learning Implicitly Recurrent CNNs Through Parameter Sharing
  - MFAS Multimodal Fusion Architecture Search
  - Neural Architecture Search with Reinforcement Learning
  - sharpDARTS Faster and More Accurate Differentiable Architecture Search
  - Simple And Efficient Architecture Search for Convolutional Neural Networks
  - Single Path One-Shot Neural Architecture Search with Uniform Sampling
  - Stochastic Neural Architecture Search
  - The Art of Getting Deep Neural Networks in Shape
  - Tuning Hyperparameters without Grad Students Scalable and Robust Bayesian Optimisation with Dragonfly
  - Exploring Randomly Wired Neural Networks for Image Recognition

Paper list (7)

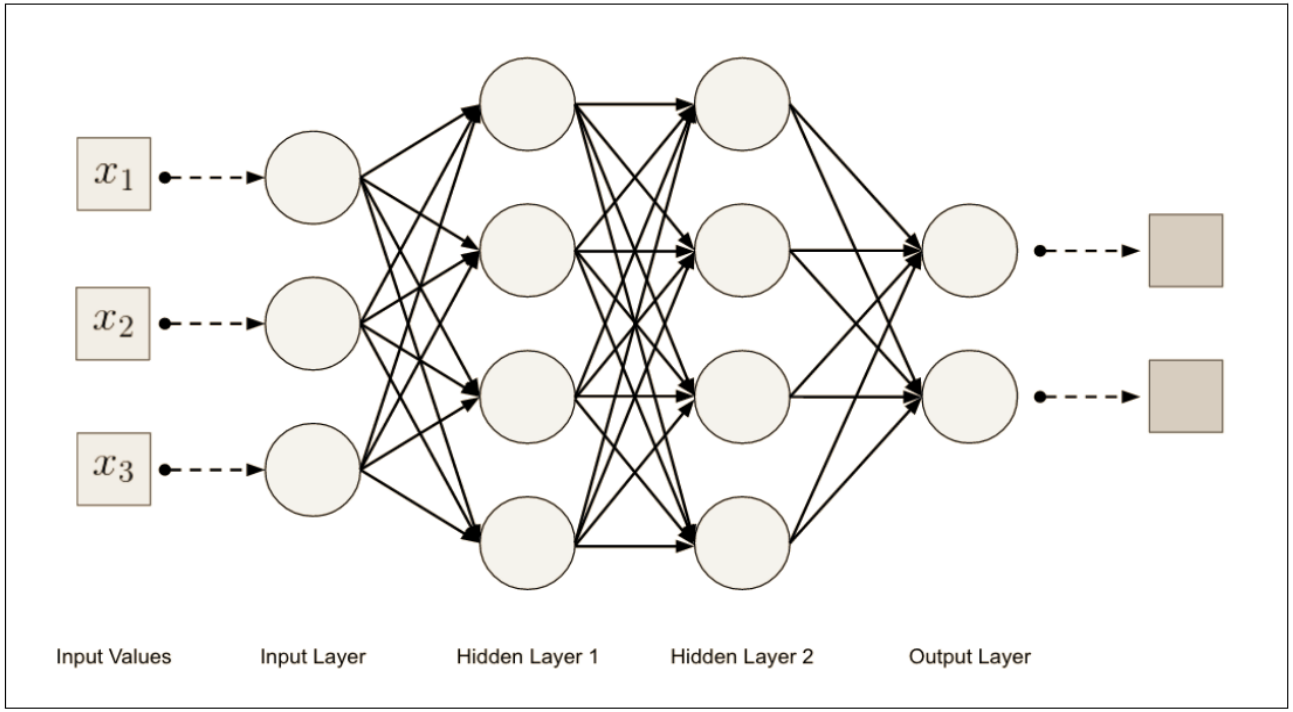
- Others
  - Attention is all you need.
  - Deep reinforcement learning in a handful of trials using probabilistic dynamics models
  - Wasserstein distance guided representation learning
  - Exploiting coarse-to-fine task transfer for aspect-level sentiment classification
  - Generalized zero-shot learning via synthesized examples
  - Importance weighted transfer of samples in reinforcement learning
  - Taskonomy disentangling task transfer learning
  - Global context-aware attention LSTM networks for 3D action recognition
  - Hierarchical graph representation learning with differentiable pooling
  - Learning to transfer
  - Large scale GAN training for high fidelity natural image synthesis
  - Neural message passing for quantum chemistry
  - Conditional adversarial domain adaptation
  - Skeleton-based action recognition using saptio-temporal LSTM network with trust gates
  - The neuro symbolic concept learner, interpreting scenes

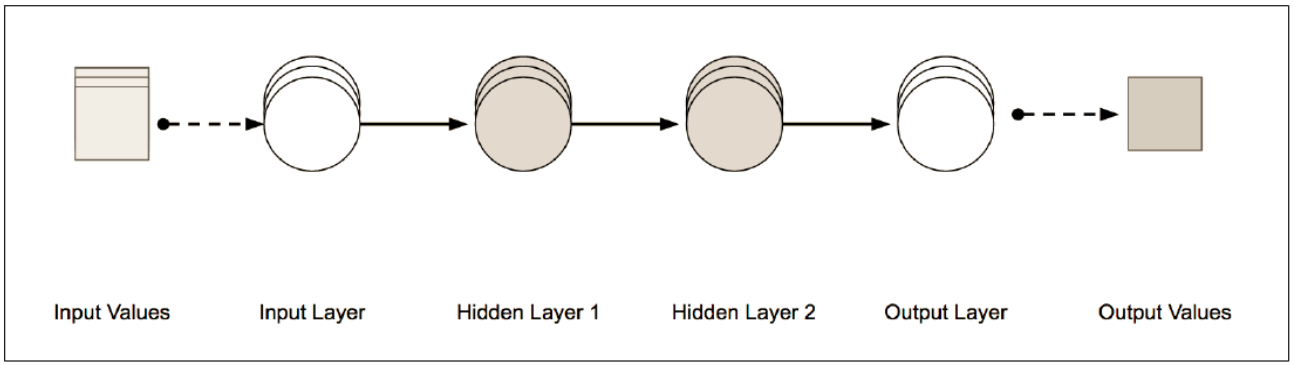
Lecture Seven

1、Recurrent Neural Networks
2、Associative Learning
3、Competitive Networks

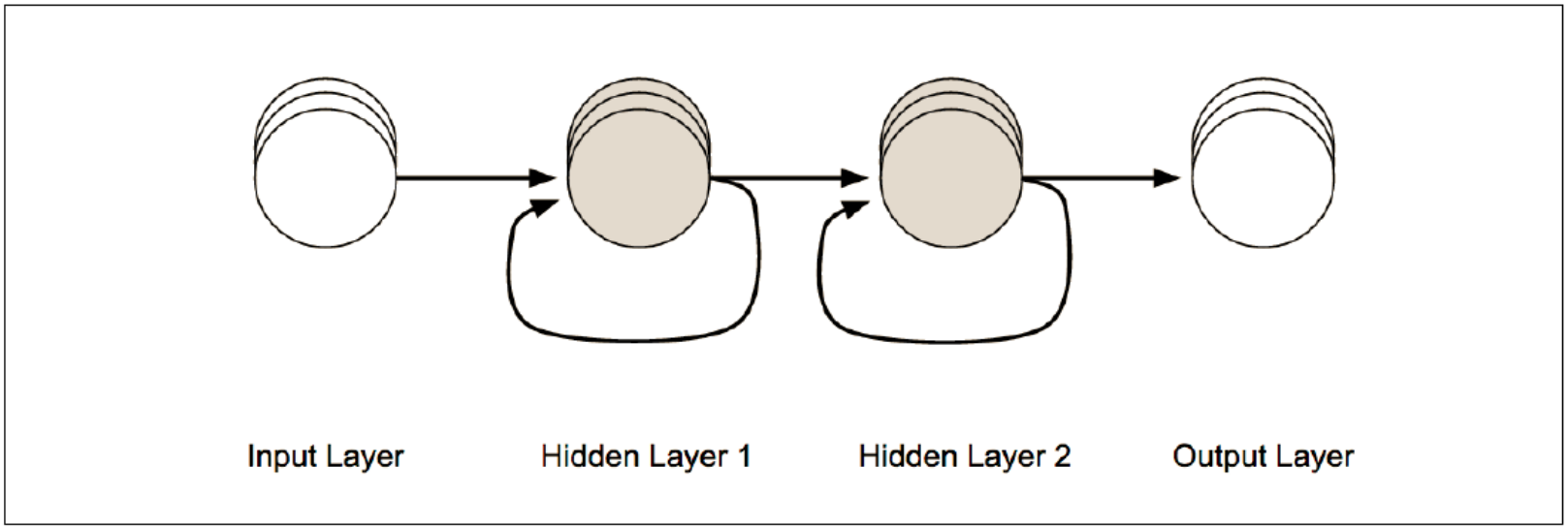
1、Josh Patterson and Adam Gibson, **Deep Learning, A Practitioner's Approach**, O'reilly, 2017
2、Ian Goodfellow, Yoshua Bengio, and Aaron Courville, **Deep Learning**, MIT Press, 2016
3、Alex Graves, **Supervised Sequence Labelling with Recurrent Neural Networks**, Springer, 2012

Feed-forward multilayer neural network

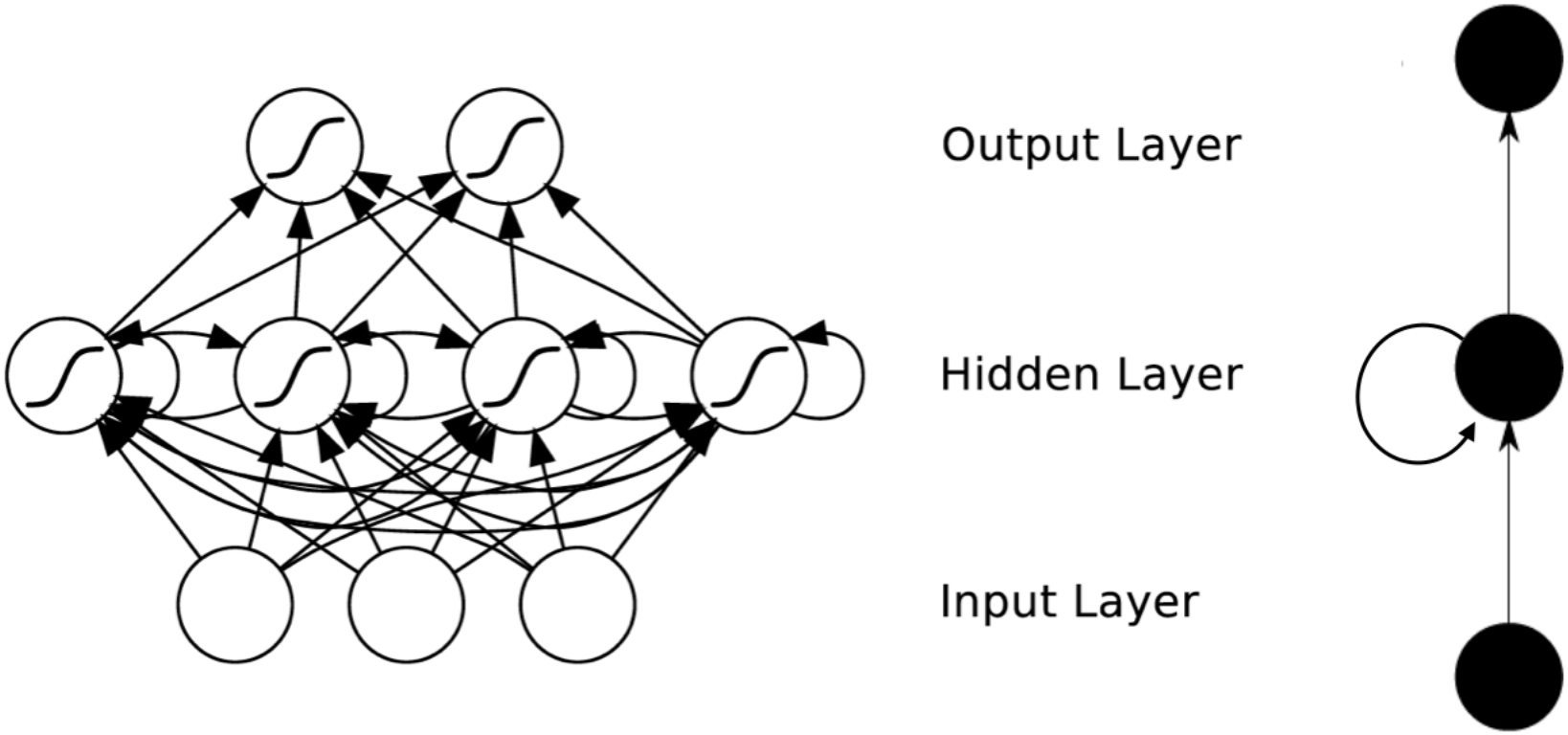




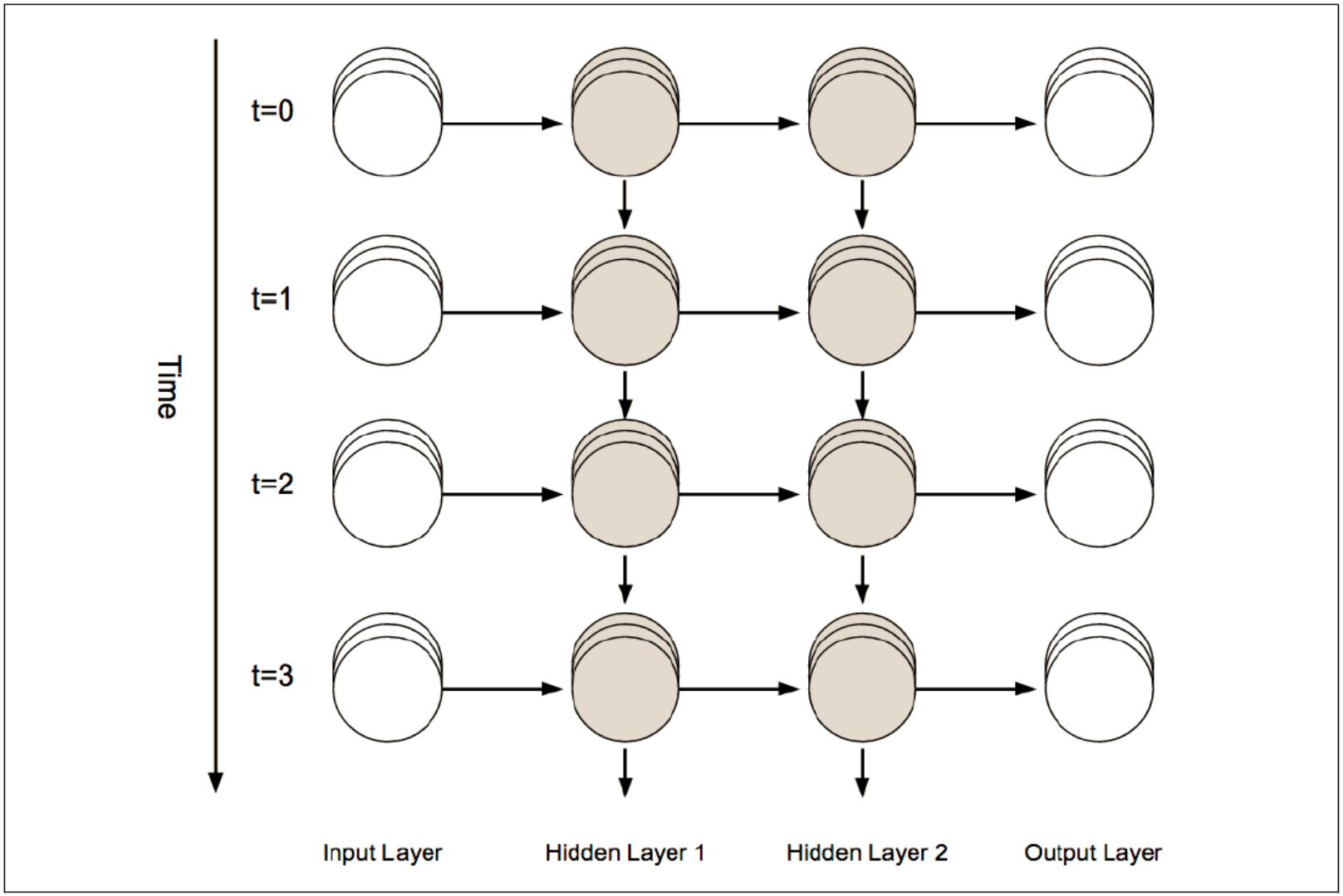
Recurrent connections on hidden-layer nodes



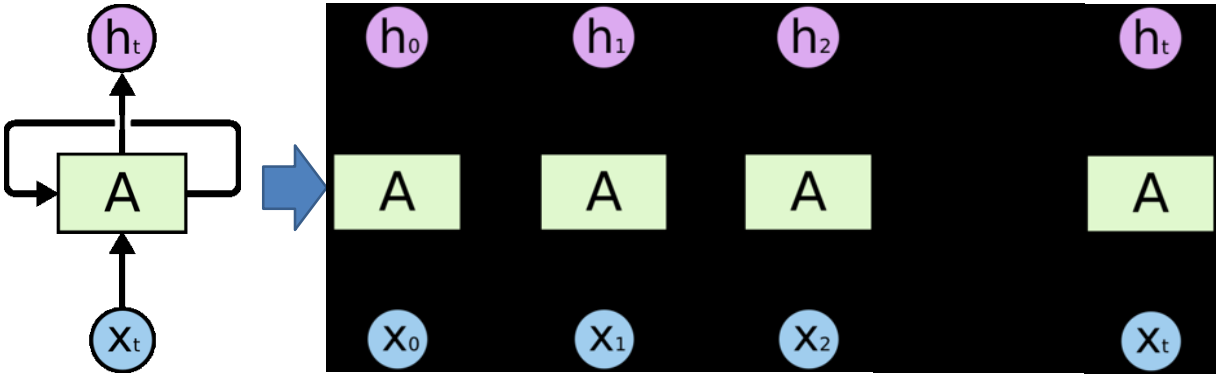
RNN Structure



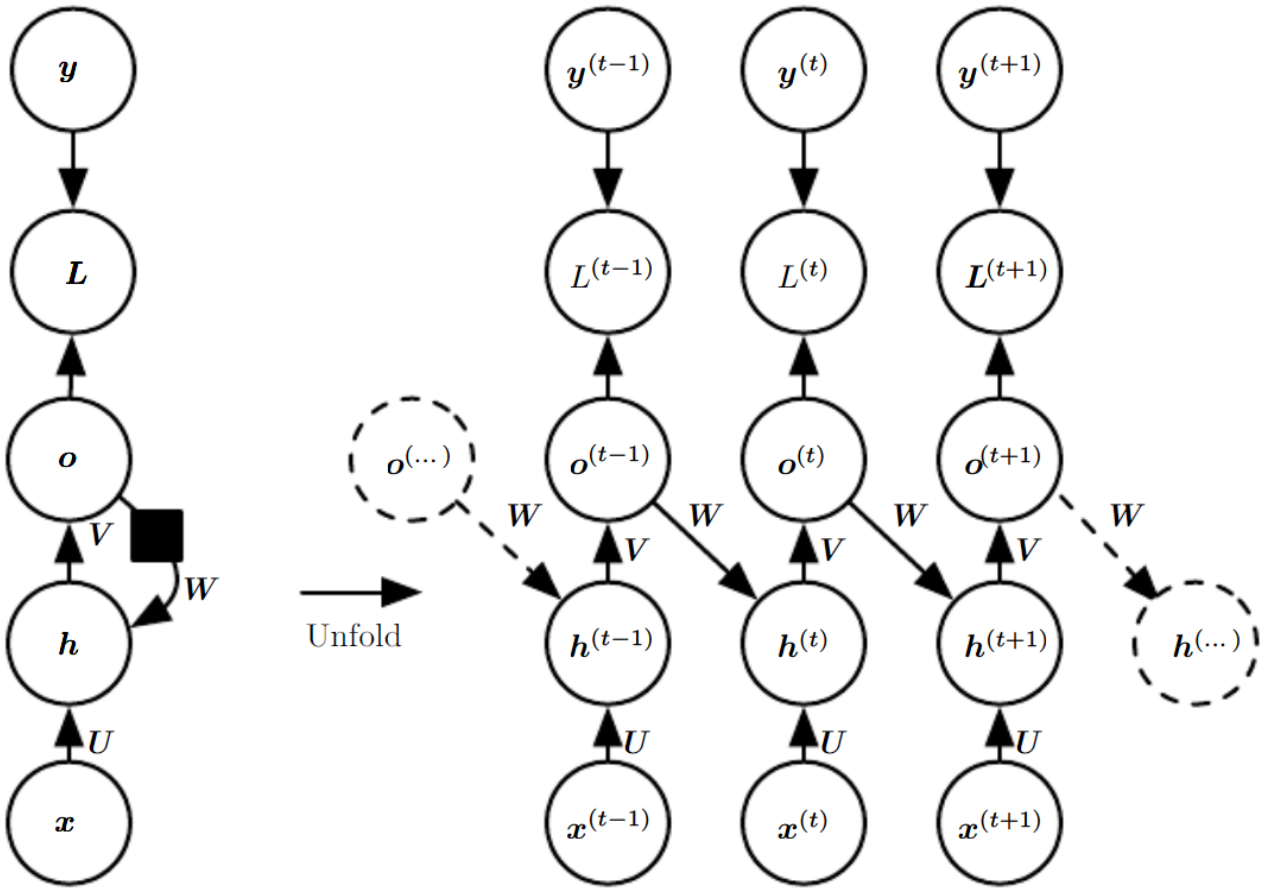
RNN unrolled along the time axis



The Process of Unfolding RNN



Recurrent Neural Networks



Modeling the Time Dimension

- In traditional modeling activities, we typically would see an input-to-output relationship of fixed input size to fixed output size.
- RNNs change this input dynamic to include multiple input vectors, one for each time-step, and each vector can have many columns.
  - One-to-many: sequence output, For example, image captioning takes an image and outputs a sequence of words
  - Many-to-one: sequence input. For example, sentiment analysis where a given sentence is input
  - Many-to-many: For example, video classification: label each frame

Sequences and time-series data

We find sequential data in many problem domains in industry for which our model needs to output a sequence of vectors

- Automated Image captioning
- Speech synthesis
- Music generation
- Plating video games
- Language modeling

"girl in pink dress is jumping in air."

"black and white dog jumps over bar."

"young girl in pink shirt is swinging on swing."

"man in blue wetsuit is surfing on wave."

Lost in Time

- Many classification tools (MLP, SVM) have been applied successfully without modeling the time dimension, assuming independence.
- Other variations of these tools capture the time dynamic by modeling a sliding window of the input (e.g., the previous, current, and next input together as a single input vector).
- A drawback of these tools is that assuming independence in the time connection between model inputs does not allow our model to capture long-range time dependencies.
- Sliding window techniques have a limited window width and will fail to capture any effects large than the fixed window size.

Temporal feedback and loops in connections

- RNNs can have loops in the connections. This allows them to model temporal behavior in domains such as time-series, language, audios, and text.
- Data in these domains are inherently ordered and context sensitive where later values depend on previous ones.
- A RNN includes a feedback loop that it uses to learn from sequences, including sequences of varying lengths.
- RNNs are trained to generate sequences, in which the output at each time-step is based on both the current input and the input at <u>all previous time steps</u>.

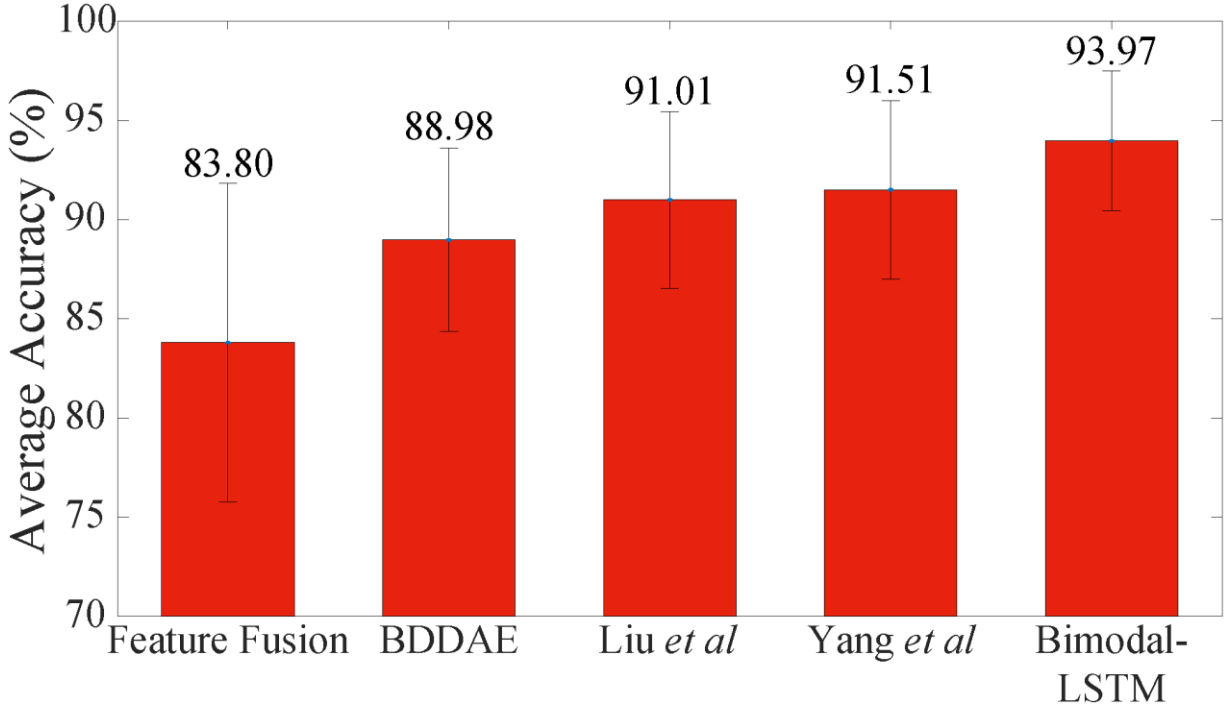
An example: Emotion Recognition using LSTM

- To incorporate the temporal dependency information of features, we use Long Short Term Memory (LSTM) neural networks as a temporal encoder.

Tang, H., Liu, W., Zheng, W.L. and Lu, B.L. Multimodal Emotion Recognition Using Deep Neural Networks. Proc. of International Conference on Neural Information Processing (pp. 811-819), 2017, November. Springer

Experimental Results – SEED dataset

- **Bimodal-LSTM achieves the best accuracy (93.97%) and the smallest standard deviation (7.03%).**



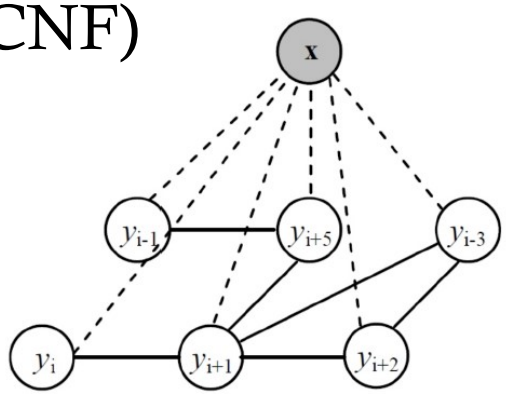
1. Liu, W., Zheng, W.L., Lu, B.L.: Emotion recognition using multimodal deep learning. In: Hirose, A., Ozawa, S., Doya, K., Ikeda, K., Lee, M., Liu, D. (eds.) ICONIP 2016. LNCS, vol. 9948, pp. 521–529. Springer, Cham (2016).
2. Yang, Y., Wu, Q.J., Zheng, W.L., Lu, B.L.: EEG-based emotion recognition using hierarchical network with subnetwork nodes. IEEE Transactions on Cognitive and Developmental Systems (2017)

Incorporating Temporal Dependency

- Vigilance is a dynamic changing process because the intrinsic mental states of users involve temporal evolution.
- Continuous conditional random field (CCRF)
- Continuous conditional neural field (CCNF)

$$P(\mathbf{y}|\mathbf{x}) = \frac{exp(\Psi)}{\int_{-\infty}^{\infty} exp(\Psi) d\mathbf{y}},$$

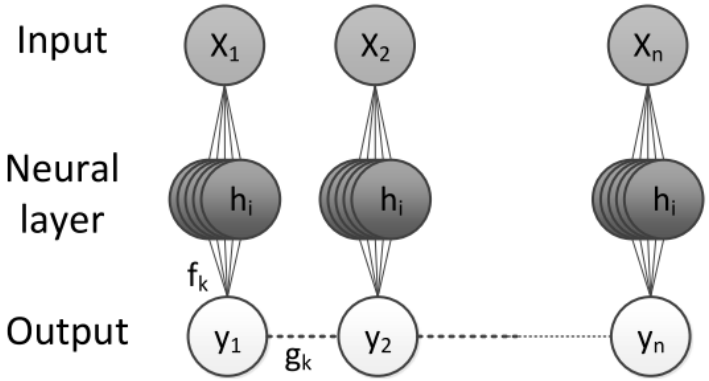
$$\Psi = \sum_i \sum_{k=1}^{K_1} \alpha_k f_k(y_i, \mathbf{x}_i, \boldsymbol{\theta}_k) + \sum_{i,j} \sum_{k=1}^{K_2} \beta_k g_k(y_i, y_j)$$



$$f_k(y_i, \boldsymbol{x}_i, \boldsymbol{\theta}_k) = -(y_i - h(\boldsymbol{\theta}_k, \boldsymbol{x}_i))^2, and$$

$$h(\boldsymbol{\theta}, \boldsymbol{x}_i) = \frac{1}{1 + e^{-\boldsymbol{\theta}^T \boldsymbol{x}_i}},$$

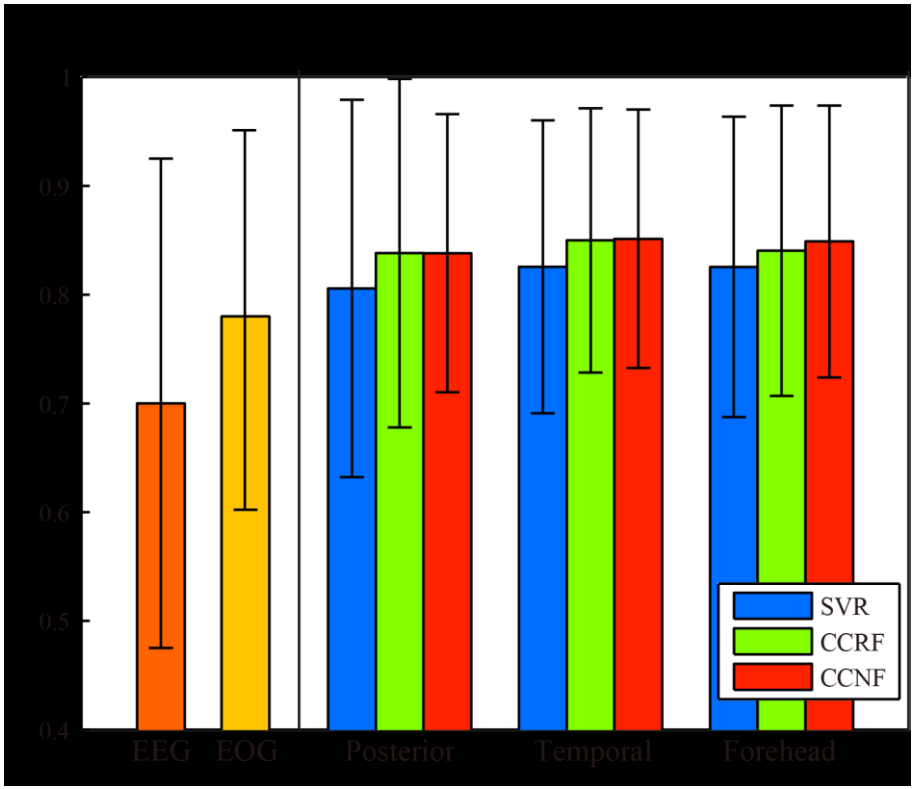
$$g_k(y_i, y_j) = -\frac{1}{2} S_{i,j}^{(k)} (y_i - y_j)^2$$



Wei-Long Zheng and Bao-Liang Lu, A Multimodal Approach to Estimating Vigilance Using EEG and Forehead EOG, Journal of Neural Engineering, 2017

Experimental Results

- For the forehead setup, the mean COR/RMSE of SVR, CCRF, and CCNF are 0.83/0.10, 0.84/0.10, and **0.85/0.09,** respectively. The CCNF achieves the best performance with higher accuracies and lower standard deviations.



Wei-Long Zheng and Bao-Liang Lu, A Multimodal Approach to Estimating Vigilance Using EEG and Forehead EOG, Journal of Neural Engineering, 2017

Recurrent Neural Networks (RNNs)

- RNNs allow for both parallel and sequential computation, and in principle can compute anything a traditional computer can compute
- Unlike traditional computers, however, RNNs are similar to the human brain, which is a large feedback network of connected neurons that somehow can learn to translate a lifelong sensory input stream into a sequence of useful motor outputs
- The brain is a remarkable role model as it can solve many problems current machine can not yet solve.

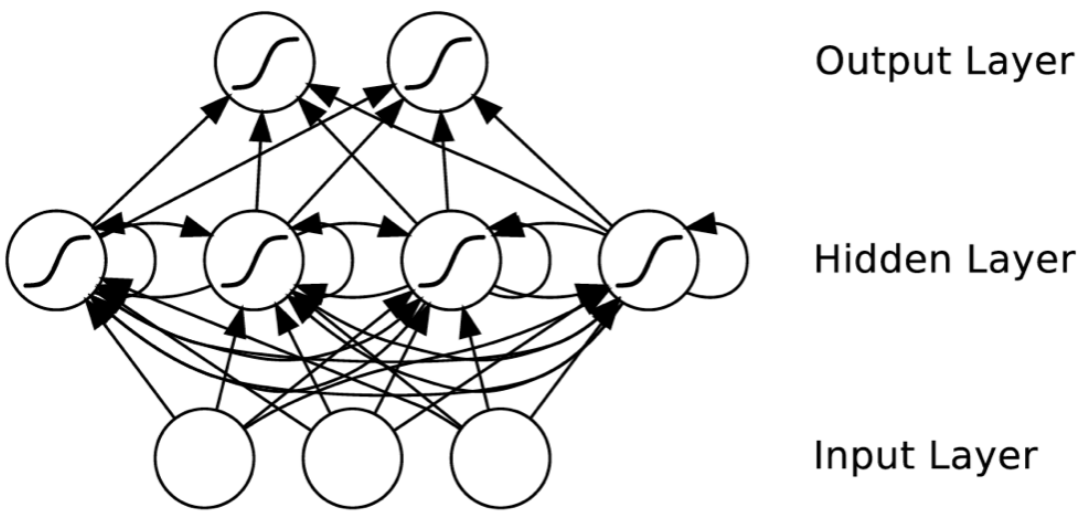
  (Juergen Schmidhuber)

- Brain is a super algorithm. Its output is an algorithm.

  (Gen Matsumoto)

Modeling the Time Dimension

- RNNs are well suited for modeling functions for which the input and/or output is composed of vectors that involve a time dependency between the values.
- RNNs model the time aspect of data by creating cycles in the network

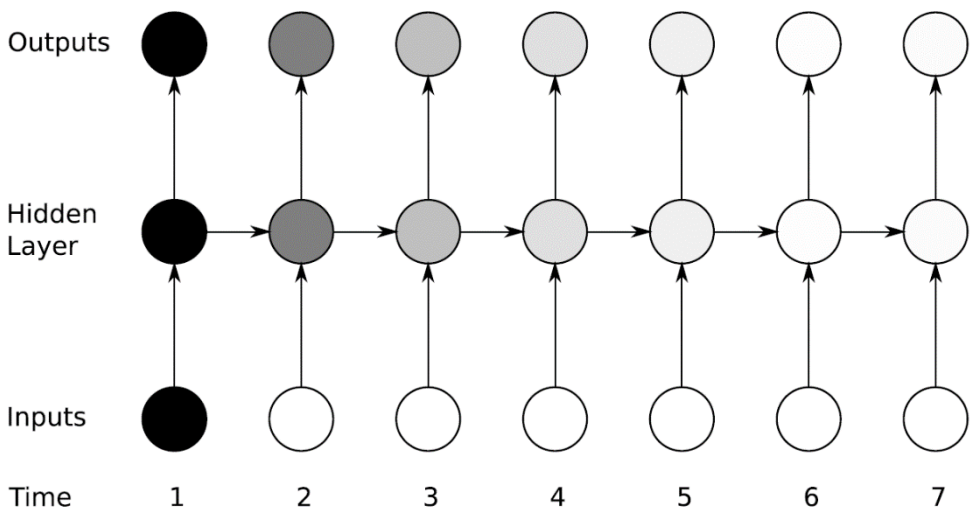


Sharing Parameters

- To go from MLPs to RNNs, we need to take advantage of sharing parameters across different parts of a model.
- Parameter sharing makes it possible to extend and apply the model to examples of different lengths and generalize across them.
- If we had separate parameters for each value of the time index, we could not generalize to sequence lengths not seen during training, and not share statistical strength across different sequence lengths and across different positions in time.
- Such sharing is particularly important when a specific piece of information can occur at multiple positions within the sequence.

Vanishing Gradient Problem

- RNNs are known to have issues with the "vanishing gradient problem". This issue occurs when the gradients become too large or too small and make it difficult to model long-range dependencies (10 time-steps or more) in the structure of the input dataset.
- The most effective way to get around this issue is to use the LSTM variant RNNs



Unfolding Computational Graphs

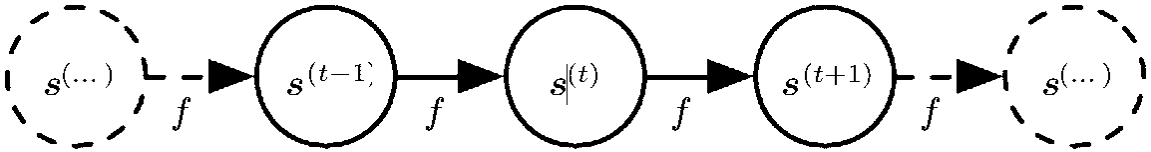
- Consider the classical form of a dynamical system

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

- For a finite number of time steps τ , the graph can be unfolded by applying the definition $\tau - 1$ times. For example, if we unfold equation for $\tau = 3$ time steps, we obtain

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

- The unfolded computational graph can be illustrated as follow.



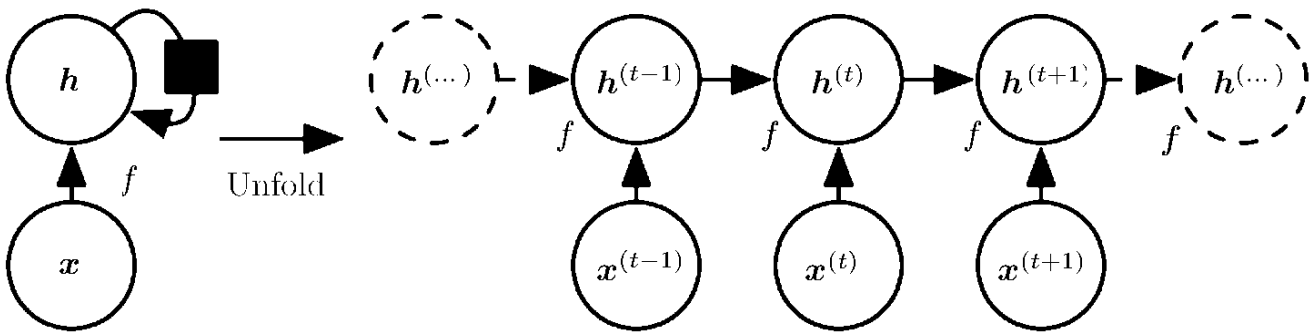
Unfolding Computational Graphs

- **Consider a dynamical system driven by an external signal $x^{(t)}$**

- **Many recurrent neural networks use the following equation or a similar one to define the values of their hidden units**

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

- **A recurrent network with no outputs**



Unfolding Computational Graphs

- We can represent the unfolded recurrence after t steps with a function $g^{(t)}$

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

- The unfolding process thus introduces two major advantages:
  - Regardless of the sequence length, the learned model always has the same input size, because it is specified in terms of transition from one state to another state, rather than specified in terms of a variable-length history of states.
  - It is possible to use the same transition function f with the same parameters at every time step.

Unfolding Computational Graphs

- **These two factors make it possible to learn a single model f that operates on all time steps and all sequence lengths, rather than needing to learn a separate model $g^{(t)}$ for all possible time steps. Learning a single, shared model allows generalization to sequence lengths that did not appear in the training set, and allows the model to be estimated with far fewer training examples than would be required without parameter sharing.**

Recurrent Neural Networks

- Armed with the graph unrolling and parameter sharing ideas, we can design a wide variety of recurrent neural networks.
- Some examples of important design patterns for recurrent neural networks include the following:
  - Recurrent networks that produce an output at each time step and have recurrent connections between hidden units
  - Recurrent 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
  - Recurrent networks with recurrent connections between hidden units, that read an entire sequence and then produce a single output

Recurrent Neural Networks

- Recurrent networks that produce an output at each time step and have recurrent connections between hidden units

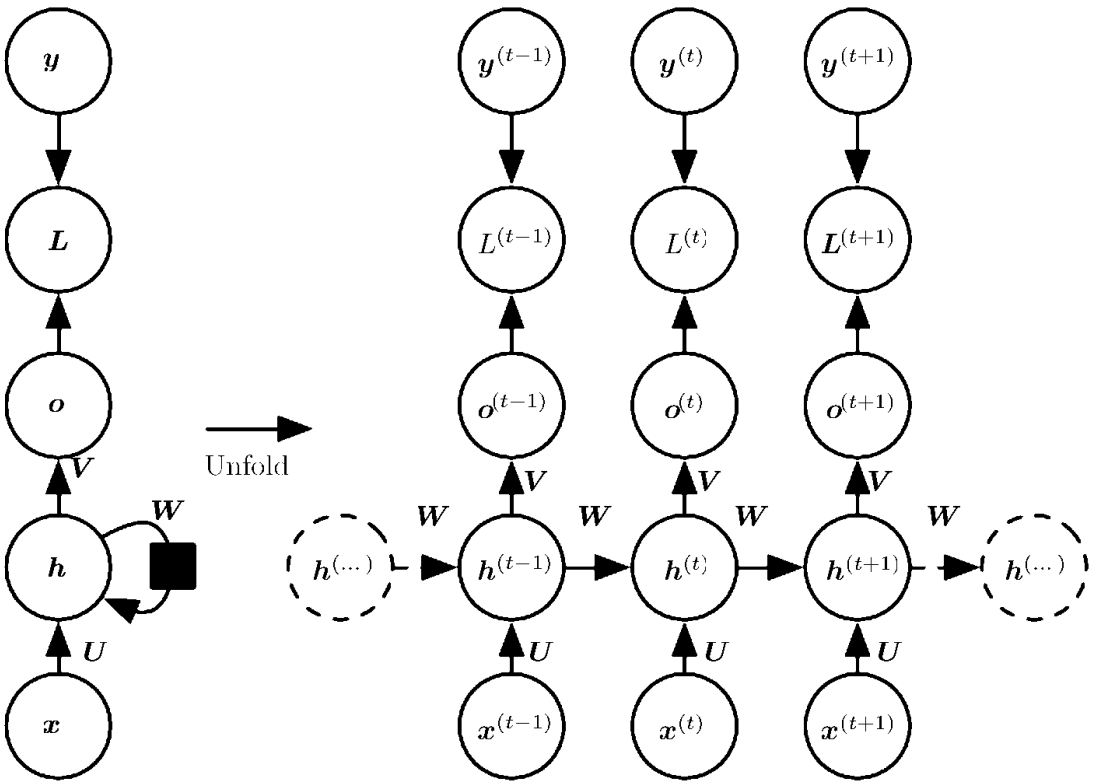


Figure 3

Recurrent Neural Networks

- Recurrent 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

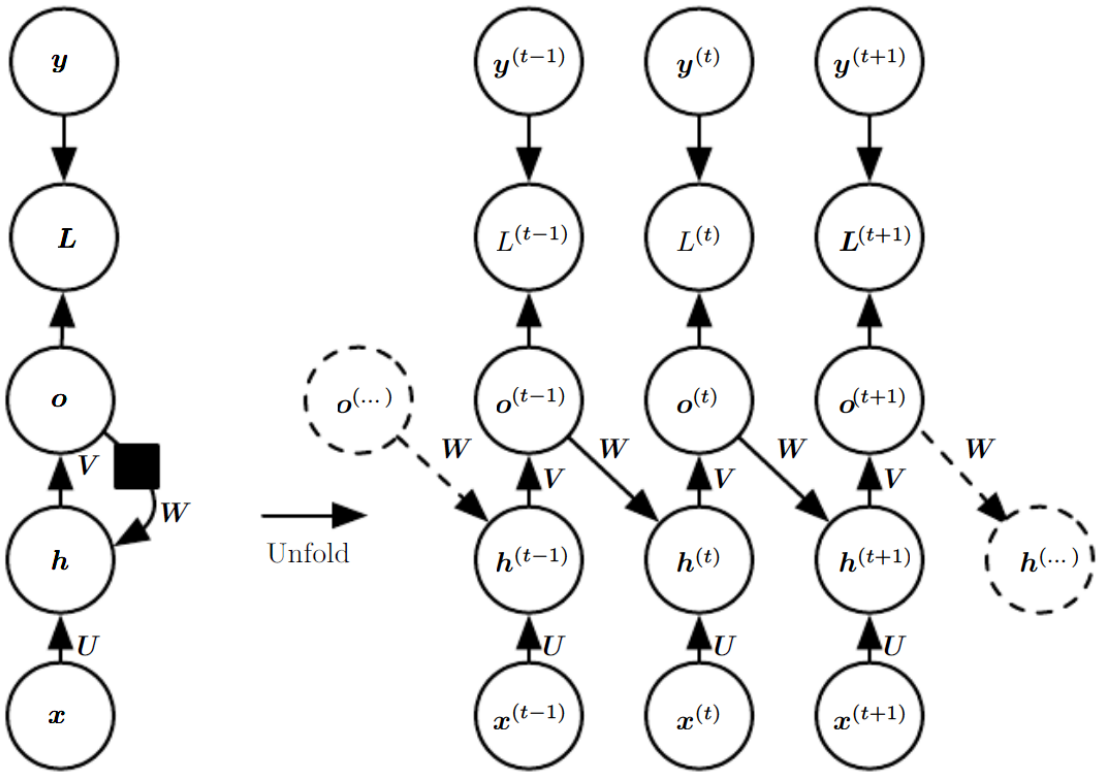


Figure 4

Recurrent Neural Networks

- Recurrent networks with recurrent connections between hidden units, that read an entire sequence and then produce a single output

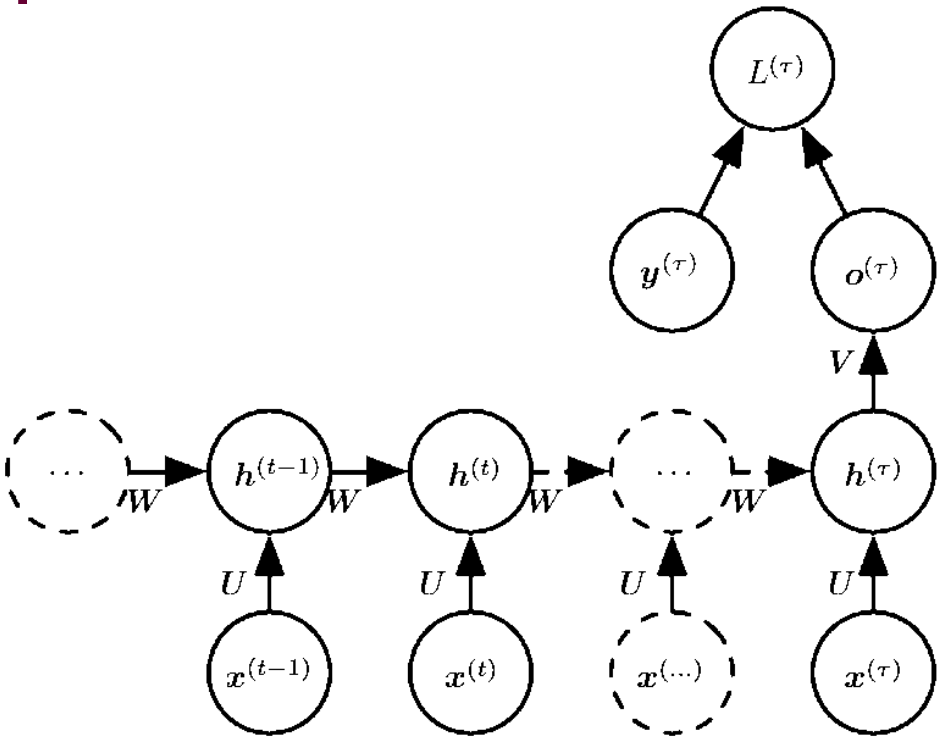


Figure 5

Recurrent Neural Networks

- **The recurrent neural network is universal in the sense that any function computable by a Turing machine can be computed by such a recurrent network of a finite size.**

- **Forward propagation begins with a specification of the initial state $h^{(0)}$. Then, for each time step from $t = 1$ to $t = \tau$, we apply the following update equations:**

$$\boldsymbol{a}^{(t)} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)},$$

$$\boldsymbol{h}^{(t)} = \tanh(\boldsymbol{a}^{(t)}),$$

$$\boldsymbol{o}^{(t)} = \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)},$$

$$\hat{\boldsymbol{y}}^{(t)} = \mathrm{softmax}(\boldsymbol{o}^{(t)}),$$

Recurrent Neural Networks

- **The total loss for a given sequence of x values paired with a sequence of y values would then be just the sum of the losses over all the time steps. For example, if $L^{(t)}$ is the negative log-likelihood of $y^{(t)}$ given $x^{(1)}, \ldots, x^{(t)}$, then**

$$= \sum_t L^{(t)}$$

$$= -\sum_t \log p_{\text{model}} \left(y^{(t)} \mid \{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}\} \right)$$

Back-Propagation through time (BPTT) Algorithm

- The use of back-propagation on the unrolled graph is called the back-propagation through time (BPTT) algorithm.
- The nodes of our computational graph include the parameters U, V, W, b and c as well as the sequence of nodes indexed by t for $x^{(t)}, h^{(t)}, o^{(t)}$ and $L^{(t)}$. We start the recursion with the nodes immediately preceding the final loss

$$\frac{\partial L}{\partial L^{(t)}} = 1.$$

- The gradient $\nabla_{o^{(t)}} L$ on the outputs at time step t, for all i, t, is as follows:

Computing the Gradient in a Recurrent Neural Network

- **At the final time step τ , $h^{(\tau)}$ only has $o^{(\tau)}$ as a descendent, so its gradient is simple:**

$$\nabla_{\boldsymbol{h}^{(\tau)}} L = \boldsymbol{V}^\top \nabla_{\boldsymbol{o}^{(\tau)}} L.$$

- **We can then iterate backwards in time to back-propagate gradients through time, from $t = \tau - 1$ down to $t = 1$, noting that $h^{(t)}$ $(for\ t < \tau)$ has as descendent both $o^{(t)}$ and $h^{(t+1)}$. Its gradient is thus given by**

$$\begin{aligned}\nabla_{\boldsymbol{h}^{(t)}} L &= \left(\frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{h}^{(t)}}\right) (\nabla_{\boldsymbol{h}^{(t+1)}} L) + \left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}}\right) (\nabla_{\boldsymbol{o}^{(t)}} L) \\ &= \boldsymbol{W}^\top (\nabla_{\boldsymbol{h}^{(t+1)}} L) \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t+1)}\right)^2\right) + \boldsymbol{V}^\top (\nabla_{\boldsymbol{o}^{(t)}} L)\end{aligned}$$

Computing the Gradient in a Recurrent Neural Network

- **Once the gradients on the internal nodes of the computational graph are obtained, we can obtain the gradients on the parameter nodes.**
- **We introduce dummy variables $W^{(t)}$ that are defined to be copies of W but with each $W^{(t)}$ used only at time step t. We may then use $\nabla_{W^{(t)}}$ to denote the contribution of the weights at time step t to the gradient.**

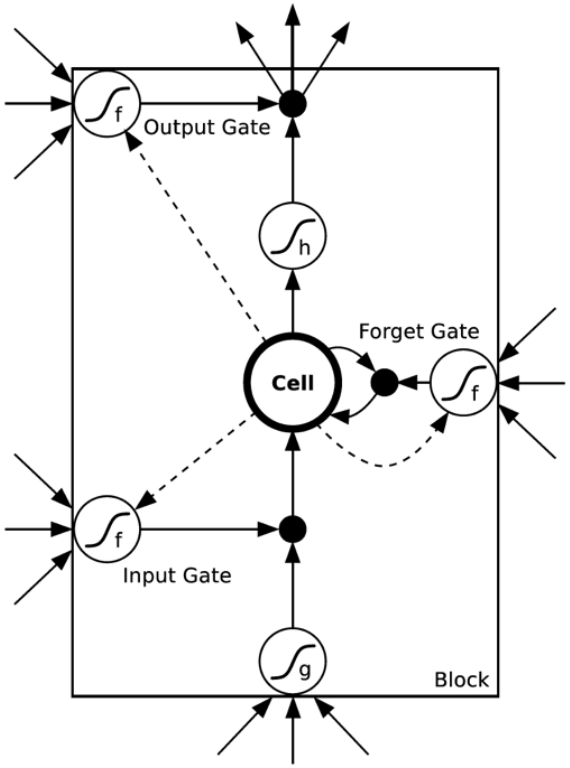
Computing the Gradient in a Recurrent Neural Network

- **Using this notation, the gradient on the remaining parameters is given by:**

$$
\begin{aligned}
\nabla_{\boldsymbol{c}} L &= \sum_t \left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{c}}\right)^{\top} \nabla_{\boldsymbol{o}^{(t)}} L = \sum_t \nabla_{\boldsymbol{o}^{(t)}} L \\
\nabla_{\boldsymbol{b}} L &= \sum_t \left(\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{b}^{(t)}}\right)^{\top} \nabla_{\boldsymbol{h}^{(t)}} L = \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \nabla_{\boldsymbol{h}^{(t)}} L \\
\nabla_{\boldsymbol{V}} L &= \sum_t \sum_i \left(\frac{\partial L}{\partial o_i^{(t)}}\right) \nabla_{\boldsymbol{V}} o_i^{(t)} = \sum_t \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) \boldsymbol{h}^{(t)\top} \\
\nabla_{\boldsymbol{W}} L &= \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}}\right) \nabla_{\boldsymbol{W}^{(t)}} h_i^{(t)} \\
&= \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) \boldsymbol{h}^{(t-1)\top} \\
\nabla_{\boldsymbol{U}} L &= \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}}\right) \nabla_{\boldsymbol{U}^{(t)}} h_i^{(t)} \\
&= \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) \boldsymbol{x}^{(t)\top}
\end{aligned}
$$

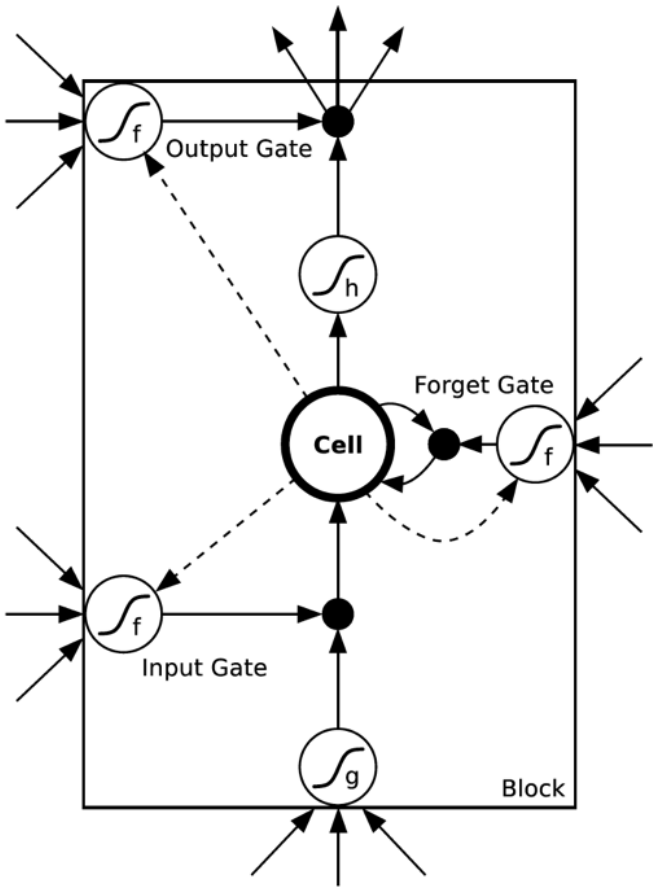
Long-Sort-Term Memory Networks

- LSTM networks were introduced in 1997 by Hochreiter and Schmidhuber.
- LSTM networks are the most commonly used variation of RNNs.
- The critical component of the LSTM is the memory CELL and the GATES (including the forget gate, but also the input gate).
- The contents of the memory cell are modulated by the input gate and forget gates.



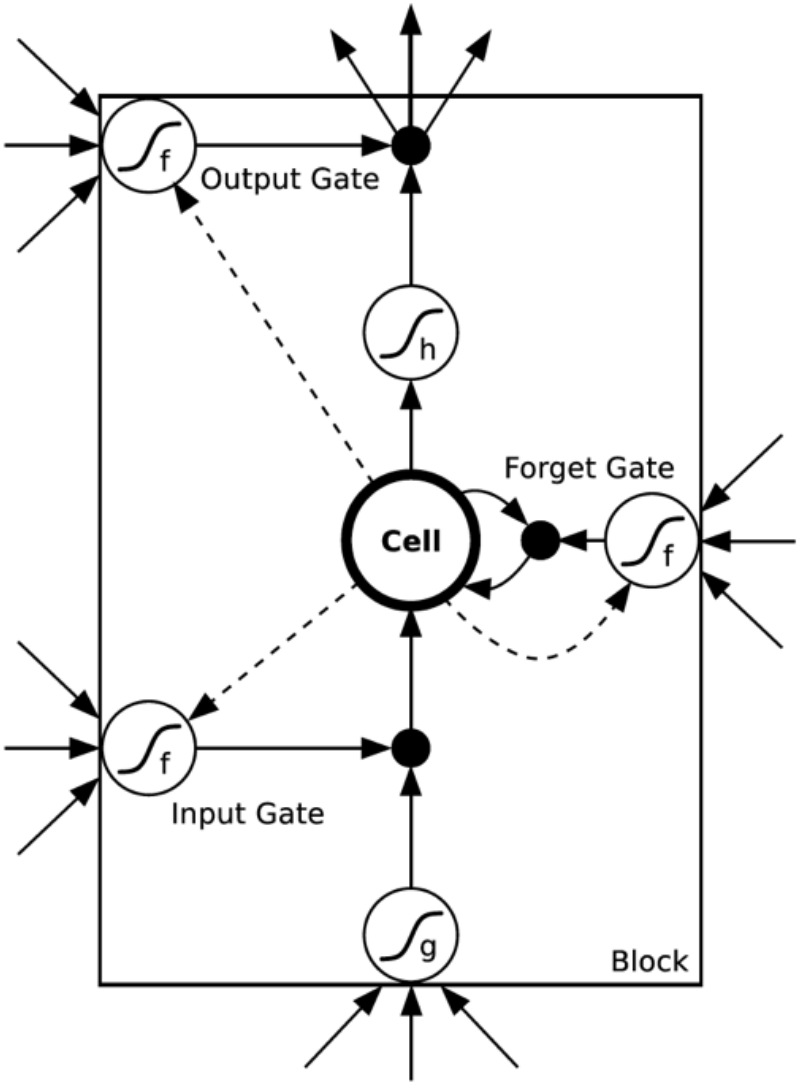
Long-Sort-Term Memory Networks

- Assuming that both of these gates are closed, the contents of the memory cell will remain unmodified between one time-step and the next.
- The gating structure allows information to be retained across many time-steps, and consequently also allows gradients to flow across many time-steps.
- This allows the LSTM model to overcome the vanishing gradient problem that occurs with most RNN models.



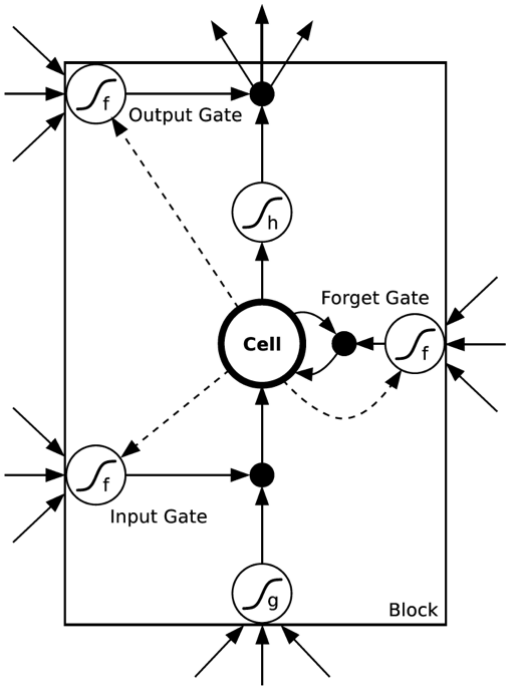
Components in an LSTM unit

- Three gates:
  - Input gate (input modulation gate)
  - Forget gate
  - Output gate
- Block input
- Memory cell
- Output activation function
- Peephole connection



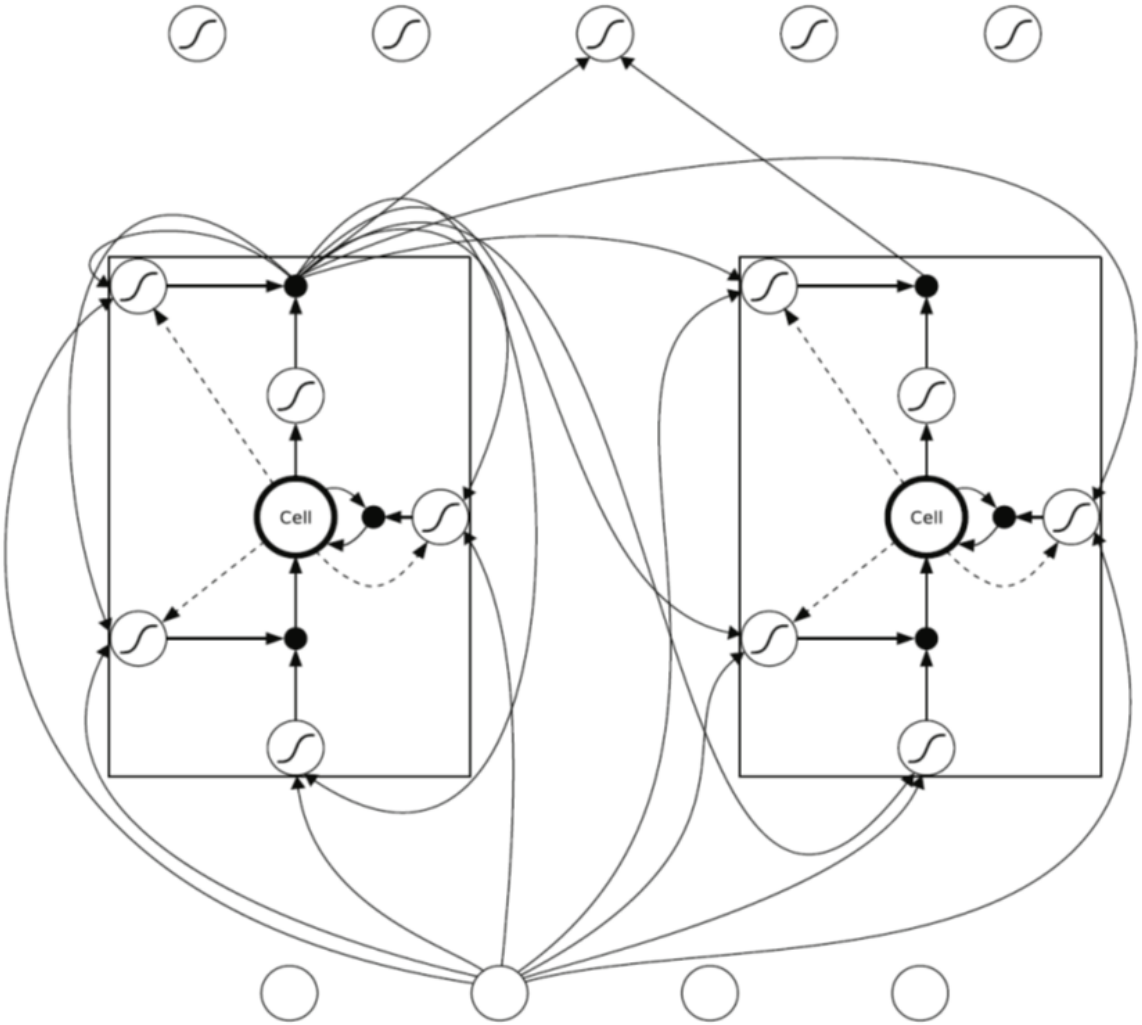
A Note about the Forget Gate

- An activation output of 1.0 means "remember everything" and activation output of 0.0 means "forget everything".
- From a different perspective a better name for the forget gate might be the "remember gate"!



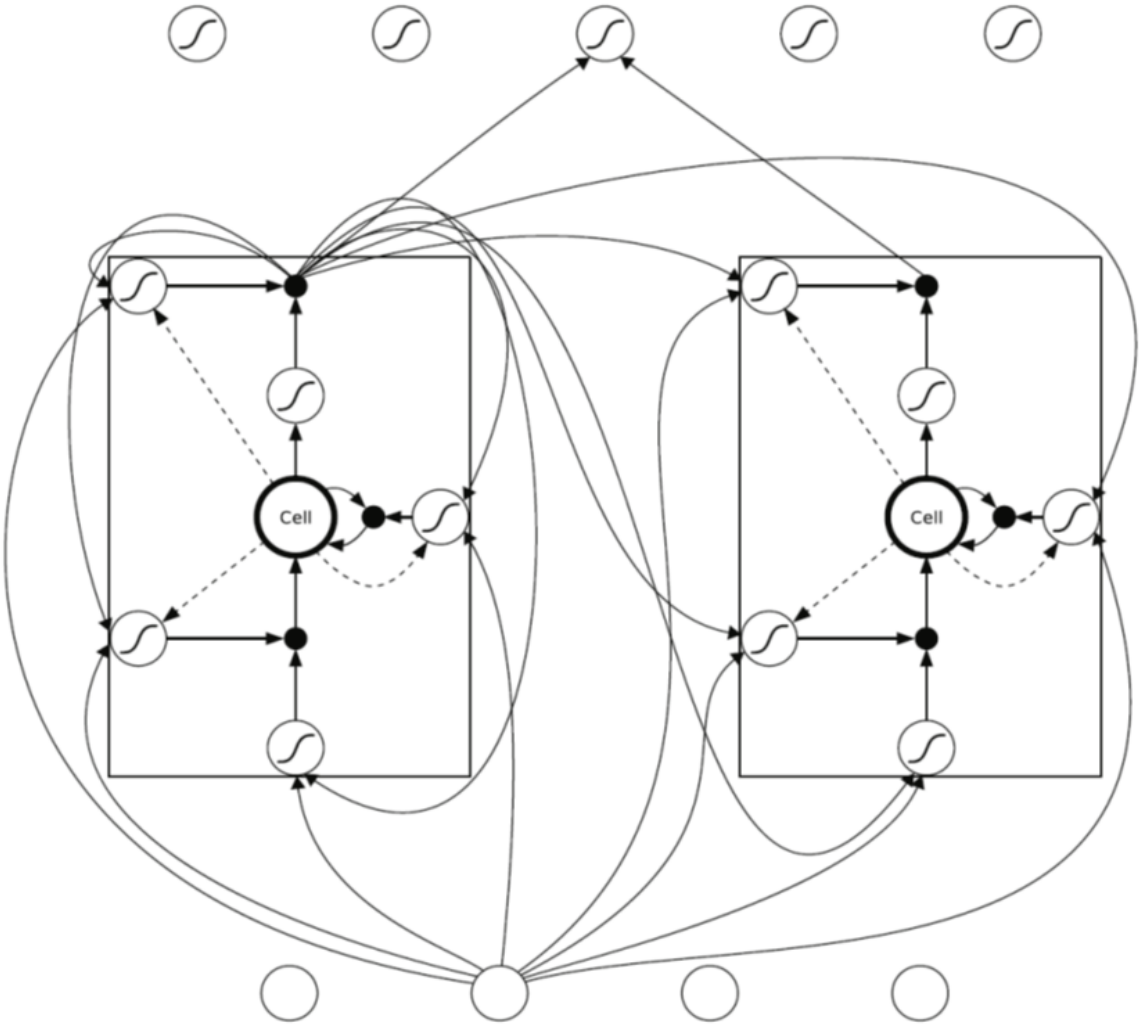
An Example of LSTM Network

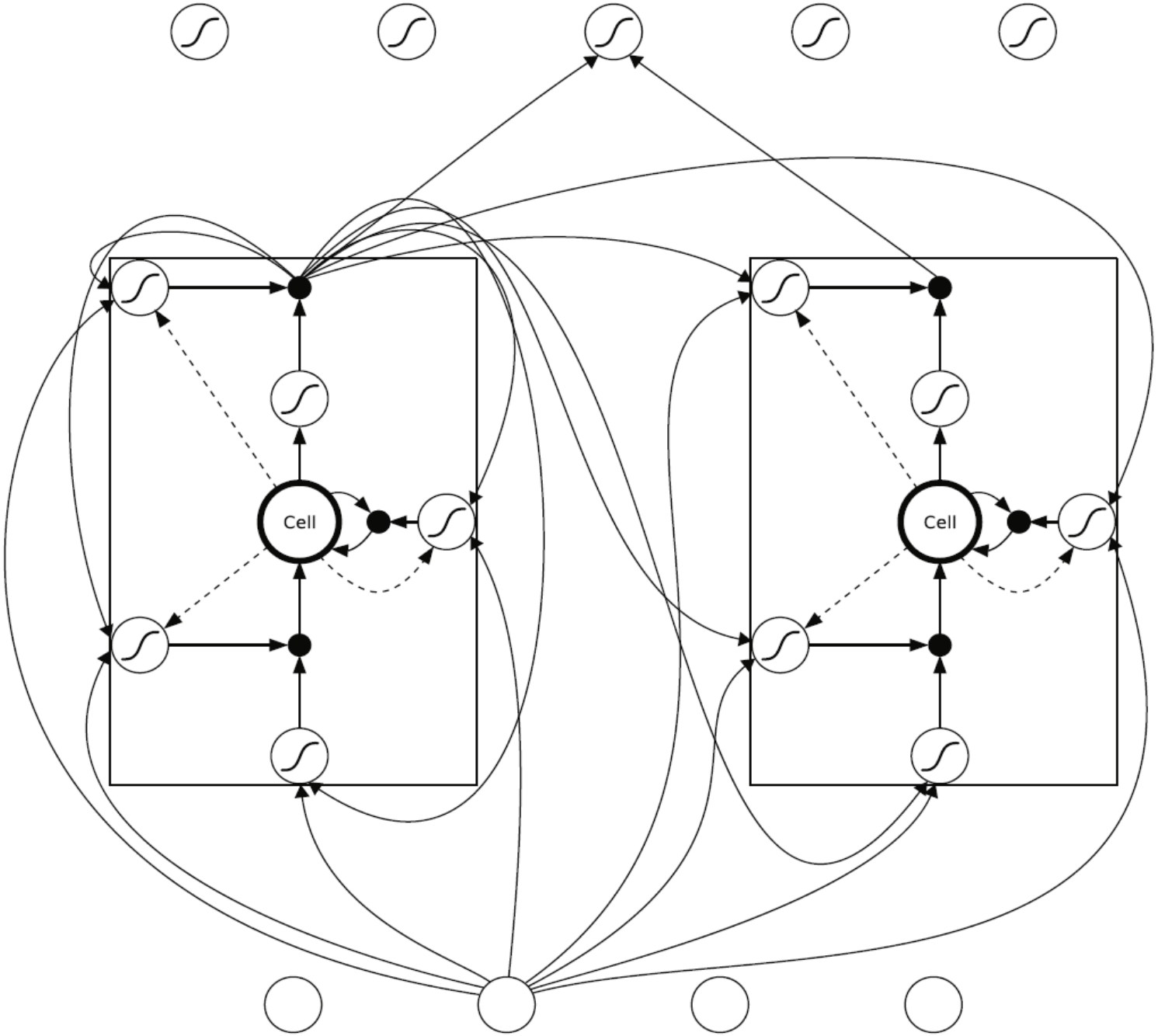
- The network consists of four input units, a hidden layer of two single-cell LSTM memory blocks and five output units.
- Each block has four inputs but only one output.
- Not all connections are shown.



LSTM unit

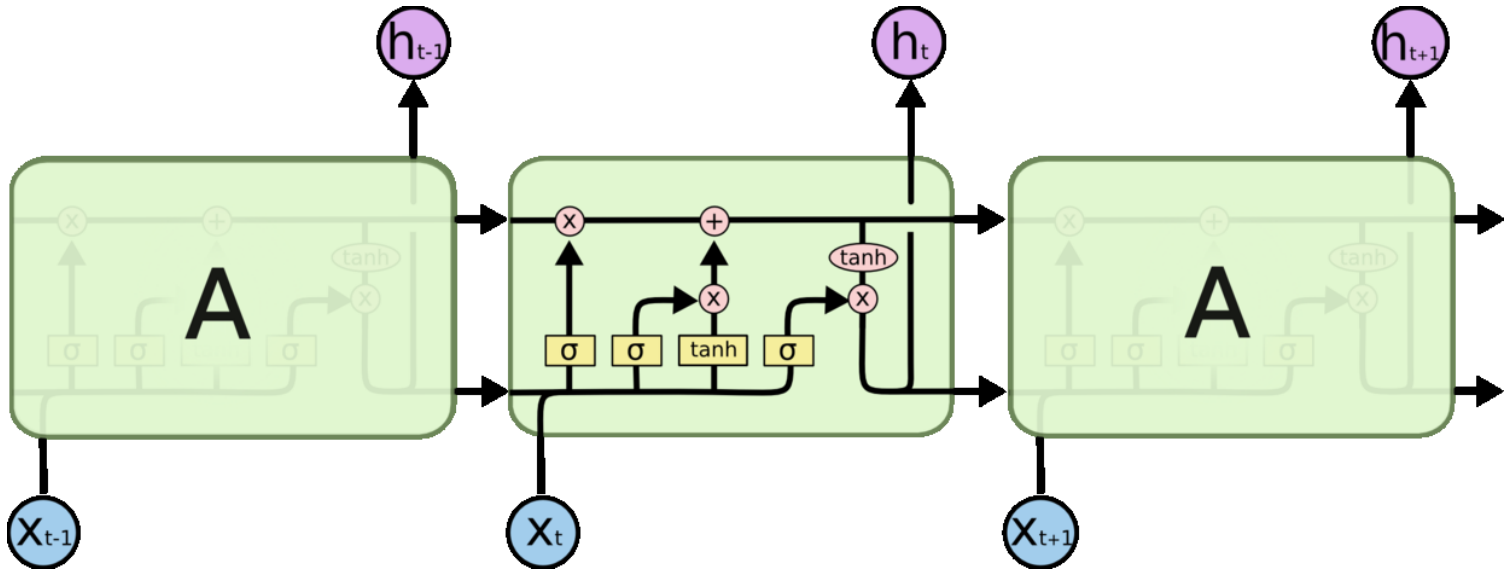
- Each LSTM unit has two types of connections:
  - Connections from the previous time-step (output of those units)
  - Connections from the previous layer




Cell
Cell

Long Short Term Memory (LSTM) Network

- LSTM neural network is a RNN using LSTM blocks.
- Using the gated units, LSTM can prevent the vanishing (and exploding) gradient problem and has the ability to learn information from long sequences.



Forward Pass

Input Gates

$$a_\iota^t = \sum_{i=1}^{I} w_{i\iota} x_i^t + \sum_{h=1}^{H} w_{h\iota} b_h^{t-1} + \sum_{c=1}^{C} w_{c\iota} s_c^{t-1}$$

$$b_\iota^t = f(a_\iota^t)$$

Forget Gates

$$a_\phi^t = \sum_{i=1}^{I} w_{i\phi} x_i^t + \sum_{h=1}^{H} w_{h\phi} b_h^{t-1} + \sum_{c=1}^{C} w_{c\phi} s_c^{t-1}$$

$$b_\phi^t = f(a_\phi^t)$$

Forward Pass-2

Cells

$$a_c^t = \sum_{i=1}^{I} w_{ic} x_i^t + \sum_{h=1}^{H} w_{hc} b_h^{t-1}$$

$$s_c^t = b_\phi^t s_c^{t-1} + b_\iota^t g(a_c^t)$$

Output Gates

$$a_\omega^t = \sum_{i=1}^{I} w_{i\omega} x_i^t + \sum_{h=1}^{H} w_{h\omega} b_h^{t-1} + \sum_{c=1}^{C} w_{c\omega} s_c^t$$

$$b_\omega^t = f(a_\omega^t)$$

Cell Outputs

$$b_c^t = b_\omega^t h(s_c^t)$$

Backward Pass

$$\epsilon_c^t \stackrel{\text{def}}{=} \frac{\partial \mathcal{L}}{\partial b_c^t} \qquad \epsilon_s^t \stackrel{\text{def}}{=} \frac{\partial \mathcal{L}}{\partial s_c^t}$$

Cell Outputs

$$\epsilon_c^t = \sum_{k=1}^{K} w_{ck}\delta_k^t + \sum_{h=1}^{H} w_{ch}\delta_h^{t+1}$$

Output Gates

$$\delta_\omega^t = f'(a_\omega^t)\sum_{c=1}^{C} h(s_c^t)\epsilon_c^t$$

Backward Pass-2

States

$$\epsilon_s^t = b_\omega^t h'(s_c^t)\epsilon_c^t + b_\phi^{t+1}\epsilon_s^{t+1} + w_{c\iota}\delta_\iota^{t+1} + w_{c\phi}\delta_\phi^{t+1} + w_{c\omega}\delta_\omega^t$$

Cells

$$\delta_c^t = b_\iota^t g'(a_c^t)\epsilon_s^t$$

Forget Gates

$$\delta_\phi^t = f'(a_\phi^t)\sum_{c=1}^{C} s_c^{t-1}\epsilon_s^t$$

Input Gates

$$\delta_\iota^t = f'(a_\iota^t)\sum_{c=1}^{C} g(a_c^t)\epsilon_s^t$$

LMS layers

- A basic layer accepts an input vector x (nonfixed) and gives output y.
- The output y is influenced by the input x and the history of all inputs
- The layer is influenced by the history of inputs through the recurrent connections.
- The RNN has some internal state that is updated every time we input a vector to the layer.
- The state consists of a single hidden vector

LSTM Training

- LSTM networks use supervised learning to update the weights in the network.
- They train on one input vector at a time in a sequence of vectors.
- Every noninput unit computes its current activation at any given time-step.
- This activation value is computed as the nonlinear function of the weighted sum of the activation of all units from which it receives connections.
- For each input vector in the sequence of input, the error is equal to the sum of the deviations of all target signals from corresponding activation computed by the network

Multimodal Emotion Recognition Using Deep Neural Networks

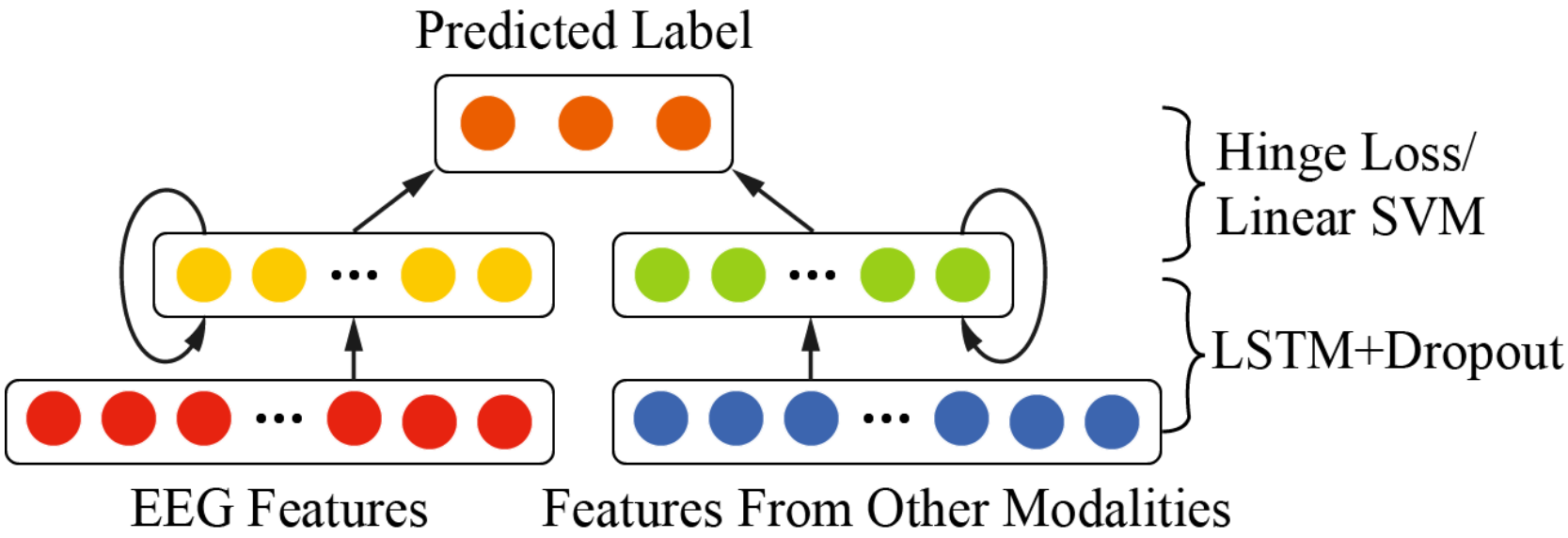
Tang, H., Liu, W., Zheng, W.L. and Lu, B.L.Multimodal Emotion Recognition Using Deep Neural Networks. Proc. of International Conference on Neural Information Processing (pp. 811-819), 2017, November. Springer, Cham.

Multimodal Emotion Recognition using LSTM

- Previous work only utilizes current time step's features.
- To incorporate the temporal dependency information of features, we use Long Short Term Memory (LSTM) neural networks as a temporal encoder.

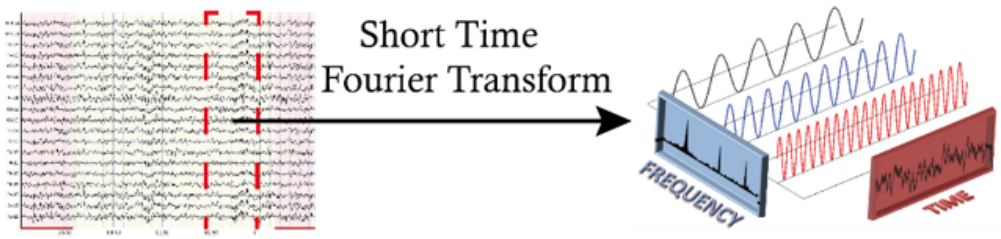
Bimodal-LSTM Network

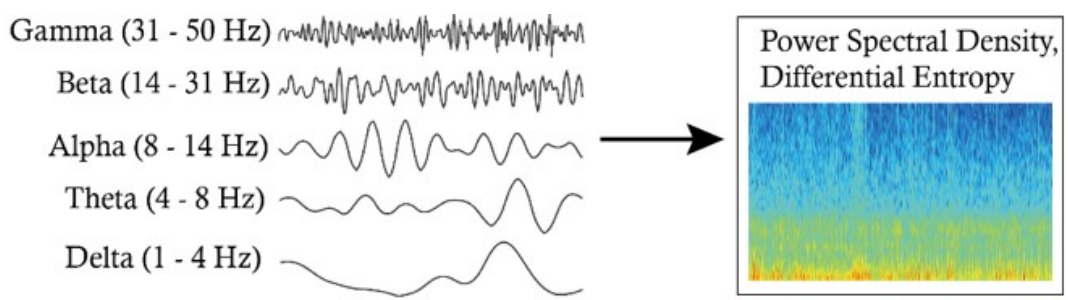
- Two LSTM layers are built for each modality to extract temporal information.
- One classification layer with L2-regularized hinge loss.
- Dropout is applied for better generalization performance.



Experiment details: EEG feature extraction

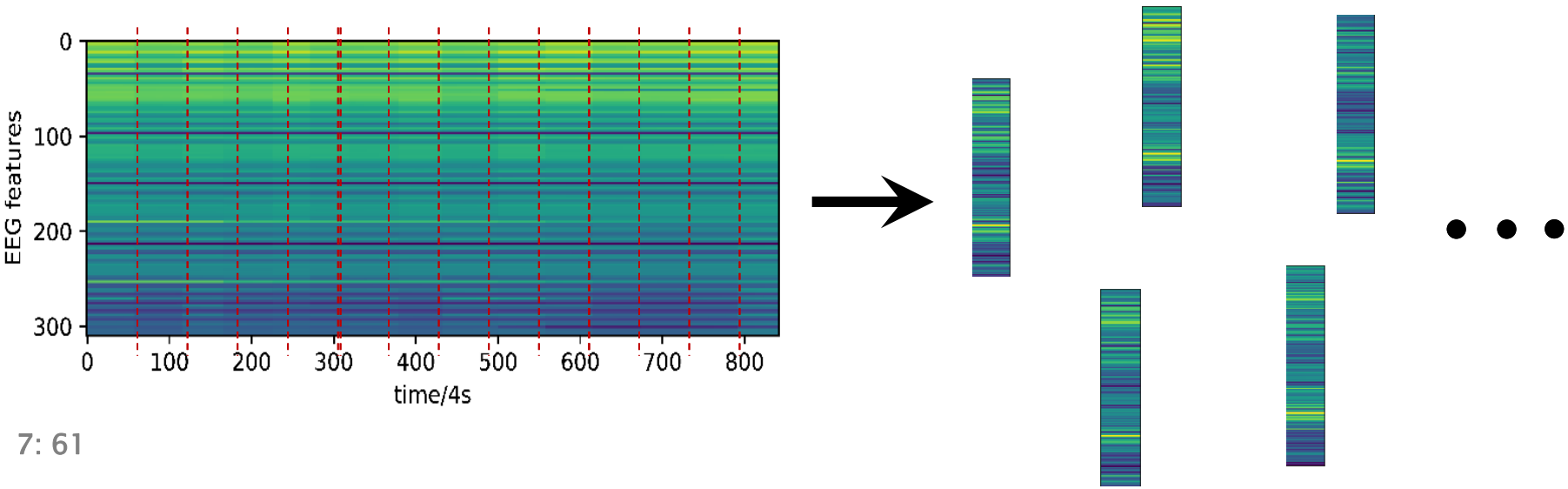
- The feature extraction procedure includes STFT, and power spectral density calculation. (Example on SEED: Window size of STFT is 4-s without overlapping, 5 bands for 62 channels → Totally 900 310-dimension feature vectors; for each movie clip, there are 60 310-dimension feature vectors continuously in temporal domain.)





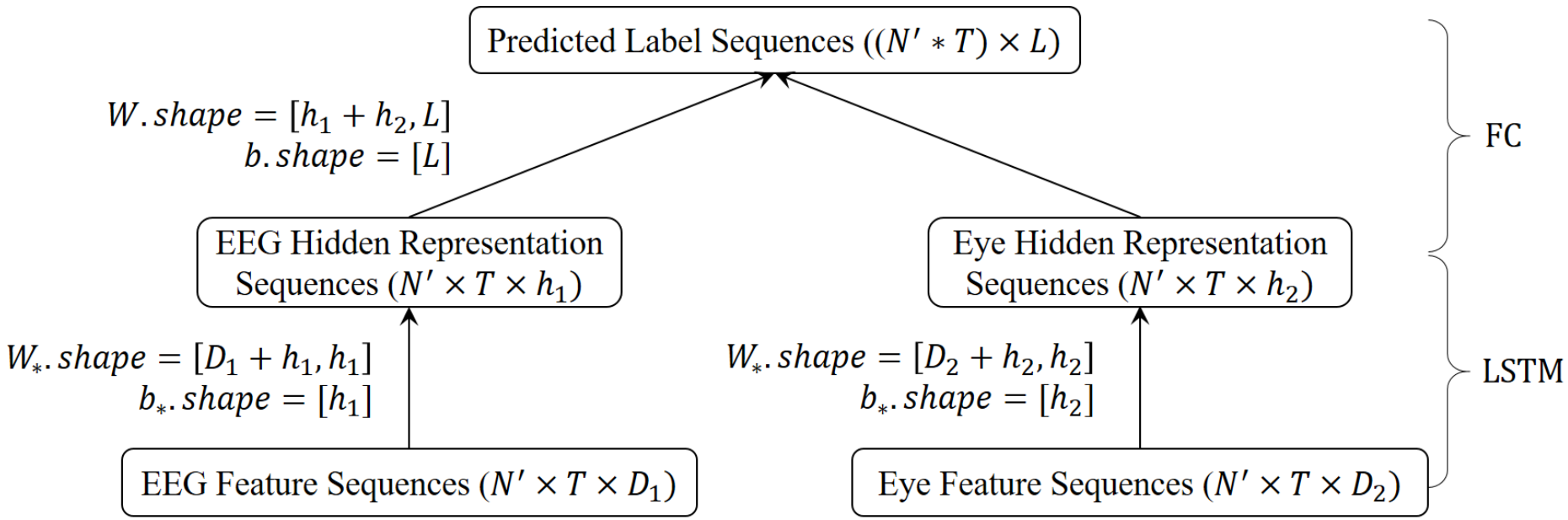
Experiment details: preprocessing

- Normalization (zero mean and unit variance)
- Data argument (random noise)
- Split the feature vectors into several feature vector sequences, such that feature vectors in the same sequence correspond to the same movie clip (to maintain continuity in temporal domain) and sorted in chronological order. (For SEED dataset, the sequences have shape 60×310 (T×D))



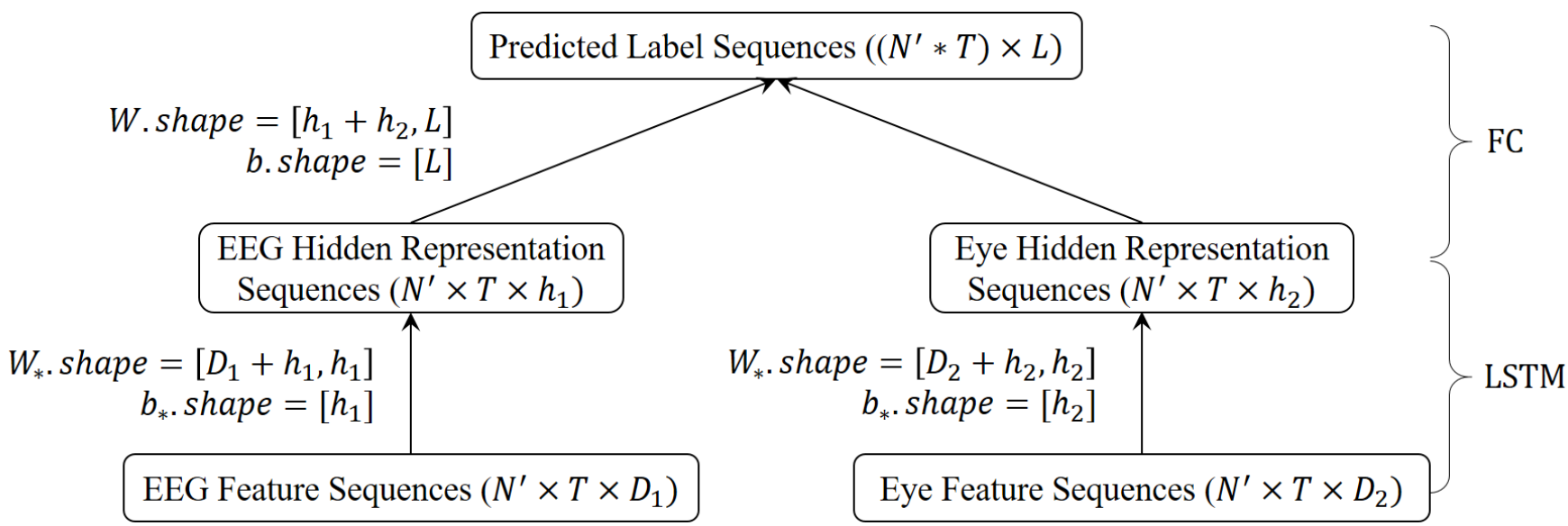
Experiment details: off-line implementation

- **After preprocessing, each sample in the dataset has shape $T \times D$, where T is the sequence length and D is the dimension number. (For SEED, $T = 60, D = 310$)**



Experiment details: off-line implementation

- After preprocessing, each sample in the dataset has shape $T \times D$, where T is the sequence length and D is the dimension number. (For SEED, $T = 60, D = 310$)
- The LSTM layer transfers the input into its hidden representation, which has shape $T \times h$, where h is the number of hidden neurons.
- The FC layer is simply matrix multiplication and addition



Experiment details: LSTM implementation

- __call__ function could be implemented in the C++ backend for all RNN cells for more efficiency.
- Naïve Pseudocode (Python+Tensorflow Style)

```
```input_tensor has shape [N',T,D]
return output_tensor with shape [N',T,h]```
output_tensor_list = []
hidden_states = tf.zeros([N',T,h])
cell_states = tf.zeros([N',T,h])
for i in range(T):
        hidden_states,cell_states = one_step(input_tensor[:,i],
                hidden_states, cell_states)
        output_tensor_list.append(hidden_states)
return tf.stack(output_tensor_list,axis=1)
```
```

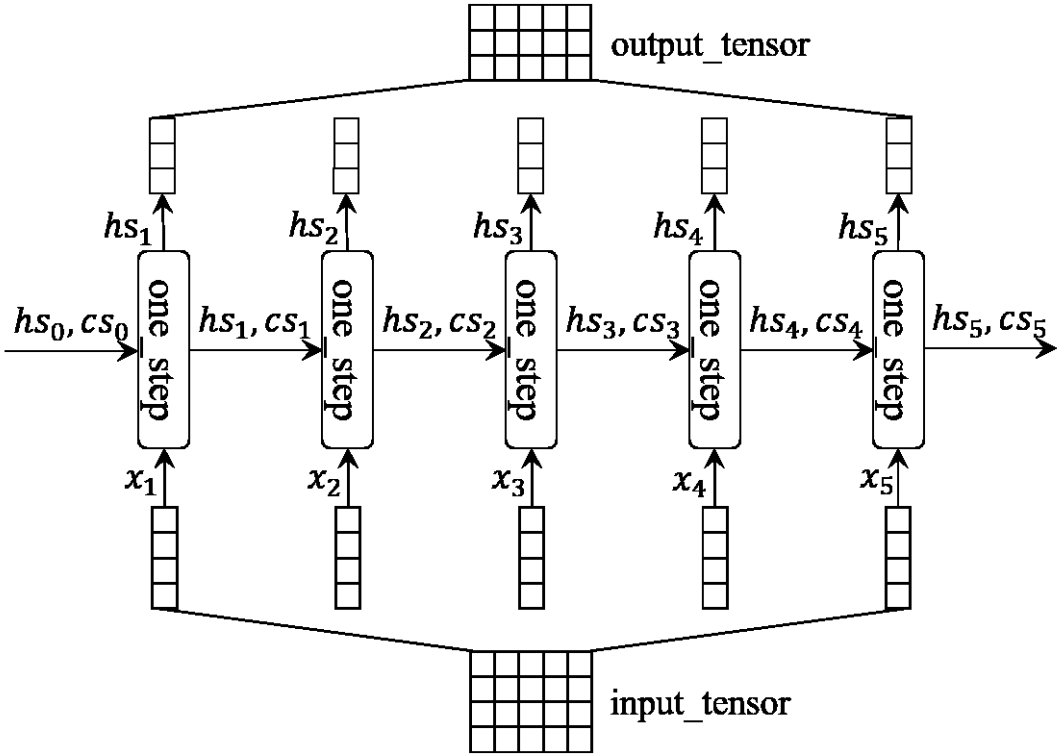
Experiment details: LSTM implementation

- one_step function differs different RNN cells.
- It calculates current states ($N' \times h$) using current input ($N' \times D$) and previous states ($N' \times h$).
- Naïve Pseudocode (Python+Tensorflow Style)

```
gates = tf.matmul(tf.concat([input_tensor,previous_hidden_states],
            axis=1) ,W)+b
f = tf.sigmoid(gates[:,:h])
i = tf.sigmoid(gates[:, h:2h])
o = tf.sigmoid(gates[:,2h:3h])
g = tf.tanh(gates[:,3h:])
cell_states = f*previous_cell_states+i*g
hidden_states = o*tf.tanh(cell_states)
```

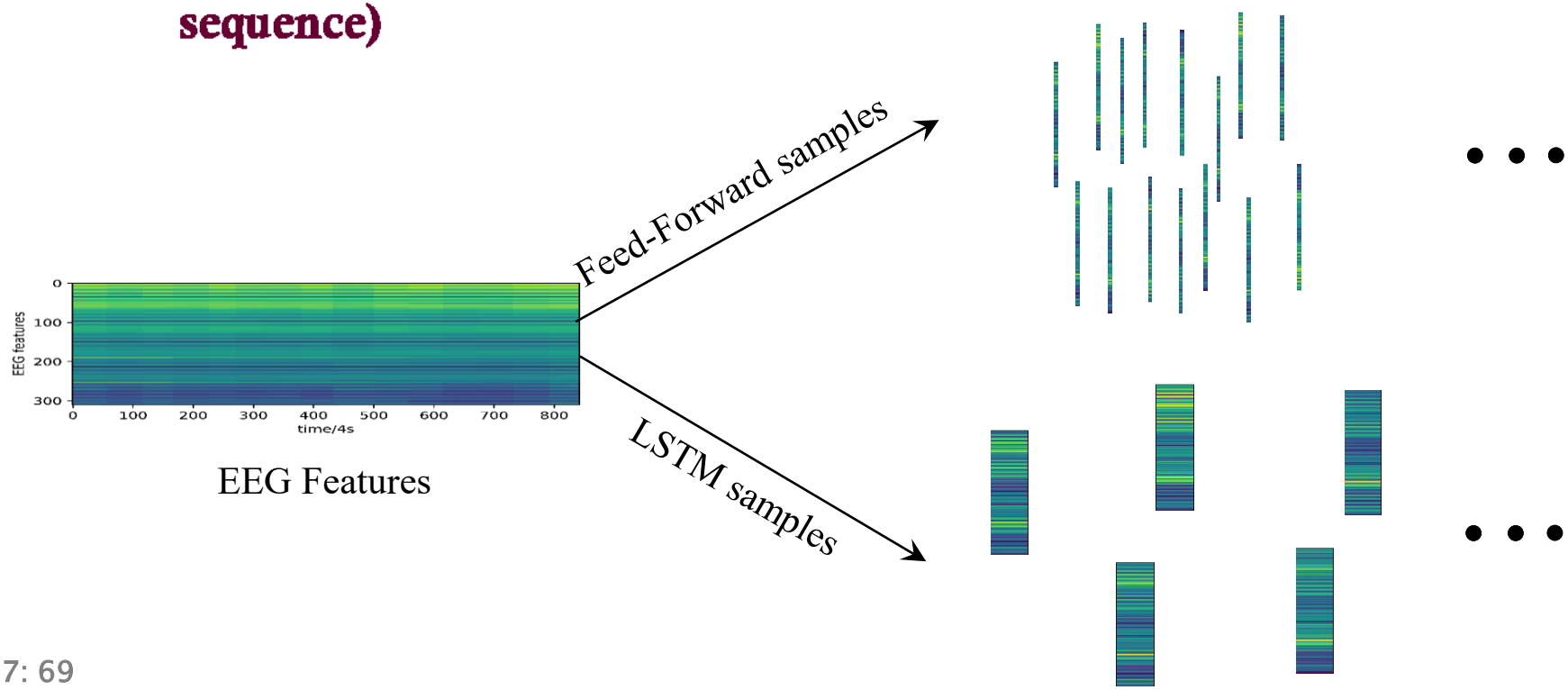
Experiment details: LSTM example

- Running example when $N' = 1, T = 5, D = 4, h = 3$
- x_i, hs_i, cs_i denotes the input feature vector, hidden states, cell states at time step i, respectively.
- All one_step functions share the weight matrix and bias.



LSTM VS Feed-Forward NN

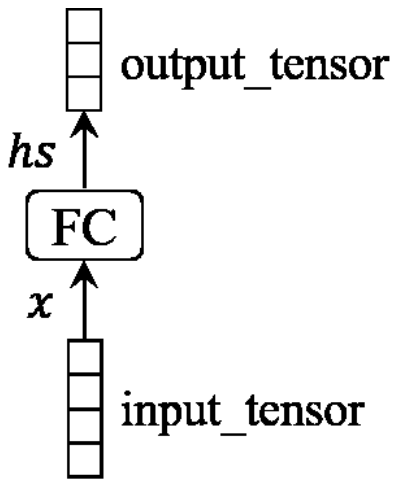
- **The shape of samples**
  - Feed-Forward: D, where D is the dimension number (Losing temporal information by ignoring the order of feature vectors)
  - LSTM: $T \times D$, where T is the sequence length, and D is the dimension number. (Preserving the chronological order in each sequence)

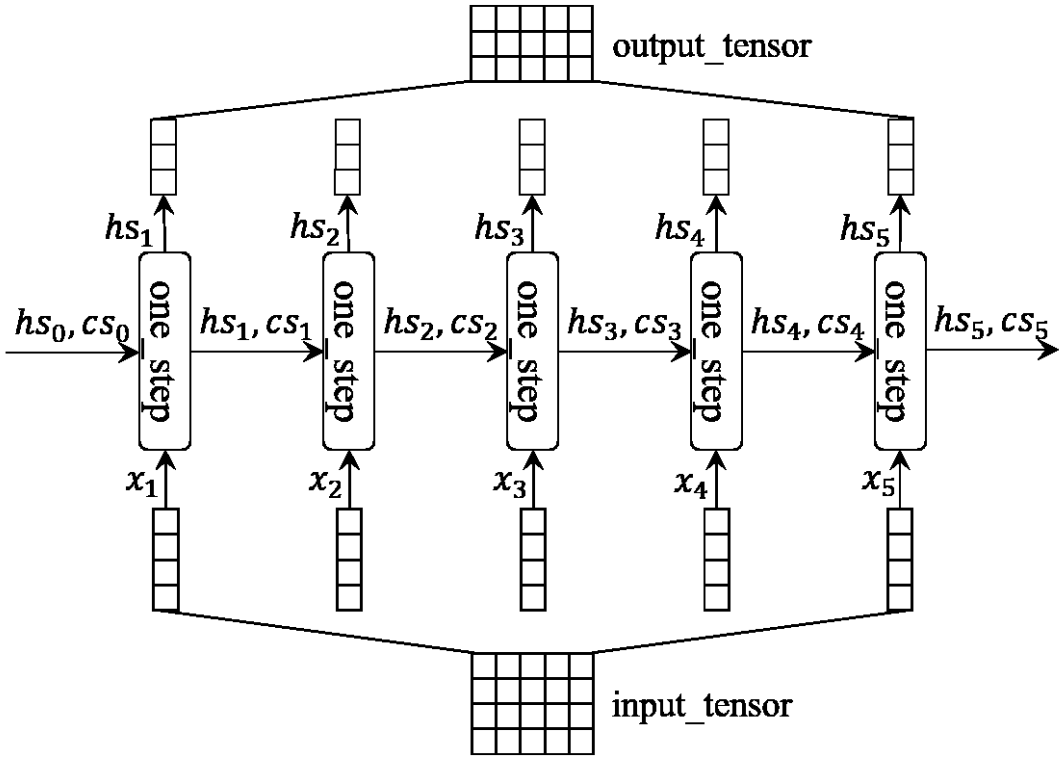


LSTM VS Feed-Forward NN

Tensor Flow

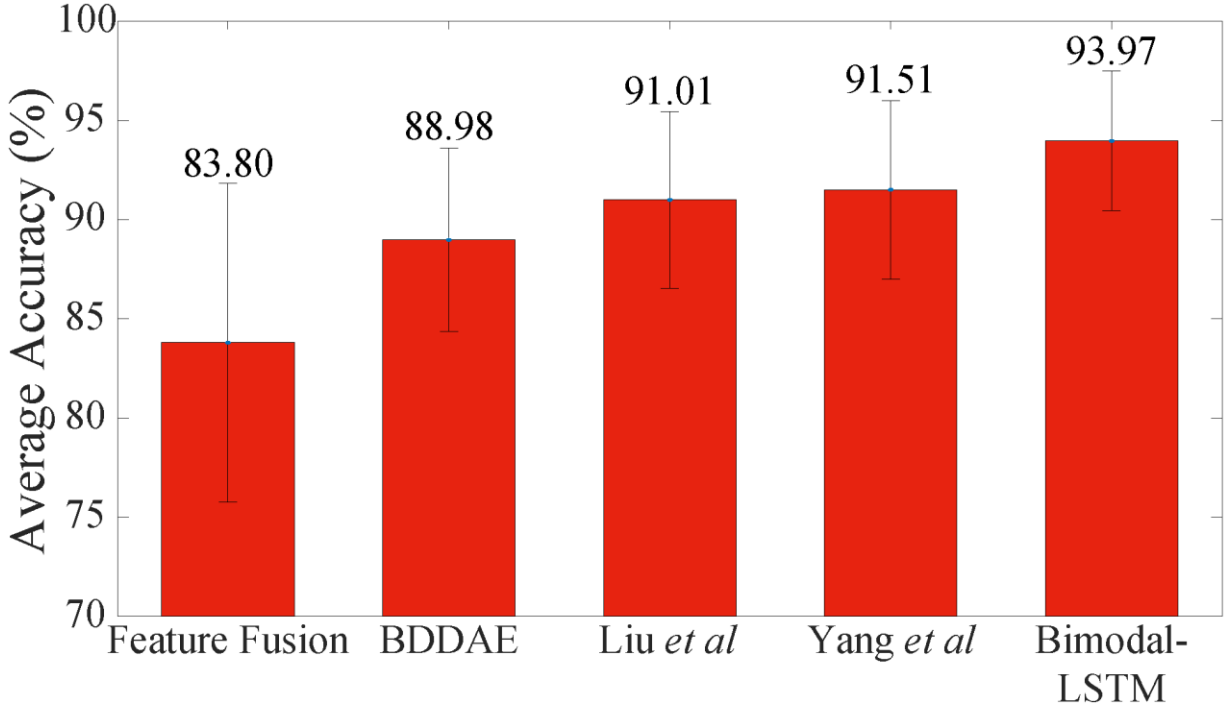
- Feed-Forward: from low layers to higher layers; no connections among different time steps.
- LSTM: states propagated through time; the processing order is the same as the chronological order.





Experimental Results – SEED dataset

- **Bimodal-LSTM achieves the best accuracy (93.97%) and the smallest standard deviation (7.03%).**



1. Liu, W., Zheng, W.L., Lu, B.L.: Emotion recognition using multimodal deep learning. In: Hirose, A., Ozawa, S., Doya, K., Ikeda, K., Lee, M., Liu, D. (eds.) ICONIP 2016. LNCS, vol. 9948, pp. 521–529. Springer, Cham (2016).
2. Yang, Y., Wu, Q.J., Zheng, W.L., Lu, B.L.: EEG-based emotion recognition using hierarchical network with subnetwork nodes. IEEE Transactions on Cognitive and Developmental Systems (2017)

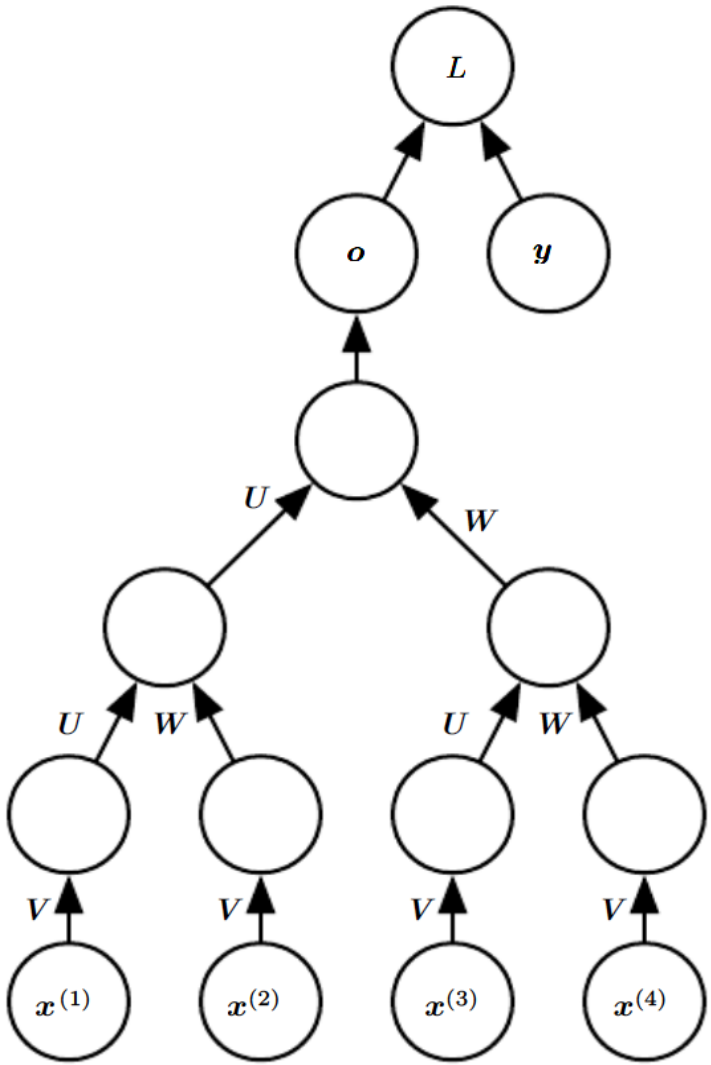
Will DL make other algorithms obsolete?

- The answer is "no" because for many simpler machine learning applications, we see far simpler algorithms work just fine for the required model accuracy.
- Different problems have different best methods. Machine learning on the whole is about applying the correct approach in the appropriate situation
- No Free Lunch Theorem states that there is no one model that works for every problem. The assumption for a great model for one problem might not hold for another problem.
- It is common in machine learning to try multiple models and find the one that works best for a particular problem.

Recursive Neural Network

- Recursive Neural Networks, like Recurrent Neural Network, can deal with variable length input.
- The primary difference is that Recursive Neural Network have the ability to model the hierarchial structure in the training dataset.
- A recursive neural network architecture is composed of a shared-weight matrix and a binary tree structure that allows the recursive network to learn varying sequence of words or parts of a image.
- Recursive neural networks use a variation of backpropagation called backpropagation through structure (BPTS)

A Recursive Neural Network



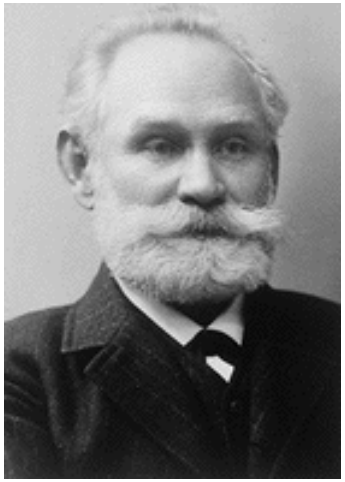
Deep Recurrent Neural Network

- The computation in most RNNs can be decomposed into three blocks for parameters and associated transformation:
  - from the input to the hidden state,
  - from the previous hidden state to the next hidden state, and
  - from the hidden state to the output.
- The experimental evidence is in agreement with the idea that we need enough depth in order to perform the required mapping.

2、Associative Learning

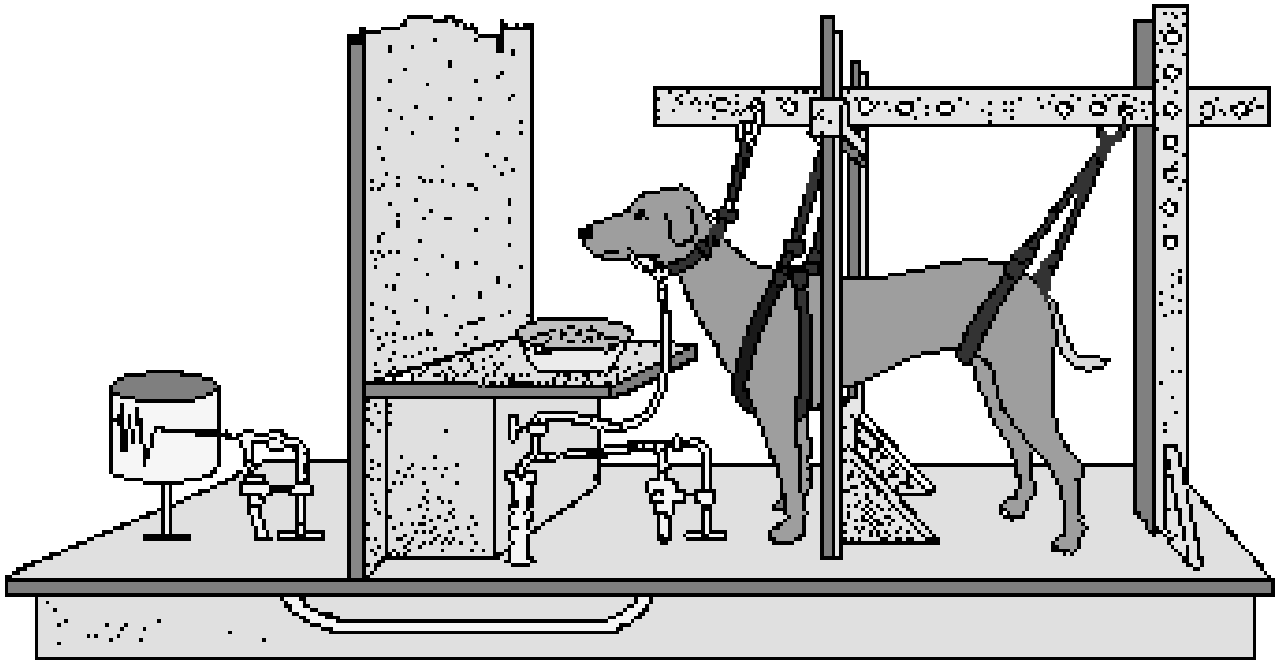
Association

- **Association: an association is any link between a system's input and output such that when a pattern *A* is presented to the system it will respond with pattern *B***
- **Stimulus: When two patterns are linked by an association, the input pattern is often referred to as the stimulus**
- **Response: Likewise, the output pattern is referred to as the response**

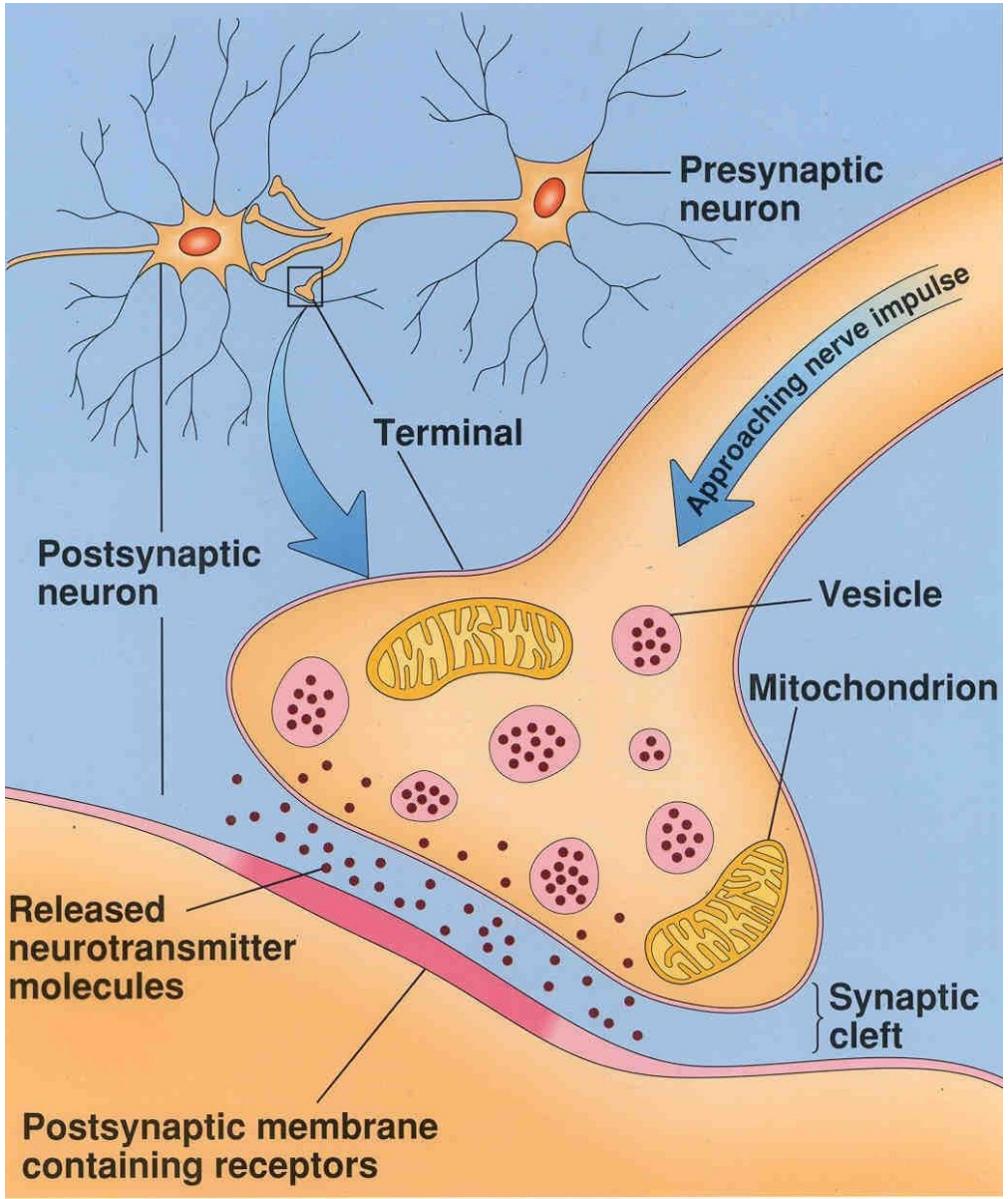


Ivan Pavlov

1849-1936



Presynaptic Neuron vs. Postsynaptic Neuron



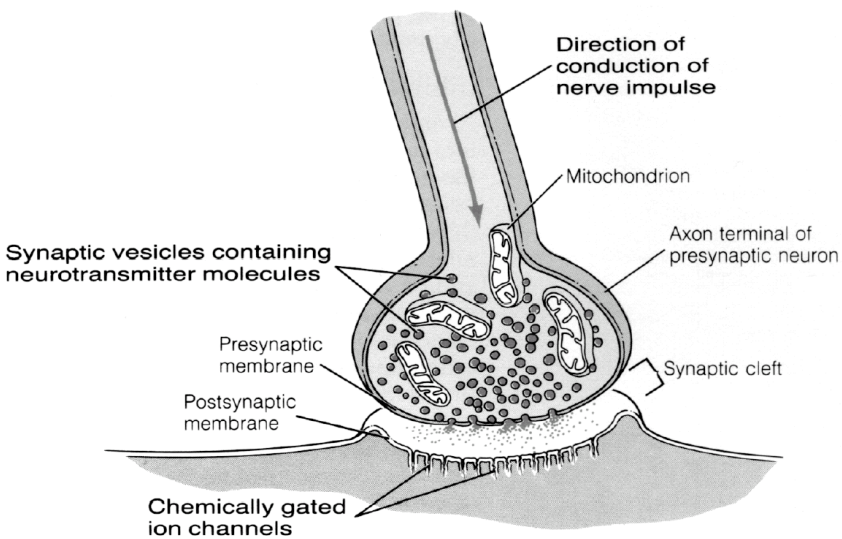
Hebbian Learning

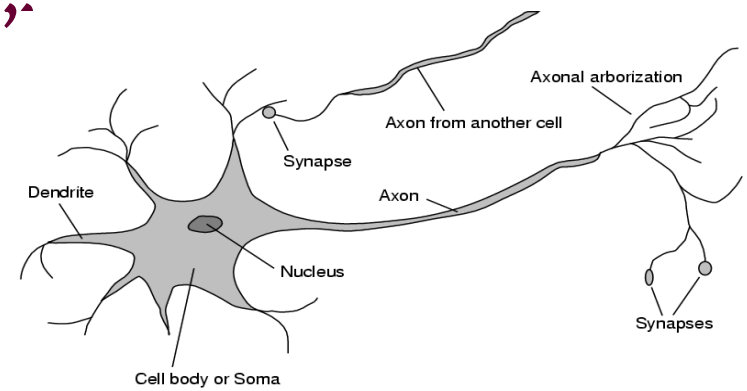
- Hebbian theory is a neuroscientific theory claiming that an increase in synaptic efficacy arises from a presynaptic cell 's repeated and persistent stimulation of a postsynaptic cell.
- It is an attempt to explain synaptic plasticity, the adaptation of brain neurons during the learning process.
- It was introduced by Donald Hebb in his 1949 book The Organization of Behavior.
- The theory is also called Hebb's rule, Hebb's postulate, and cell assembly theory

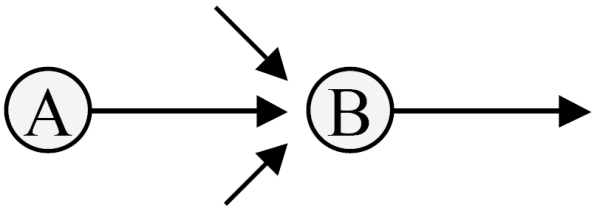
Donald Hebb

Hebbian Learning

- Hebb's rule: The Organization of Behavior (1949, p.62)
  - " When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic changes take place in one or both cells such that A's efficiency as one of the cells firing B, is increased."







Key Properties of a Hebbian Synapse

1. *Time-dependent mechanism*

   - Modification depends on the exact time of occurrence of presynaptic and postsynaptic signals.

2. *Local mechanism*

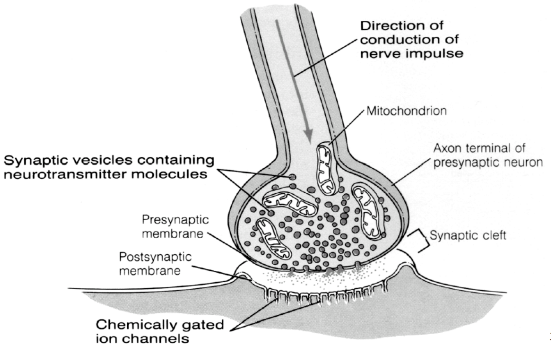
   - A Hebbian synapse uses only local information available for a neuron.

3. *Interactive mechanism*

   - Change in a Hebbian synapse depends both on presynaptic and postsynaptic signals.
   - Interaction between these signals can be either deterministic or statistical in nature.

4. *Conjunctional or correlational mechanism*

   - Correlation over time between presynaptic and postsynaptic signals is responsible for a synaptic change.



Hebb Rule

Simplified Unsupervised Form:

$$w_{ij}^{new} = w_{ij}^{old} + \alpha a_{iq} p_{jq}$$

Supervised Form (α is set to one):

$$w_{ij}^{new} = w_{ij}^{old} + t_{iq} p_{jq}$$

Matrix Form:

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{t}_q \mathbf{p}_q^T$$

Unsupervised Hebb Rule

$$w_{ij}(q) = w_{ij}(q-1) + \alpha a_i(q) p_j(q)$$

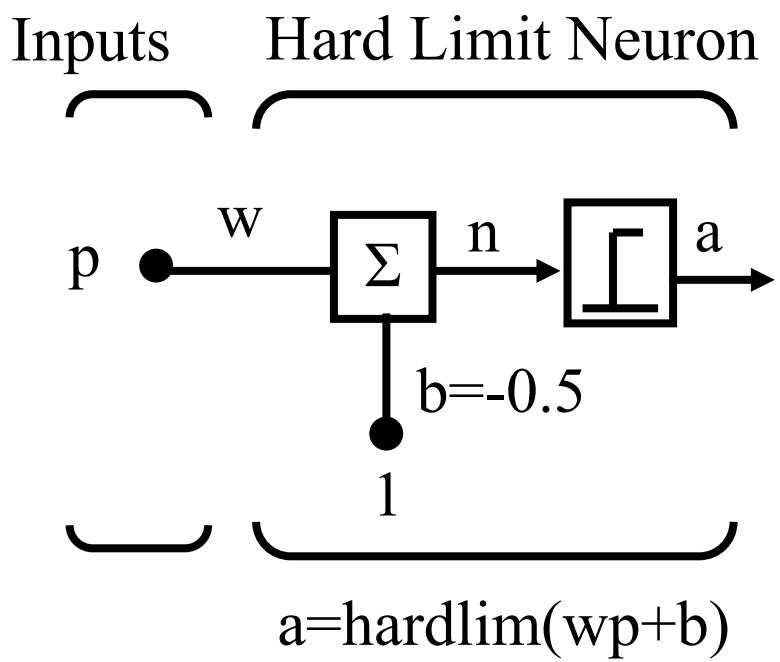
Vector Form:

$$\boxed{\mathbf{W}(q) = \mathbf{W}(q-1) + \alpha \mathbf{a}(q) \mathbf{p}^T(q)}$$

Training Sequence:

$$\mathbf{p}(1), \mathbf{p}(2), \ldots, \mathbf{p}(Q)$$

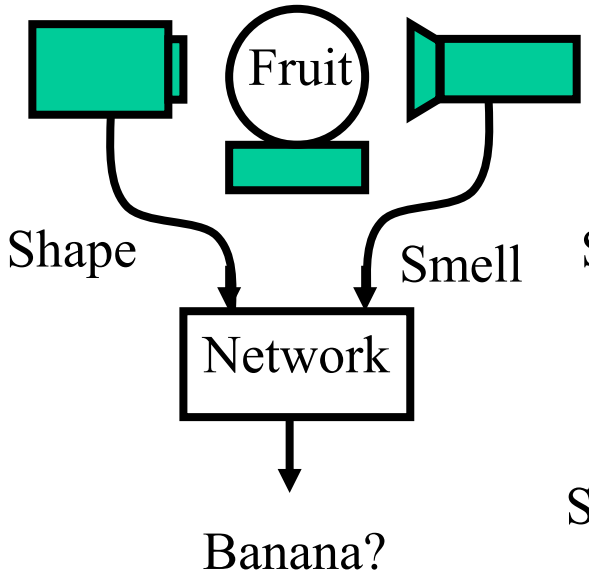
Simple Associative Network

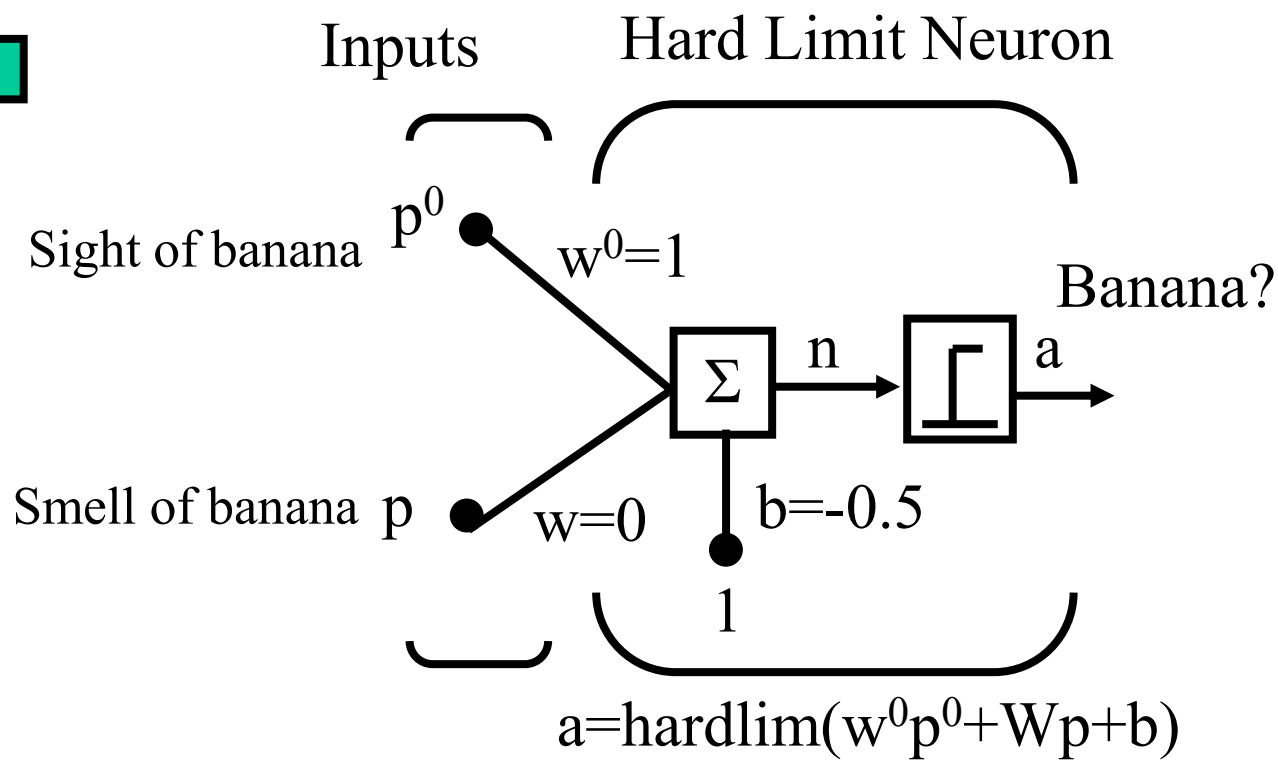


$$a = hardlim(wp + b) = hardlim(wp - 0.5)$$

$$p = \begin{cases} 1, & \text{stimulus} \\ 0, & \text{no stimulus} \end{cases} \qquad a = \begin{cases} 1, & \text{response} \\ 0, & \text{no response} \end{cases}$$

Banana Associator





Unconditioned Stimulus

$$p^0 = \begin{cases} 1, & \text{shape detected} \\ 0, & \text{shape not detected} \end{cases}$$

Conditioned Stimulus

$$p = \begin{cases} 1, & \text{smell detected} \\ 0, & \text{smell not detected} \end{cases}$$

Banana Recognition Example

Initial Weights:

$$w^0 = 1, w(0) = 0$$

Training Sequence:

$$\{p^0(1) = 0, p(1) = 1\}, \{p^0(2) = 1, p(2) = 1\}, \ldots$$

$$\alpha = 1$$

$$w(q) = w(q-1) + a(q)p(q)$$

First Iteration (sight fails):

$$\begin{aligned} a(1) &= hardlim(w^0 p^0(1) + w(0)p(1) - 0.5) \\ &= hardlim(1 \times 0 + 0 \times 1 - 0.5) = 0 \quad \text{(no response)} \end{aligned}$$

$$w(1) = w(0) + a(1)p(1) = 0 + 0 \cdot 1 = 0$$

Example

Second Iteration (sight works):

$$a(2) = hardlim(w^0 p^0(2) + w(1)p(2) - 0.5)$$
$$= hardlim(1 \times 1 + 0 \times 1 - 0.5) = 1 \quad \text{(banana)}$$

$$w(2) = w(1) + a(2)p(2) = 0 + 1 \cdot 1 = 1$$

Third Iteration (sight fails):

$$a(3) = hardlim(w^0 p^0(3) + w(2)p(3) - 0.5)$$
$$= hardlim(1 \cdot 0 + 1 \cdot 1 - 0.5) = 1 \quad \text{(banana)}$$

$$w(3) = w(2) + a(3)p(3) = 1 + 1 \cdot 1 = 2$$

Banana will now be detected if either sensor works.

Problems with Hebb Rule

- Weights can become arbitrarily large
- There is no mechanism for weights to decrease

Hebb Rule with Decay

$$\mathbf{W}(q) = \mathbf{W}(q-1) + \alpha\mathbf{a}(q)\mathbf{p}^T(q) - \underline{\gamma\mathbf{W}(q-1)}$$

$$\boxed{\mathbf{W}(q) = (1-\gamma)\mathbf{W}(q-1) + \alpha\mathbf{a}(q)\mathbf{p}^T(q)}$$

This keeps the weight matrix from growing without bound, which can be demonstrated by setting both a_i and p_j to 1:

$$w_{ij}^{max} = (1-\gamma)w_{ij}^{max} + \alpha a_i p_j$$

$$w_{ij}^{max} = (1-\gamma)w_{ij}^{max} + \alpha$$

$$\boxed{w_{ij}^{max} = \frac{\alpha}{\gamma}}$$

Example: Banana Associator

$\alpha = 1 \qquad \gamma = 0.1$

First Iteration (sight fails):

$$a(1) = hardlim(w^0 p^0(1) + w(0)p(1) - 0.5)$$
$$= hardlim(1\times 0 + 0\times 1 - 0.5) = 0 \quad \text{(no response)}$$

$$w(1) = w(0) + a(1)p(1) - 0.1w(0) = 0 + 0 \cdot 1 - 0.1(0) = 0$$

Second Iteration (sight works):

$$a(2) = hardlim(w^0 p^0(2) + w(1)p(2) - 0.5)$$
$$= hardlim(1\times 1 + 0\times 1 - 0.5) = 1 \quad \text{(banana)}$$

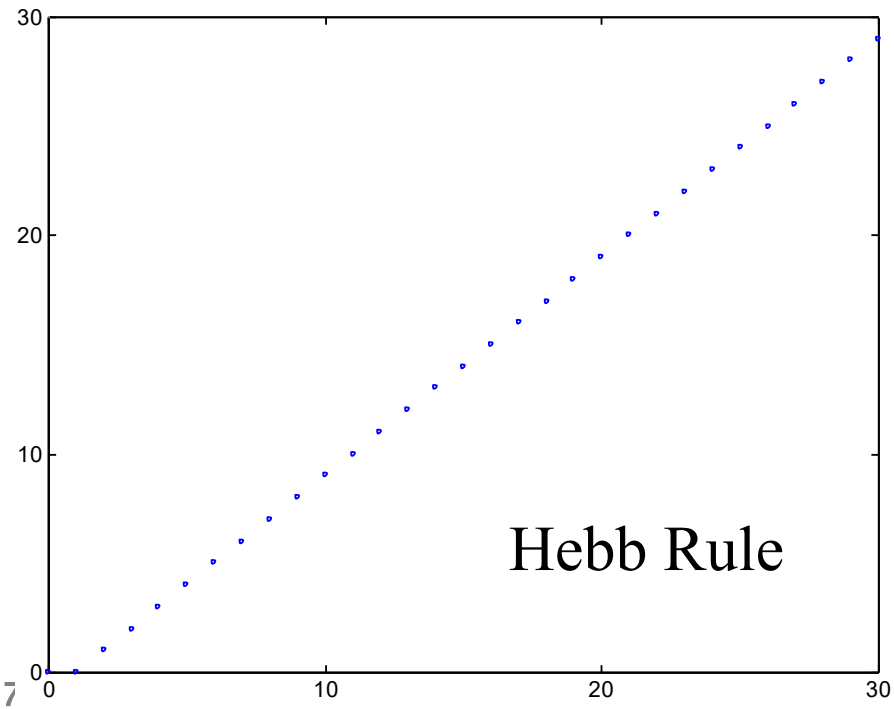
$$w(2) = w(1) + a(2)p(2) - 0.1w(1) = 0 + 1 \cdot 1 - 0.1(0) = 1$$

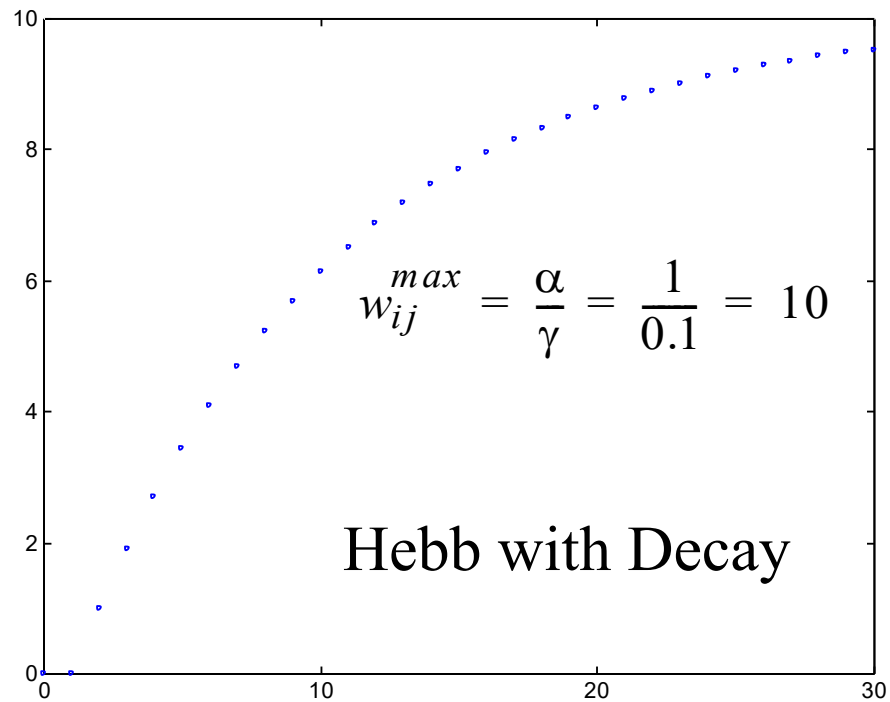
Example

Third Iteration (sight fails):

$$a(3) = hardlim(w^0 p^0(3) + w(2)p(3) - 0.5)$$
$$= hardlim(1 \cdot 0 + 1 \cdot 1 - 0.5) = 1 \quad \text{(banana)}$$

$$w(3) = w(2) + a(3)p(3) - 0.1w(3) = 1 + 1 \cdot 1 - 0.1(1) = 1.9$$





Problem of Hebb with Decay

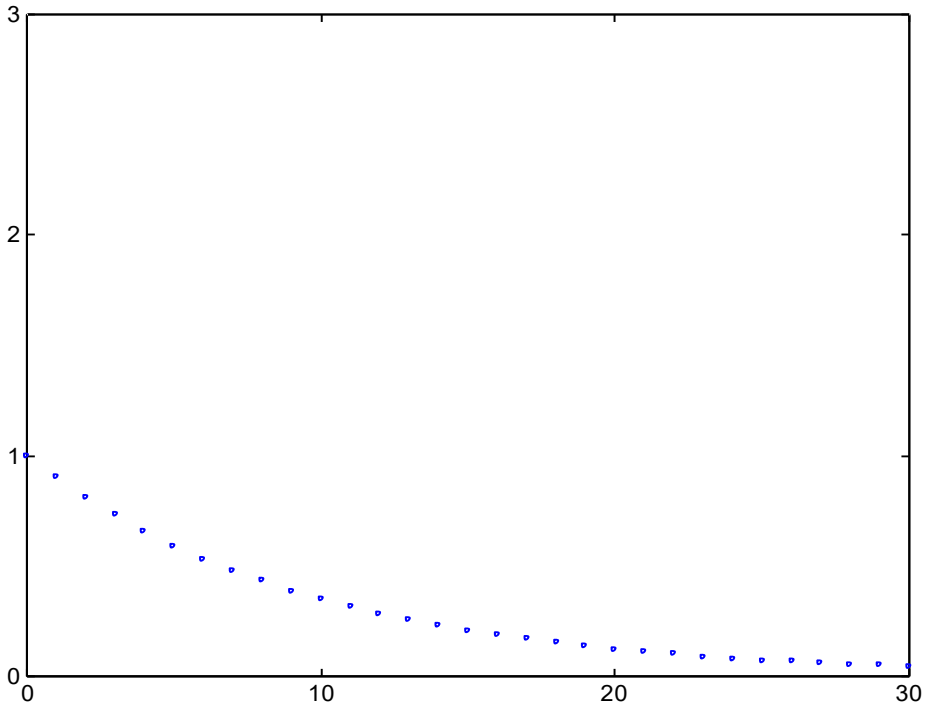
- Associations will decay away if stimuli are not occasionally presented.

If $a_i = 0$, then

$$w_{ij}(q) = (1-\gamma)w_{ij}(q-1)$$

If $\gamma = 0.1$, this becomes

$$w_{ij}(q) = (0.9)w_{ij}(\mathrm{q}-1)$$

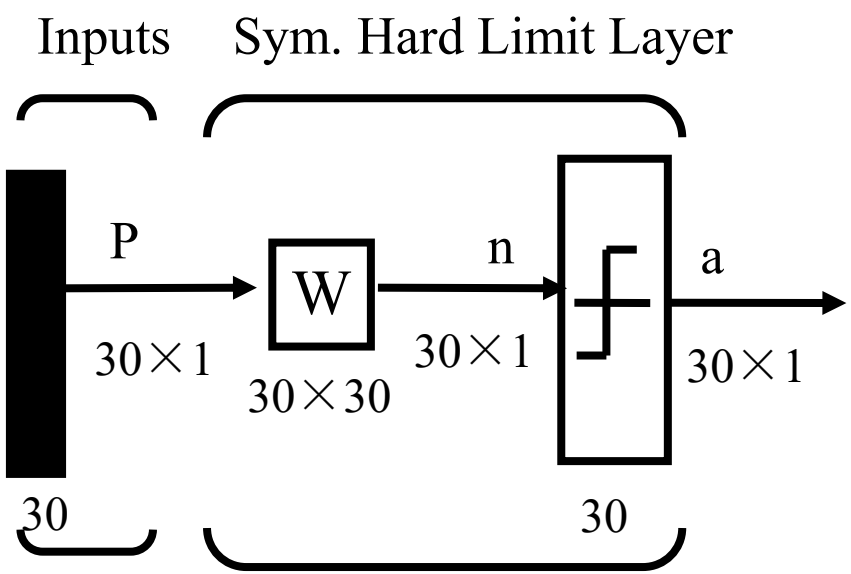


Therefore the weight decays by 10% at each iteration
where there is no stimulus.

Auto-associative Memory

$p_1,t_1 \quad p_2,t_2 \quad p_3,t_3$

$$\mathbf{p}_1 = \begin{bmatrix} -1 & 1 & 1 & 1 & 1 & -1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 & -1 & \ldots & 1 & -1 \end{bmatrix}^T$$



$$\mathbf{W} = \mathbf{p}_1\mathbf{p}_1^T + \mathbf{p}_2\mathbf{p}_2^T + \mathbf{p}_3\mathbf{p}_3^T$$

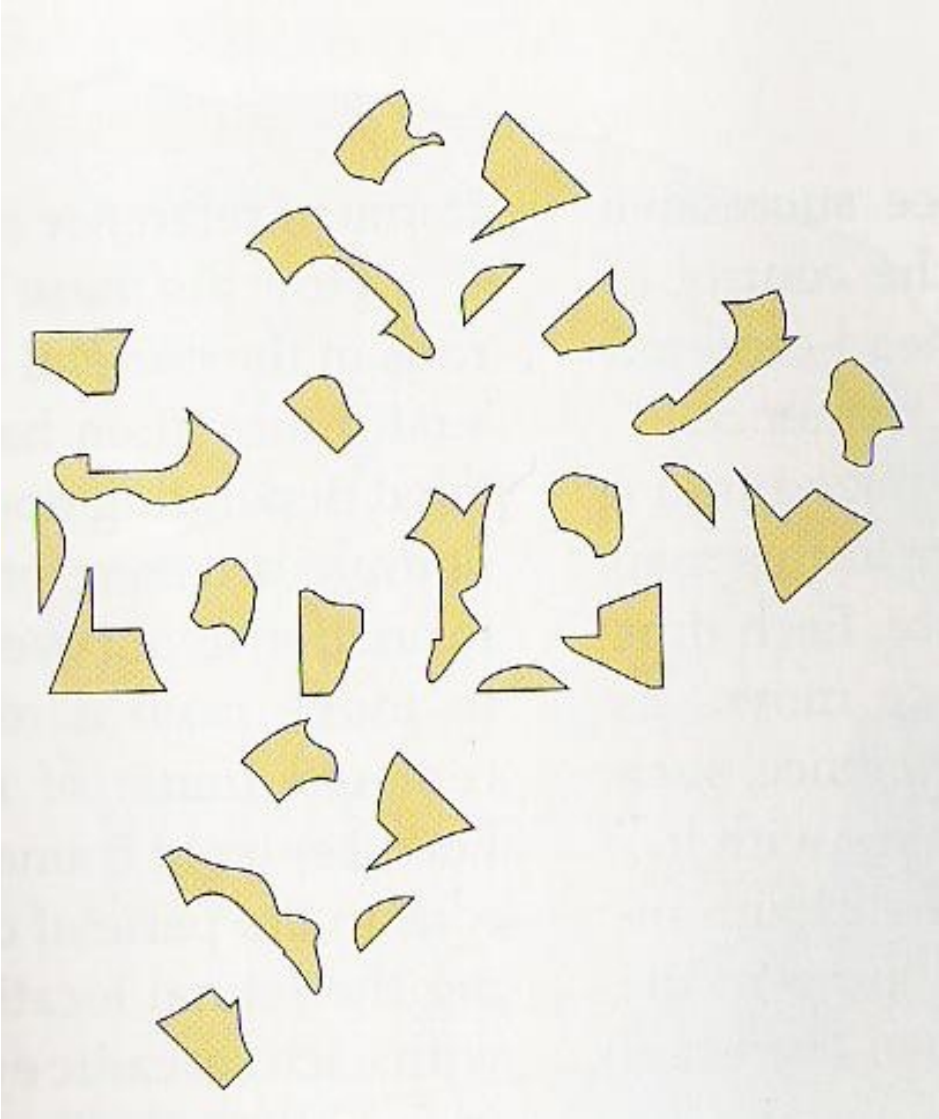
a=*hardlims*(Wp)

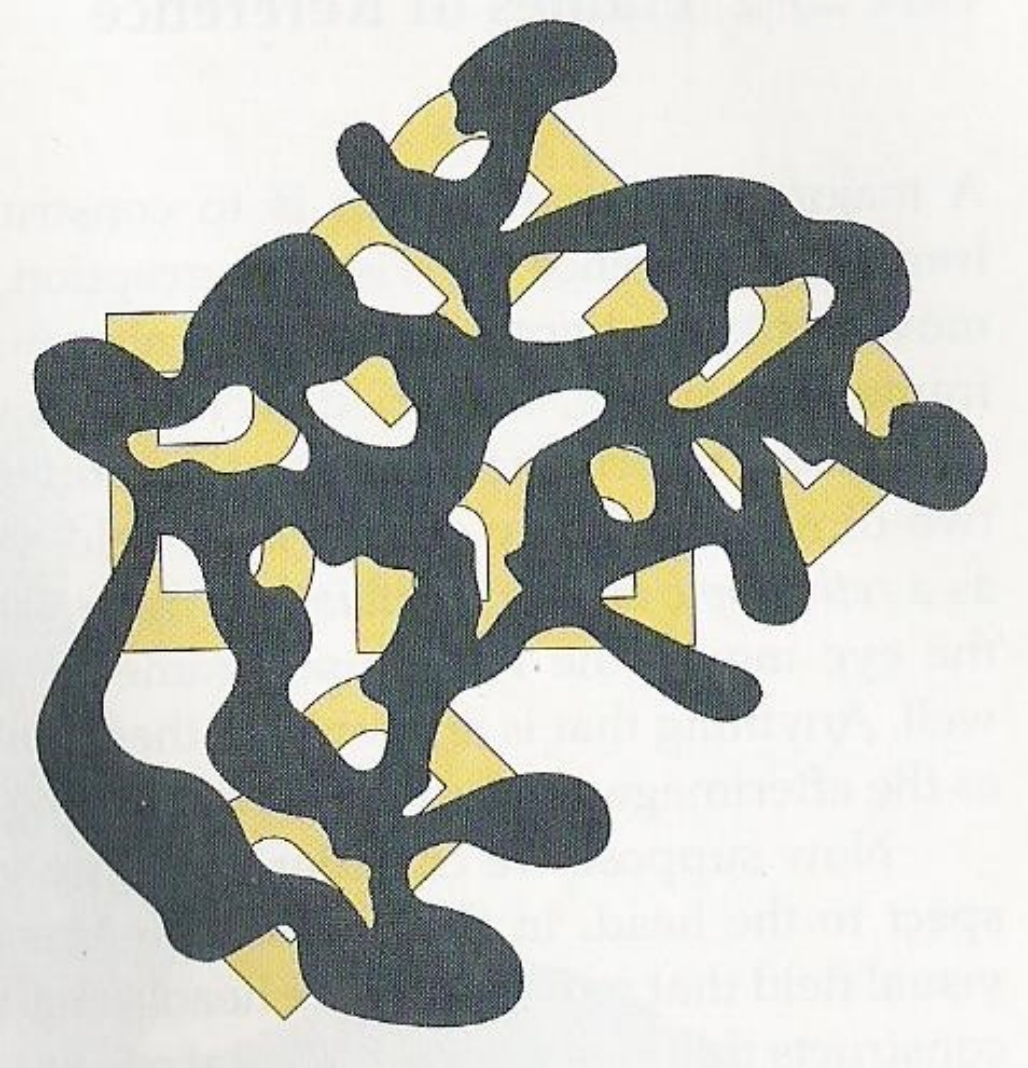
Tests

50% Occluded

67% Occluded

Noisy Patterns (7 pixels)



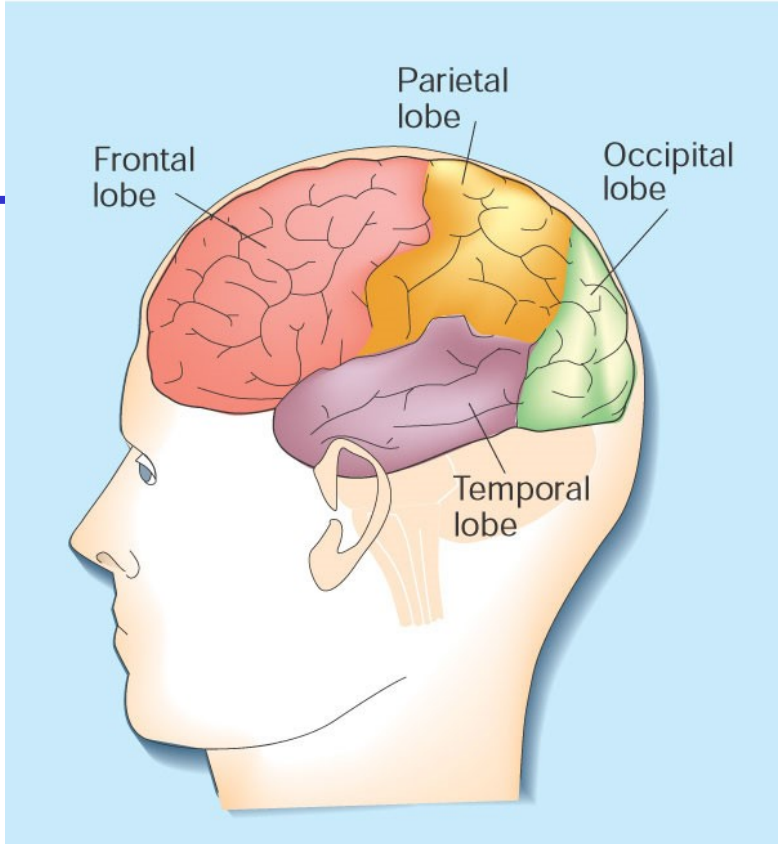


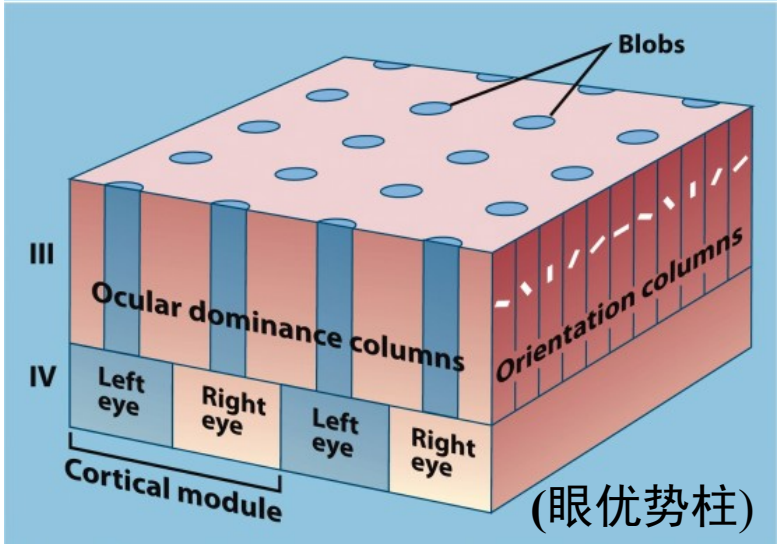
3、Competitive Networks

Biological Background

Neurons are wired topographically, nearby neurons connect to nearby neurons. In visual cortex, neurons are organized in functional columns.

- Ocular dominance columns: one region responds to one eye input
- Orientation columns: one region responds to one direction





(眼优势柱)

Turing's Observation (1952)

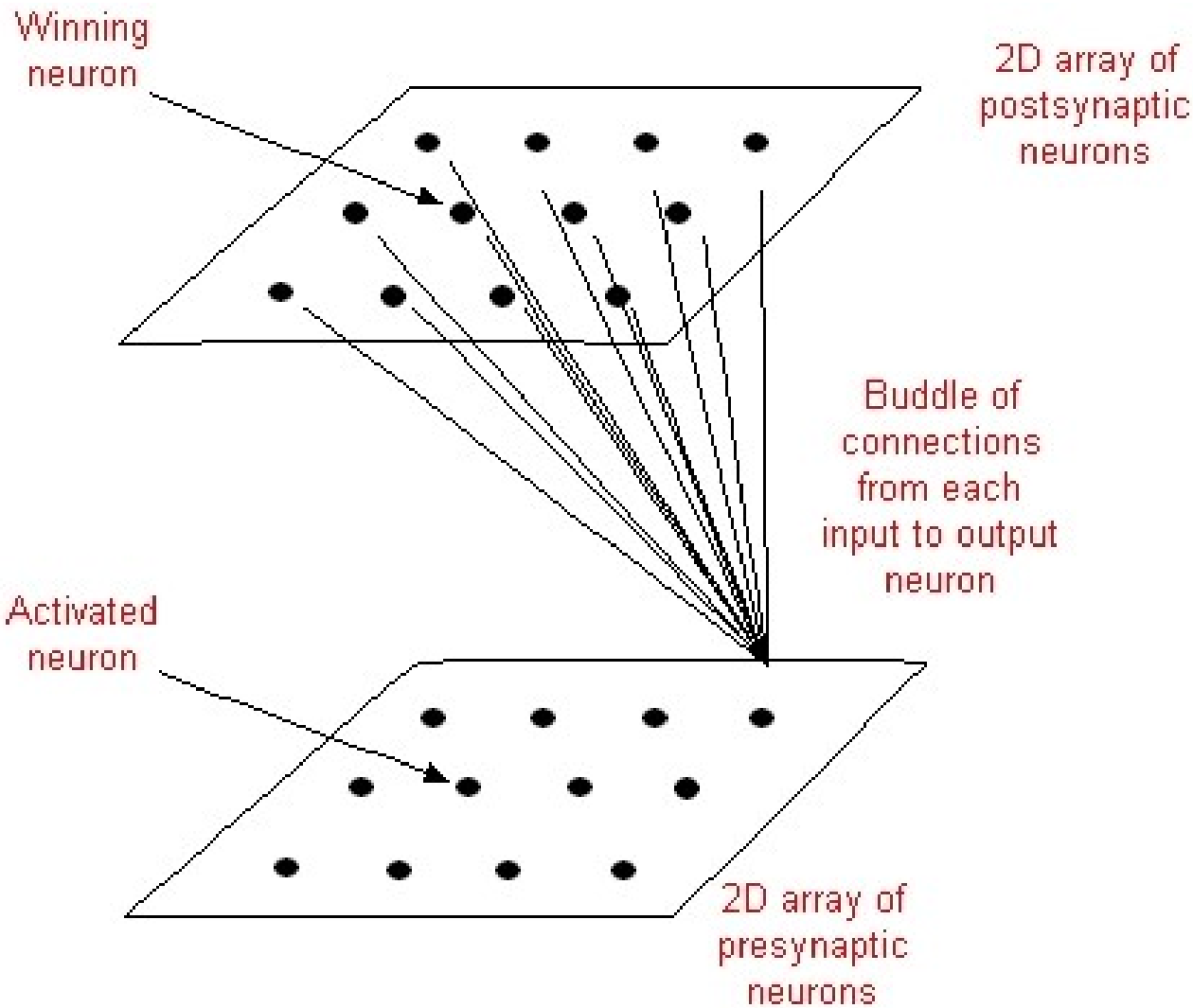
Global order can arise
from local interactions

Many originally random local interactions between neighboring neurons of a network can coalesce into states of global order and ultimately lead to coherent behavior in the form spatial patterns or temporal rhythms.

Two Self-organising Models

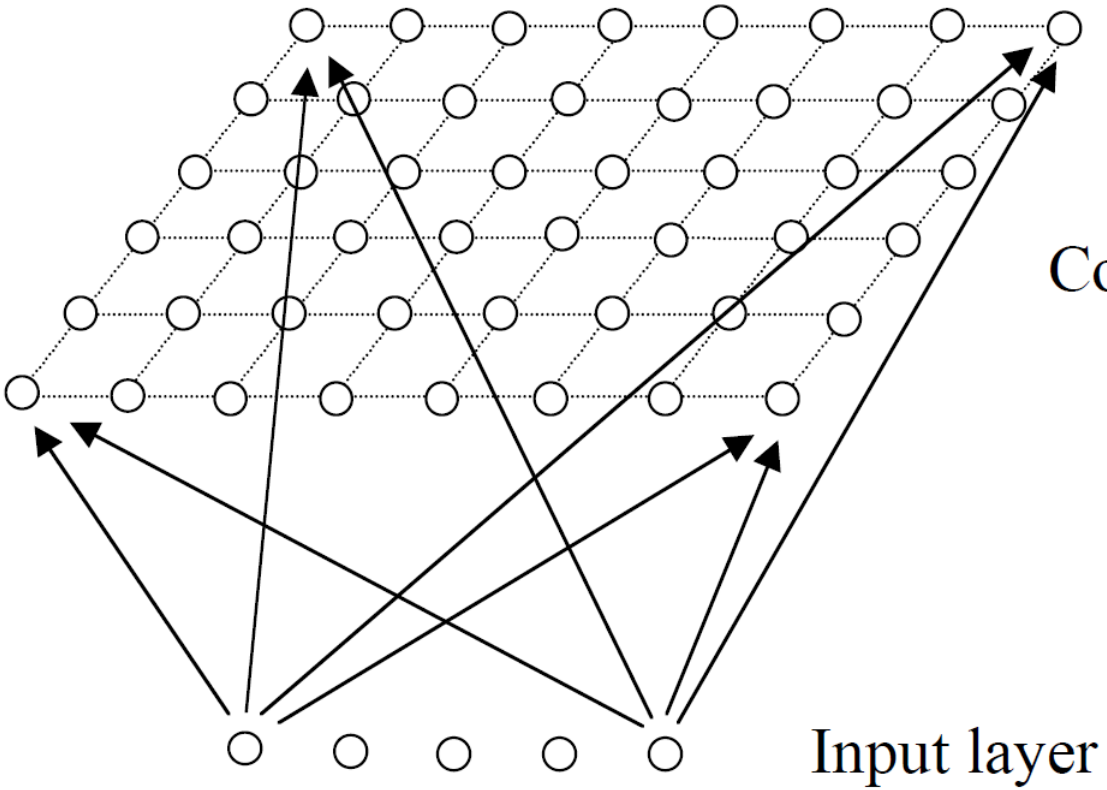
- There are two different models for the self-organising map:
  - Willshaw-von der Malsburg model;
  - Kohonen model
- In both models the output neurons are placed in a 2D lattice.
- They differ in the way input is given:
  - In the Willshaw-von der Malsburg model, the input is also a 2D lattice of equal number of neurons;
  - In the Kohonen model, there isn't any input lattice, but an array of input neurons

Willshaw-von der Malsburg model



Kohonen model

Teuvo Kohonen



What is a Self-organizing Map?

So far we have looked at networks with ***supervised training*** techniques, in which there is a target output for each input pattern, and the network learns to produce the required outputs.

We now turn to ***unsupervised training***, in which the networks learn to form their own classifications of the training data without external help. To do this we have to assume that class membership is broadly defined by the input patterns sharing ***common features***, and that the network will be able to identify those features across the range of input patterns.

One particularly interesting class of unsupervised system is based on ***competitive learning***, in which the output neurons compete amongst themselves to be activated, with the result that only one is activated at any one time. This activated neuron is called a ***winner-takes-all neuron*** or simply the ***winning neuron***. Such competition can be induced/implemented by having ***lateral inhibition connections*** (negative feedback paths) between the neurons. The result is that the neurons are forced to organise themselves. For obvious reasons, such a network is called a ***Self Organizing Map (SOM)***.

Topographic Maps

Neurobiological studies indicate that different sensory inputs (motor, visual, auditory, etc.) are mapped onto corresponding areas of the cerebral cortex in an ***orderly fashion***.

This form of map, known as a ***topographic map***, has two important properties:

1. At each stage of representation, or processing, each piece of incoming information is kept in its proper context/neighbourhood.
2. Neurons dealing with closely related pieces of information are kept close together so that they can interact via short synaptic connections.

Our interest is in building artificial topographic maps that learn through self-organization in a neurobiologically inspired manner.

We shall follow the ***principle of topographic map formation***: "The spatial location of an output neuron in a topographic map corresponds to a particular domain or feature drawn from the input space".

Setting up a Self-organizing Map

The principal goal of an SOM is to ***transform*** an incoming signal pattern of arbitrary dimension into a one or two dimensional discrete map, and to perform this transformation adaptively in a topologically ordered fashion.

We therefore set up our SOM by placing neurons at the nodes of a one or two dimensional lattice. Higher dimensional maps are also possible, but not so common.

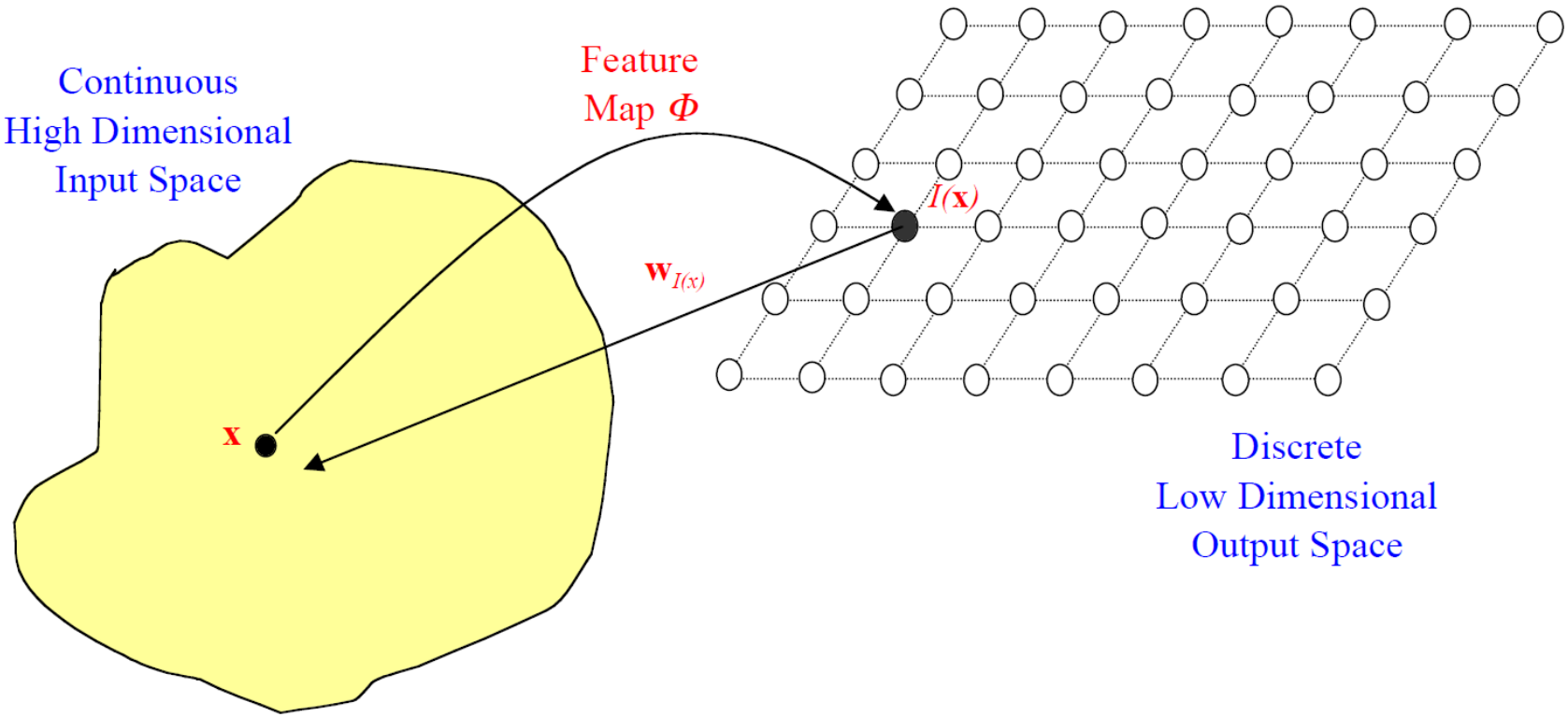
The neurons become ***selectively tuned*** to various input patterns (stimuli) or classes of input patterns during the course of the competitive learning.

The locations of the neurons so tuned (i.e. the winning neurons) become ordered and a meaningful ***coordinate system*** for the ***input features*** is created on the lattice. The SOM thus forms the required topographic map of the input patterns.

We can view this as a non-linear generalization of principal component analysis (PCA).

Organization of the Mapping

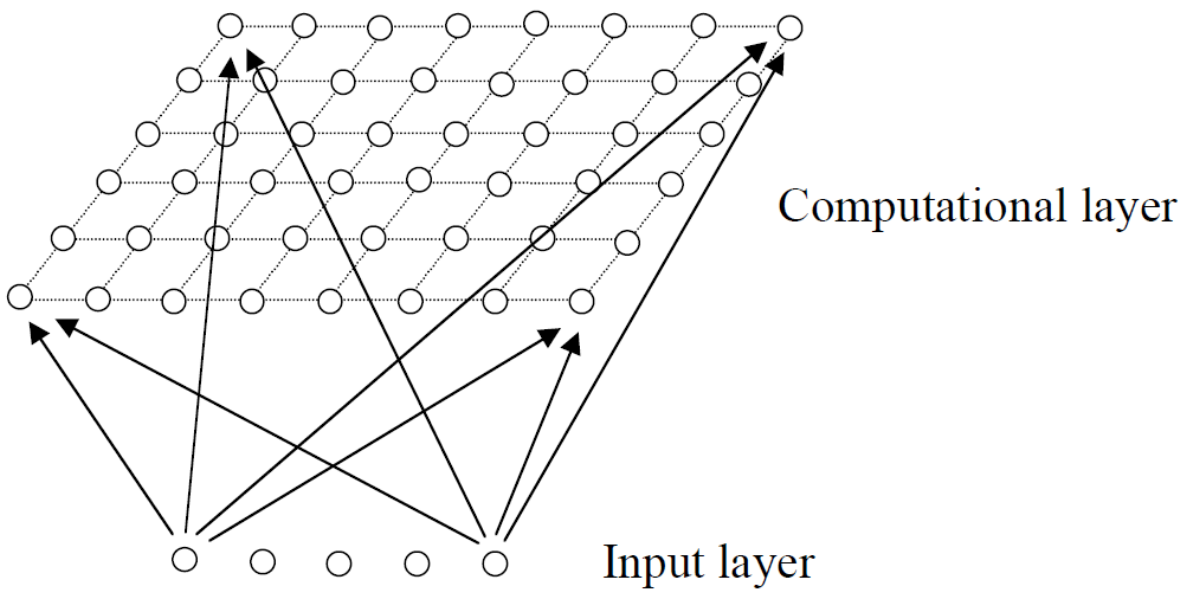
We have points $\mathbf{x}$ in the input space mapping to points $I(\mathbf{x})$ in the output space:



Each point I in the output space will map to a corresponding point $\mathbf{w}(I)$ in the input space.

Kohonen Networks

We shall concentrate on the particular kind of SOM known as a ***Kohonen Network***. This SOM has a feed-forward structure with a single computational layer arranged in rows and columns. Each neuron is fully connected to all the source nodes in the input layer:



Clearly, a one dimensional map will just have a single row (or a single column) in the computational layer.

Components of Self-Organization

The self-organization process involves four major components:

Initialization: All the connection weights are initialized with small random values.

Competition: For each input pattern, the neurons compute their respective values of a *discriminant function* which provides the basis for competition. The particular neuron with the smallest value of the discriminant function is declared the winner.

Cooperation: The winning neuron determines the spatial location of a topological neighbourhood of excited neurons, thereby providing the basis for cooperation among neighbouring neurons.

Adaptation: The excited neurons decrease their individual values of the discriminant function in relation to the input pattern through suitable adjustment of the associated connection weights, such that the response of the winning neuron to the subsequent application of a similar input pattern is enhanced.

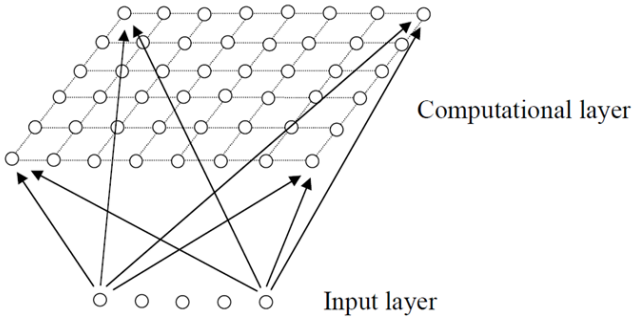
The Competitive Process

If the input space is D dimensional (i.e. there are D input units) we can write the input patterns as $\mathbf{x} = \{x_i : i = 1, \ldots, D\}$ and the connection weights between the input units i and the neurons j in the computation layer can be written $\mathbf{w}_j = \{w_{ji} : j = 1, \ldots, N; i = 1, \ldots, D\}$ where N is the total number of neurons.

We can then define our ***discriminant function*** to be the squared Euclidean distance between the input vector $\mathbf{x}$ and the weight vector $\mathbf{w}_j$ for each neuron j

$$d_j(\mathbf{x}) = \sum_{i=1}^{D} (x_i - w_{ji})^2$$

In other words, the neuron whose weight vector comes closest to the input vector (i.e. is most similar to it) is declared the winner.



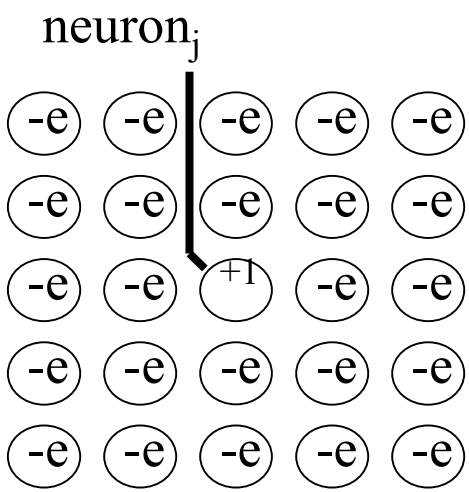
Competitive Layers in Biology

On-Center/Off-Surround Connections for Competition

Weights in the competitive layer of the Hamming network:

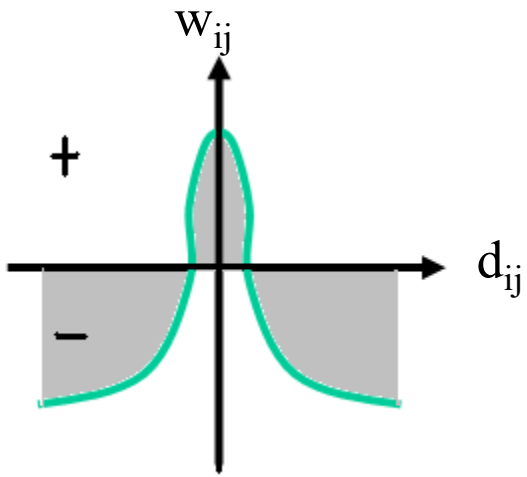
$$w_{i,j} = \begin{cases} 1, & \text{if } i = j \\ -\varepsilon, & \text{if } i \neq j \end{cases}$$

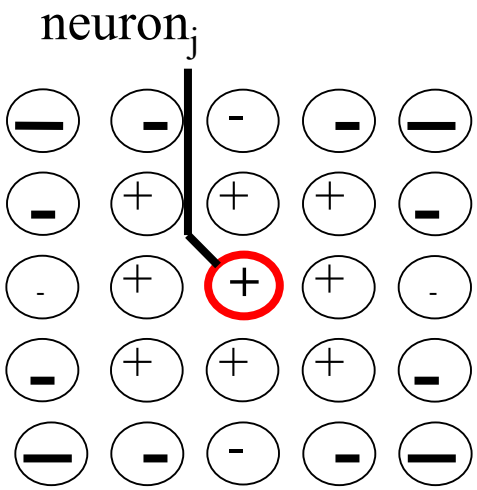
Weights assigned based on distance:



$$w_{i,j} = \begin{cases} 1, & \text{if } d_{i,j} = 0 \\ -\varepsilon, & \text{if } d_{i,j} > 0 \end{cases}$$

Mexican-Hat Function





Feature Maps

Update weight vectors in a neighborhood of the winning neuron.

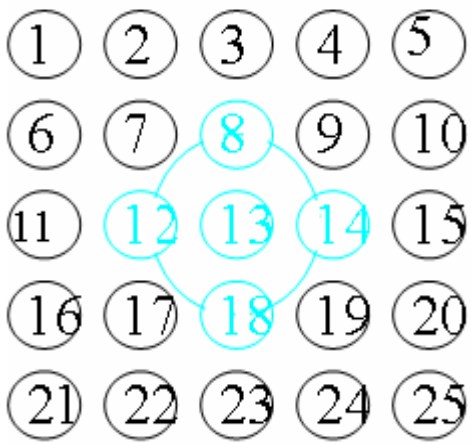
$$ {}_i\mathbf{w}(q) = {}_i\mathbf{w}(q-1) + \alpha(\mathbf{p}(q) - {}_i\mathbf{w}(q-1)) $$

$$ {}_i\mathbf{w}(q) = (1-\alpha){}_i\mathbf{w}(q-1) + \alpha\mathbf{p}(q) \qquad i \in N_{i*}(d) $$

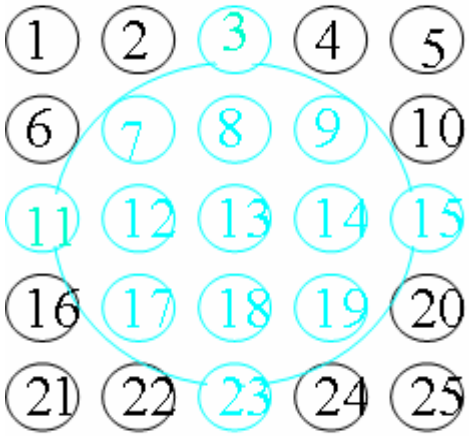
$$ N_i(d) = \{j, \mathrm{d}_{i,j} \le d\} $$

$$ N_{13}(1) = \{8, 12, 13, 14, 18\} $$

$$ N_{13}(2) = \{3, 7, 8, 9, 11, 12, 13, 14, 15, 17, 18, 19, 23\} $$



$N_{13}(1)$



$N_{13}(2)$

The Cooperative Process

In neurobiological studies we find that there is ***lateral interaction*** within a set of excited neurons. When one neuron fires, its closest neighbours tend to get excited more than those further away. There is a ***topological neighbourhood*** that decays with distance.

We want to define a similar topological neighbourhood for the neurons in our SOM. If S_{ij} is the lateral distance between neurons i and j on the grid of neurons, we take

$$T_{j,I(\mathbf{x})} = \exp(-S^2_{j,I(\mathbf{x})} / 2\sigma^2)$$

as our topological neighbourhood, where $I(\mathbf{x})$ is the index of the winning neuron. This has several important properties: it is maximal at the winning neuron, it is symmetrical about that neuron, it decreases monotonically to zero as the distance goes to infinity, and it is translation invariant (i.e. independent of the location of the winning neuron)

A special feature of the SOM is that the size σ of the neighbourhood needs to decrease with time. A popular time dependence is an exponential decay: $\sigma(t) = \sigma_0 \exp(-t / \tau_\sigma)$.

The Adaptive Process

Clearly our SOM must involve some kind of adaptive, or learning, process by which the outputs become self-organised and the ***feature map*** between inputs and outputs is formed.

The point of the topographic neighbourhood is that not only the winning neuron gets its weights updated, but its neighbours will have their weights updated as well, although by not as much as the winner itself. In practice, the appropriate weight update equation is

$$\Delta w_{ji} = \eta(t) \,.\, T_{j,I(\mathbf{x})}(t) \,.\, (x_i - w_{ji})$$

in which we have a time (epoch) t dependent learning rate $\eta(t) = \eta_0 \exp(-t/\tau_\eta)$, and the updates are applied for all the training patterns $\mathbf{x}$ over many epochs.

The effect of each learning weight update is to move the weight vectors $\mathbf{w}_i$ of the winning neuron and its neighbours towards the input vector $\mathbf{x}$. Repeated presentations of the training data thus leads to topological ordering.

Ordering and Convergence

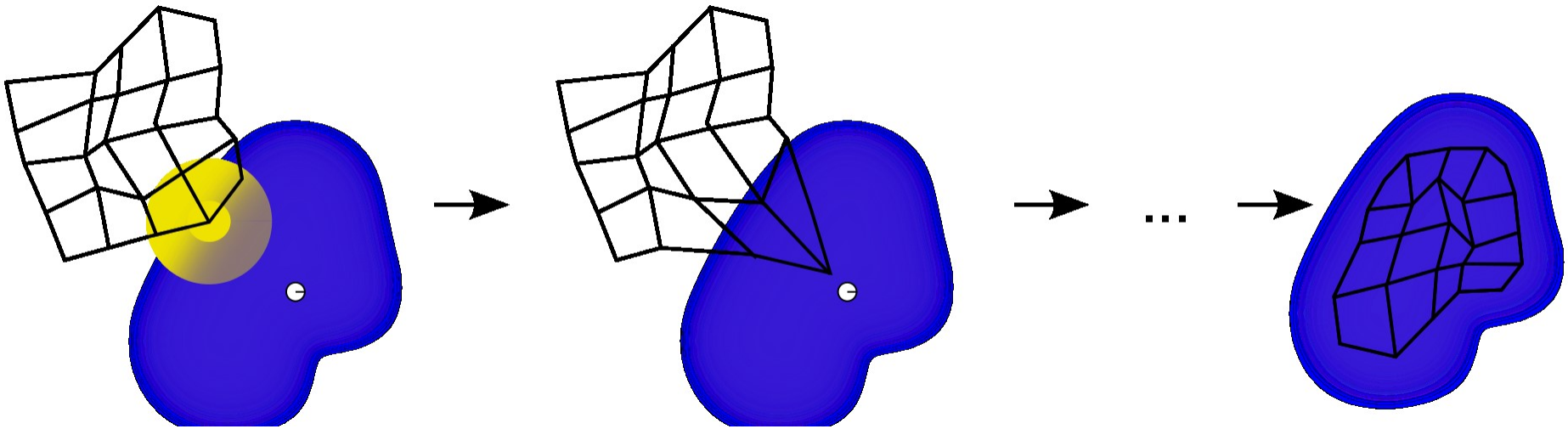
Provided the parameters (σ_0, τ_σ, η_0, τ_η) are selected properly, we can start from an initial state of complete disorder, and the SOM algorithm will gradually lead to an organized representation of activation patterns drawn from the input space. (However, it is possible to end up in a ***metastable state*** in which the feature map has a topological defect.)

There are two identifiable phases of this adaptive process:

1. **Ordering or self-organizing phase** – during which the topological ordering of the weight vectors takes place. Typically this will take as many as 1000 iterations of the SOM algorithm, and careful consideration needs to be given to the choice of neighbourhood and learning rate parameters.

2. **Convergence phase** – during which the feature map is fine tuned and comes to provide an accurate statistical quantification of the input space. Typically the number of iterations in this phase will be at least 500 times the number of neurons in the network, and again the parameters must be chosen carefully.

Visualizing the Self-organization Process



An illustration of the training of a self-organizing map. The blue blob is the distribution of the training data, and the small white disc is the current training datum drawn from that distribution. At first (left) the SOM nodes are arbitrarily positioned in the data space. The node (highlighted in yellow) which is nearest to the training datum is selected. It is moved towards the training datum, as (to a lesser extent) are its neighbors on the grid. After many iterations the grid tends to approximate the data distribution (right).

Overview of the SOM Algorithm

We have a spatially ***continuous input space***, in which our input vectors live. The aim is to map from this to a low dimensional spatially ***discrete output space***, the topology of which is formed by arranging a set of neurons in a grid. Our SOM provides such a non-linear transformation called a ***feature map***.

The stages of the SOM algorithm can be summarised as follows:

1. ***Initialization*** – Choose random values for the initial weight vectors $\mathbf{w}_j$.
2. ***Sampling*** – Draw a sample training input vector $\mathbf{x}$ from the input space.
3. ***Matching*** – Find the winning neuron $I(\mathbf{x})$ with weight vector closest to input vector.
4. ***Updating*** – Apply the weight update equation $\Delta w_{ji} = \eta(t)\ T_{j,I(\mathbf{x})}(t)\ (x_i - w_{ji})$.
5. ***Continuation*** – keep returning to step 2 until the feature map stops changing.

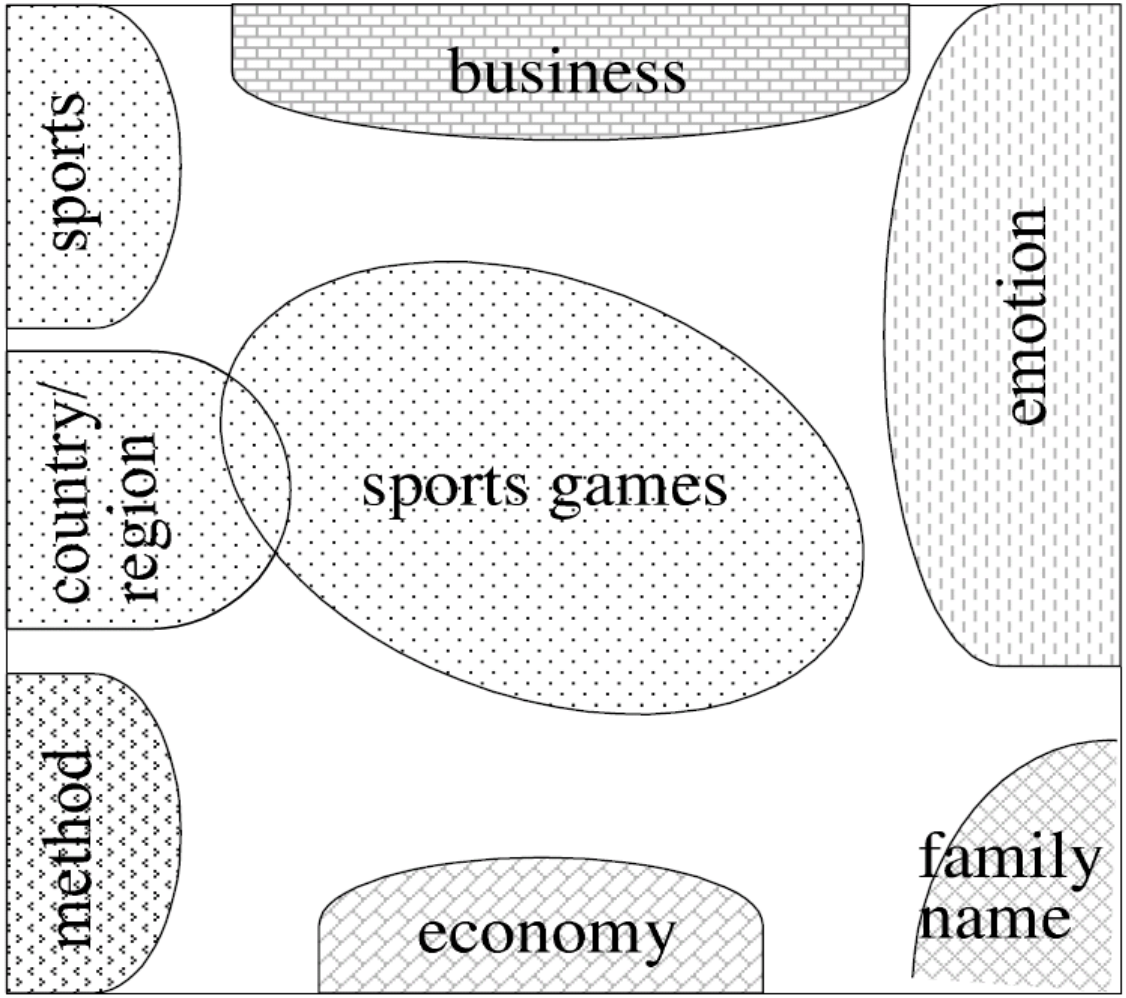
Next lecture we shall explore the properties of the resulting feature map and look at some simple examples of its application.

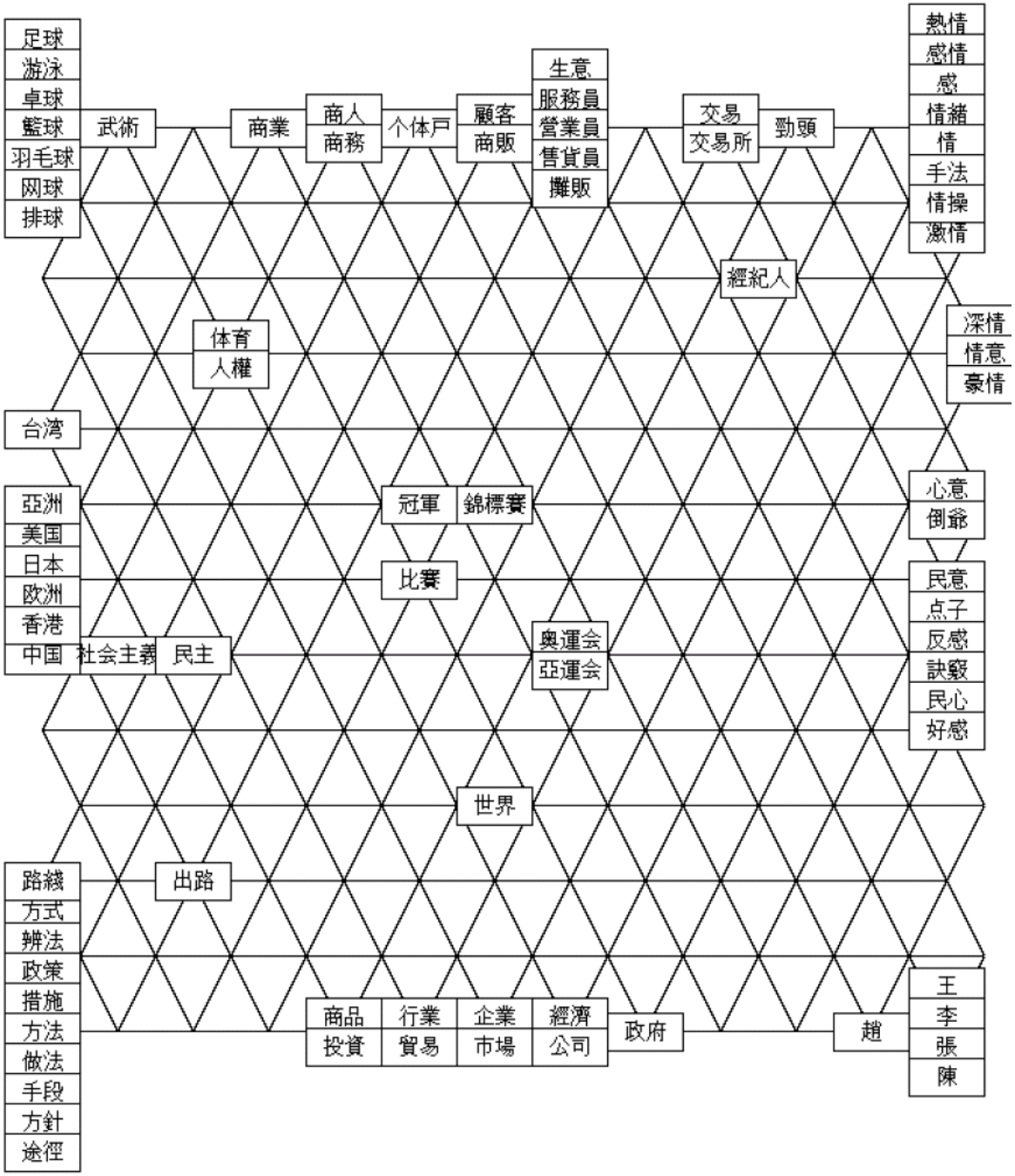
Application of SOM to NLP

Self-Organizing Chinese and Japanese Semantic Maps

Qing Ma†, Min Zhang‡, Masaki Murata†, Ming Zhou§, Hitoshi Isahara†
†Communications Research Laboratory, Kyoto 619-0289, Japan
‡State Key Lab of Intelligent Tech. and Sys, Tsinghua University, Beijing 100084, China
§Microsoft Research Asia, Beijing 100080, China
E-mail: qma@crl.go.jp

Self-Organizing Chinese




足球
游泳
卓球
籃球
羽毛球
网球
排球
武術
商業
商人
商務
个体戸
顧客
商販
生意
服務員
營業員
售貨員
攤販
交易
交易所
勁頭
熱情
感情
感
情緒
情
手法
情操
激情
經紀人
体育
人權
深情
情意
豪情
台湾
亞洲
美国
日本
欧洲
香港
中国
社会主義
民主
冠軍
錦標賽
比賽
奥運会
亞運会
心意
倒爺
民意
点子
反感
訣竅
民心
好感
世界
路綫
方式
辨法
政策
措施
方法
做法
手段
方針
途徑
出路
商品
投資
行業
貿易
企業
市場
經濟
公司
政府
趙
王
李
張
陳