Machine Learning

Volker Roth

Department of Mathematics & Computer Science
University of Basel

Section 5

Neural Networks

Subsection 1

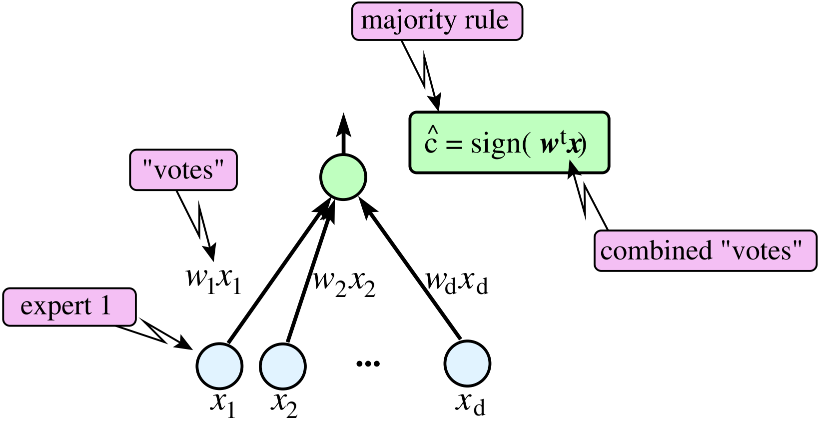
Feed-forward Neural Networks

Linear classifier

We can understand the simple **linear classifier**

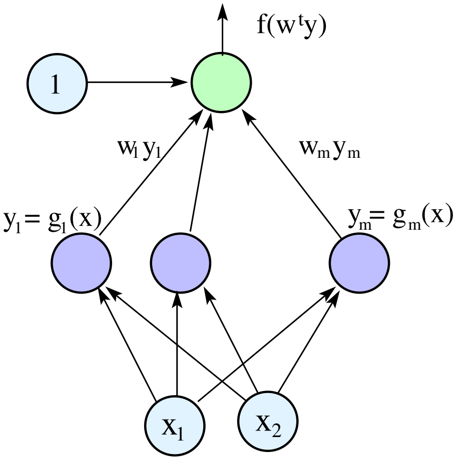
$$\hat{c} = \mathrm{sign}(\boldsymbol{w}^t \boldsymbol{x}) = \mathrm{sign}(w_1 x_1 + \cdots + w_d x_d)$$

as a way of **combining expert opinion**



Additive models cont'd

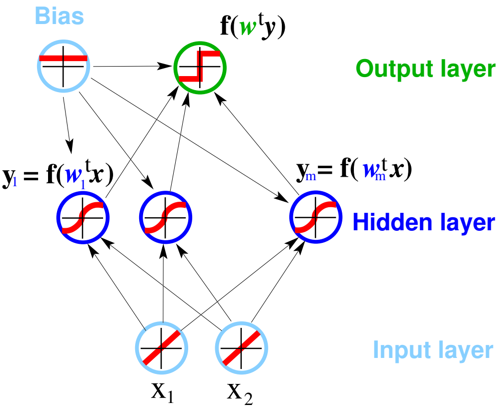
View additive models graphically in terms of **units** and **weights**.



In **neural networks** the basis functions themselves have adjustable parameters.

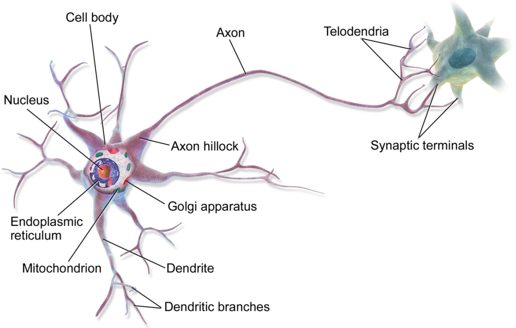
From Additive Models to Multilayer Networks

Separate units ($\rightsquigarrow$ artificial **neurons**) with **activation** f(net activation), where **net activation** is the weighted sum of all inputs, $\text{net}_j = \boldsymbol{w}_j^t \boldsymbol{x}$.



Biological neural networks

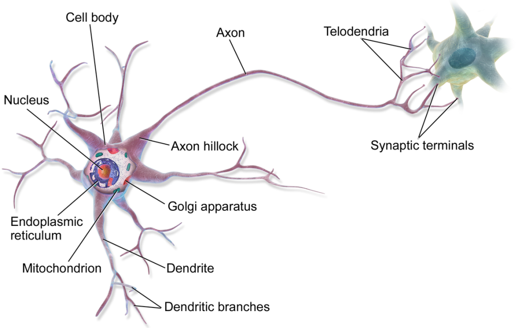
- Neurons (nerve cells): core components of **brain** and **spinal cord**. Information processing via **electrical and chemical signals**.
- Connected neurons form **neural networks**.
- Neurons have a cell body (**soma**), **dendrites**, and an **axon**.
- **Dendrites** are thin structures that arise from the soma, branching multiple times, forming a **dendritic tree**.
- **Dendritic tree collects input from other neurons.**



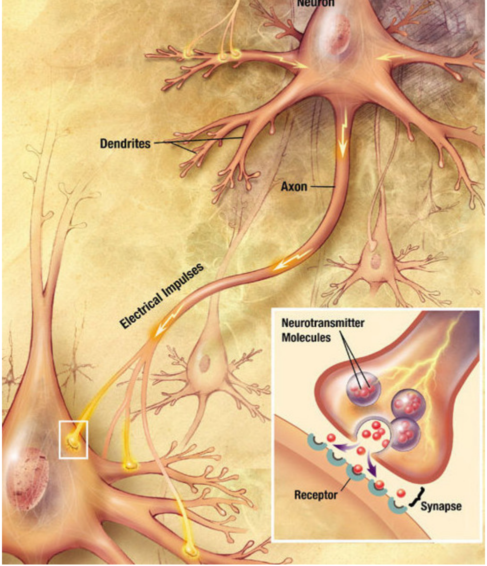
Author: BruceBlaus, Wikipedia

A typical cortical neuron

- **Axon:** cellular extension, contacts dendritic trees at **synapses.**
- **Spike of activity** in the axon
  $\rightsquigarrow$ charge injected into **post-synaptic neuron**
  $\rightsquigarrow$ chemical **transmitter molecules** released
  $\rightsquigarrow$ they **bind to receptor molecules** $\rightsquigarrow$ **in-/outflow** of ions.
- The effect of inputs is controlled by a **synaptic weight.**
- **Synaptic weights adapt $\rightsquigarrow$ whole network learns**

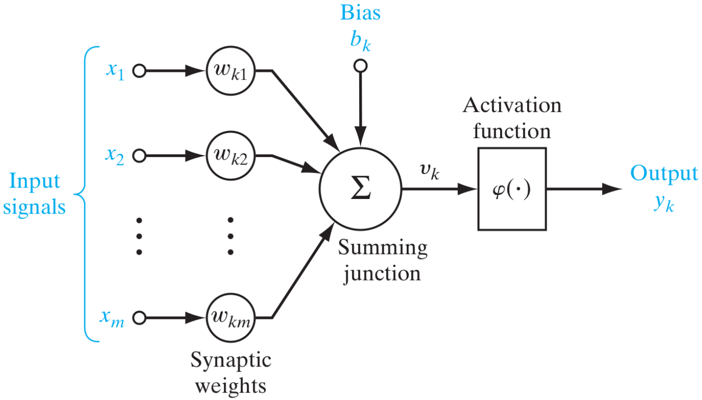


Author: BruceBlaus, Wikipedia



By user:Looie496 created file, US National Institutes of Health, National Institute on Aging created original - http://www.nia.nih.gov/alzheimers/publication/alzheimers-disease-unraveling-mystery/preface, Public Domain, https://commons.wikimedia.org/w/index.php?curid=8882110

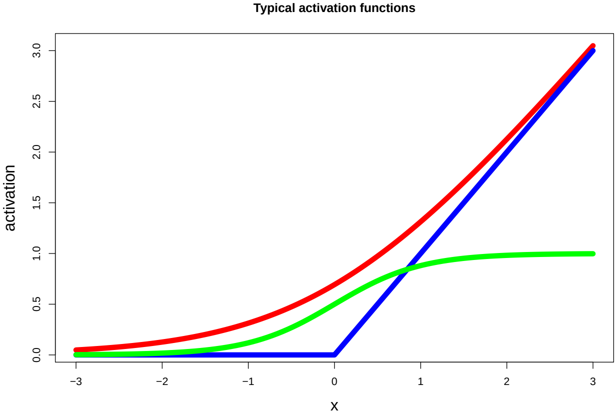
Idealized Model of a Neuron



from (Haykin, Neural Networks and Learning Machines, 2009)

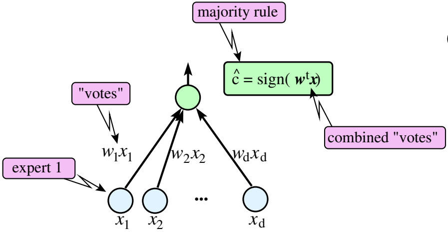
Hyperbolic tangent / Rectified / Softplus Neurons

- "Classical" activations are **smooth and bounded**, such as **tanh.**
- In modern networks **unbounded** activations are more common, like **rectifiers** ("plus"): $f(x) = x^+ = \max(0, x)$ or **softplus** $f(x) = \log(1 + \exp(x))$.

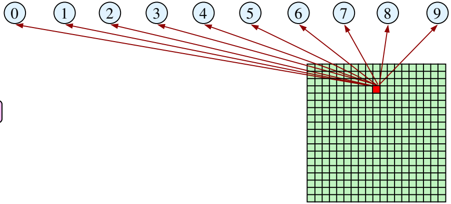


Simple NN for recognizing handwritten shapes

Two classes

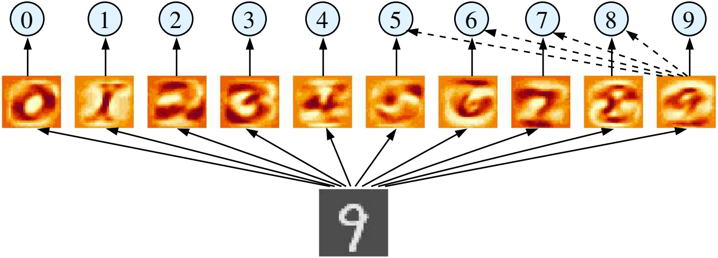


10 classes



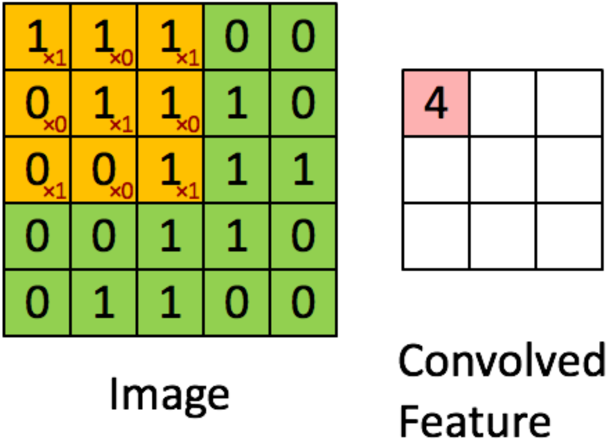
- Consider a neural network with **two layers** of neurons.
- Each pixel can vote for **several different shapes**.
- The shape that gets the **most votes** wins.

Why the simple NN is insufficient



- Simple two layer network is essentially equivalent to having a **rigid template** for each shape.
- Hand-written digits vary in many complicated ways
  $\rightsquigarrow$ simple template matches of whole shapes are not sufficient.
- To capture all variations we need to **learn the features**
  $\rightsquigarrow$ **add more layers.**
- One possible way: learn different (linear) filters
  $\rightsquigarrow$ **convolutional neural nets (CNNs).**

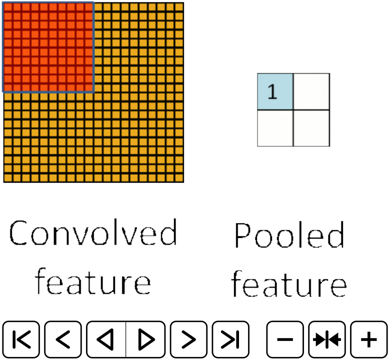
Convolutions



deeplearning.stanford.edu/wiki/index.php/Feature_extraction_using_convolution

Pooling the outputs of replicated feature detectors

- **Averaging neighboring detectors**
  ⇝ Some amount of **translational invariance.**
- **Reduces the number of inputs** to the next layer.
- Taking the **maximum** works slightly better in practice.



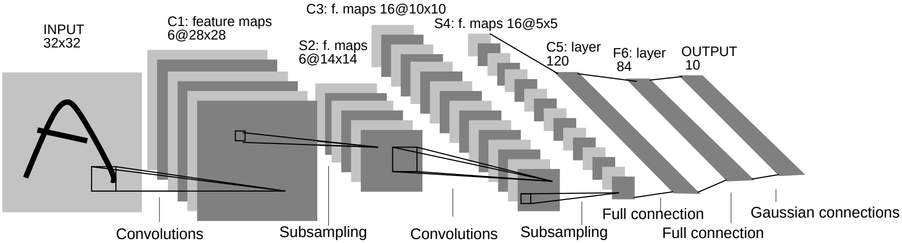
Source: deeplearning.stanford.edu/wiki/index.php/File:Pooling_schematic.gif

LeNet

Yann LeCun and his collaborators developed a really good recognizer for handwritten digits by using **backpropagation** in a feedforward net with:

- many hidden layers
- many maps of replicated units in each layer.
- pooling of the outputs of nearby replicated units.

On the **US Postal Service** handwritten digit benchmark dataset the error rate was only 4% (human error $\approx 2 - 3\%$).



Original Image published in [LeCun et al., 1998]

Network learning: Backpropagation

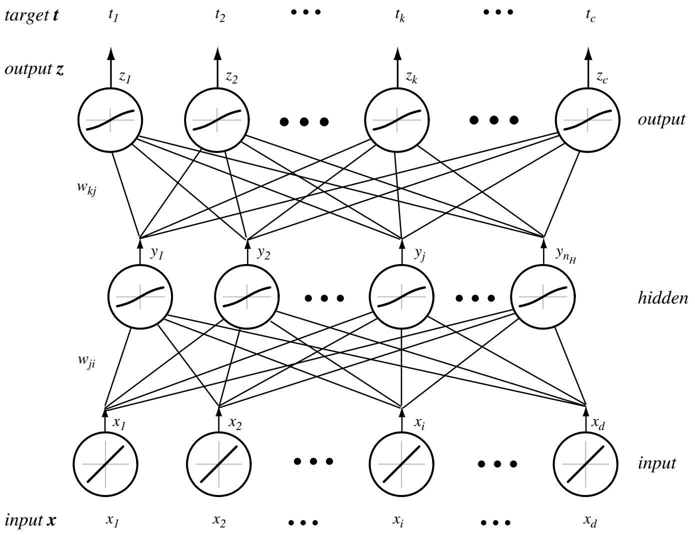


Fig 6.4 in (Duda, Hart & Stork)

Network learning: Backpropagation

- Mean squared training error: $J(\boldsymbol{w}) = \frac{1}{2n}\sum_{l=1}^{n} \|\boldsymbol{t}_l - \boldsymbol{z}_l(\boldsymbol{w})\|^2$
  $\rightsquigarrow$ all derivatives will be sums over the n training samples.
  In the following, we will focus **on one term only.**
- Gradient descent: $\boldsymbol{w} \leftarrow \boldsymbol{w} + \Delta\boldsymbol{w}$, $\Delta\boldsymbol{w} = -\eta\frac{\partial J}{\partial \boldsymbol{w}}$.
- Hidden-to-output units:

$$\frac{\partial J}{\partial w_{kj}} = \frac{\partial J}{\partial net_k}\frac{\partial net_k}{\partial w_{kj}} =: \delta_k \frac{\partial net_k}{\partial w_{kj}} = \delta_k \frac{\partial \boldsymbol{w}_k^t \boldsymbol{y}}{\partial w_{kj}} = \delta_k y_j.$$

- The **sensitivity** $\delta_k = \frac{\partial J}{\partial net_k}$ **describes how the overall error changes with the unit's net activation** $net_k = \boldsymbol{w}_k^t \boldsymbol{y}$:

$$\delta_k = \frac{\partial J}{\partial z_k}\frac{\partial z_k}{\partial net_k} = -(t_k - z_k)f'(net_k).$$

- In summary: $\Delta w_{kj} = -\eta\frac{\partial J}{\partial w_{kj}} = -\eta\delta_k y_j = \eta(t_k - z_k)f'(net_k)y_j$.

Backpropagation: Input-to-hidden units

- Output of hidden units:

$$y_j = f(net_j) = f(\boldsymbol{w}_j^t \boldsymbol{x}), \quad j = 1, \ldots, n_H.$$

- Derivative of loss w.r.t. weights at hidden units:

$$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}} =: \delta_j \frac{\partial net_j}{\partial w_{ji}} = \delta_j \, x_i.$$

- **Sensitivity at hidden unit:**

$$\delta_j = \frac{\partial J}{\partial net_j} = \frac{\partial J}{\partial y_j} \frac{\partial y_j}{\partial net_j} = \Big[\sum_{k=1}^{c} \frac{\partial J}{\partial net_k} \quad \frac{\partial net_k}{\partial y_j} \Big] \quad f'(net_j)$$

$$= \Big[\sum_{k=1}^{c} \delta_k \qquad w_{kj} \Big] \quad f'(net_j)$$

- **Interpretation:** Sensitivity at a hidden unit is proportional to weighted sum of sensitivities at output units
$\rightsquigarrow$ **output sensitivities are propagated back to the hidden units**.
- Thus, $\Delta w_{ji} = -\eta \frac{\partial J}{\partial w_{ji}} = -\eta \delta_j x_i = -\eta \big[\sum_{k=1}^{c} \delta_k w_{kj} \big] \; f'(net_j) x_i$.

Backpropagation: Sensitivity at hidden units

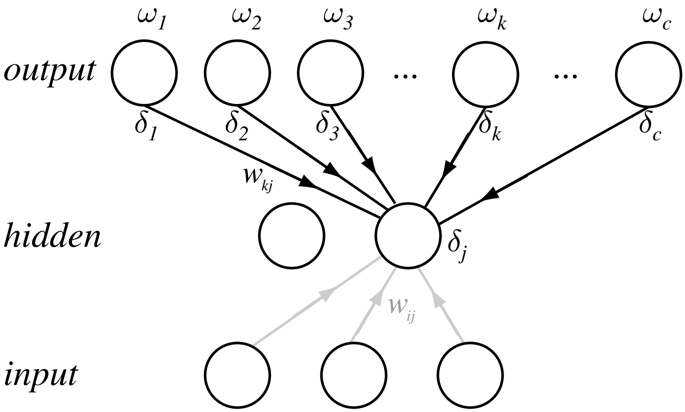


Fig 6.5 in (Duda, Hart & Stork)

Sensitivity at a hidden unit is proportional to weighted sum of sensitivities at output units

$\rightsquigarrow$ **output sensitivities are propagated back to the hidden units.**

Stochastic Backpropagation

In the previous algorithm (batch version), all gradient-based updates $\Delta \boldsymbol{w}$ were (implicitly) sums over the n input samples.
But there is also a sequential "online" variant:

Initialize $\boldsymbol{w}, m \leftarrow 1$.
Do

- $x^m \leftarrow$ randomly chosen pattern
- $w_{kj} \leftarrow w_{kj} - \eta \delta_k^m y_j^m$
- $w_{ji} \leftarrow w_{ji} - \eta \delta_j^m x_i^m$
- $m \leftarrow m + 1$

until $\|\nabla J(\boldsymbol{w})\| < \epsilon$.

Many (!) variants of this basic algorithm have been proposed.
Mini-batches are usually better than this "online" version.

Expressive Power of Networks

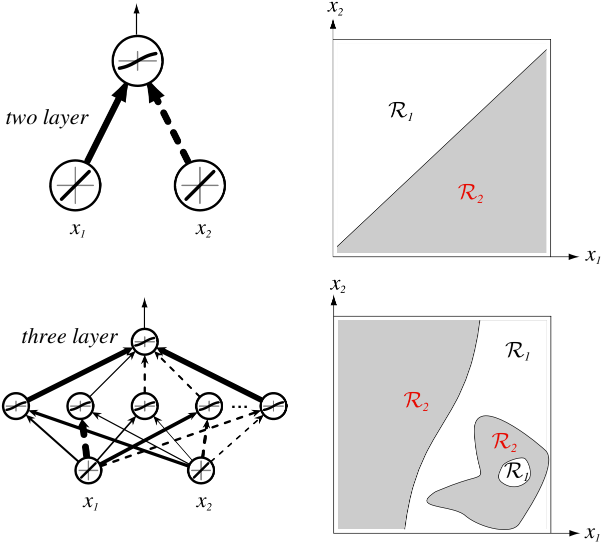


Fig 6.3 in (Duda, Hart & Stork)

Expressive Power of Networks

- **Question:** can every decision be implemented by a **three-layer network?**
- **Answer:** Basically yes – if the input-output relation is continuous and if there are **sufficiently many hidden units.**
- **Theorem** (Kolmogorov 61, Arnold 57, Lorentz 62): every continuous function $f(x)$ on the hypercube I^d ($I = [0,1], d \geq 2$) can be represented in the form

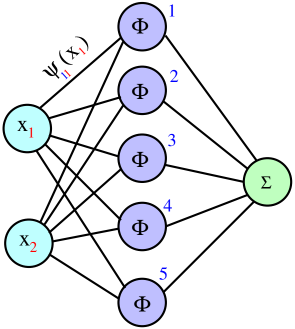
$$f(x) = \sum_{j=1}^{2d+1} \Phi\left(\sum_{i=1}^{d} \psi_{ji}(x_i)\right),$$

  for properly chosen functions Φ, ψ_{ji}.
- Note that we can always rescale the input region to lie in a hypercube.

Expressive Power of Networks

Relation to three-layer network:

- Each of $2d + 1$ hidden units takes as input a sum of d nonlinear functions, one for each input feature x_i.
- Each hidden unit emits a nonlinear function Φ of its total input.
- The output unit emits the sum of all contributions of the hidden units.



Problem: Theorem guarantees only existence

$\rightsquigarrow$ might be hard to find these functions.

Are there "simple" function families for Φ, ψ_{ji}?

Let's review some classical function approximation results...

Polynomial Function Approximation

Theorem (Weierstrass Approximation Theorem)

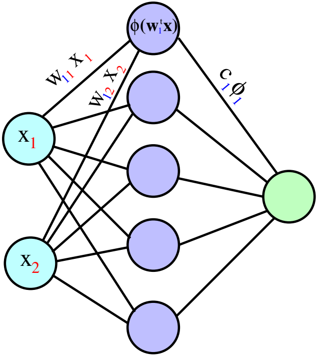
Suppose f is a continuous real-valued function defined on the real interval $[a, b]$, i.e. $f \in C([a, b])$. For every $\epsilon > 0$, there exists a polynomial p such that $\|f - p\|_{\infty,[a,b]} < \epsilon$.

In other words: Any given real-valued continuous function on $[a, b]$ can be **uniformly approximated by a polynomial function.**

Polynomial functions are dense in $C([a, b])$.

Ridge functions

- Ridge function (1d):
  $f(x) = \varphi(wx + b)$, $\varphi : \mathbb{R} \to \mathbb{R}$, $w, b \in \mathbb{R}$.
- General form: $f(\boldsymbol{x}) = \varphi(\boldsymbol{w}^t \boldsymbol{x} + b)$,
  $\varphi : \mathbb{R} \to \mathbb{R}$, $\boldsymbol{w} \in \mathbb{R}^d$, $b \in \mathbb{R}$.
- Assume $\varphi(\cdot)$ is differentiable at
  $z = \boldsymbol{w}^t \boldsymbol{x} + b$
  $\rightsquigarrow \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = \varphi'(z) \nabla_{\boldsymbol{x}}(\boldsymbol{w}^t \boldsymbol{x} + b) = \varphi'(z)\boldsymbol{w}$.
- Gradient descent is simple: direction defined by linear part.



Relation to function approximation:

(i) polynomials can be represented arbitrarily well by combinations of ridge functions $\rightsquigarrow$ **ridge functions are dense on $C([0,1])$.**

(ii) **"Dimension lifting"** argument (Hornik 91, Pinkus 99): density on the unit interval also implies **density on the hypercube.**

Universal approximations by ridge functions

Theorem (Cybenko 89, Hornik 91, Pinkus 99)

Let $\varphi(\cdot)$ be a non-constant, bounded, and monotonically-increasing continuous function. Let I^d denote the unit hypercube $[0,1]^d$, and $C(I^d)$ the space of continuous functions on I^d. Then, given any $\varepsilon > 0$ and any function $f \in C(I^d)$, there exist an integer N, real constants $v_i, b_i \in \mathbb{R}$ and real vectors $\boldsymbol{w}_i \in \mathbb{R}^d$, $i = 1, \cdots, N$, such that we may define:

$$F(x) = \sum_{i=1}^{N} v_i \varphi\left(\boldsymbol{w}_i^t \boldsymbol{x} + b_i\right)$$

as an approximate realization of the function f, i.e. $\|F - f\|_{\infty, I^d} < \varepsilon$.

In other words, functions of the form $F(x)$ are dense in $C(I^d)$.

This still holds when replacing I^d with any compact subset of $\mathbb{R}^d$.

Artificial Neural Networks: Rectifiers

- Classic activation functions are indeed **bounded** and **monotonically-increasing continuous** functions like *tanh*.
- In practice, however, it is often better to use "simpler" activations.
- **Rectifier:** activation function defined as:

  $$f(x) = x^+ = \max(0, x),$$

  where x is the input to a neuron.
  Analogous to **half-wave rectification in electrical engineering.**
- A unit employing the rectifier is called **rectified linear unit (ReLU).**
- What about approximation guarantees?
  Basically, we have the same guarantees,
  but at the price of **wider layers**...

Universal Approximation by ReLu networks

- Any $f \in C[0;1]$ can be uniformly approximated to arbitrary precision by a **polygonal line** (cf. Shekhtman, 1982)
- Lebesgue (1898): polygonal line on $[0,1]$ with m pieces can be written

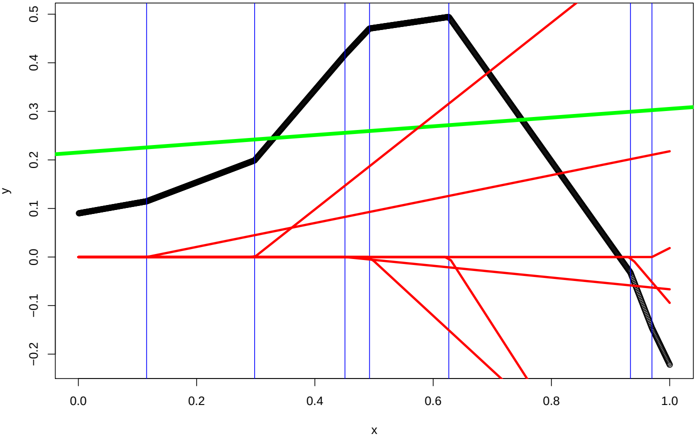
$$g(x) = ax + b + \sum_{i=1}^{m-1} c_i (x - x_i)_+,$$

  with knots $0 = x_0 < x_1 < \cdots < x_{m-1} < x_m = 1$,
  and $m+1$ parameters $a, b, c_i \in \mathbb{R}$.
- We might call this a **ReLU function approximation** in 1d.
  A **dimension lifting** argument similar to above leads to:

Theorem

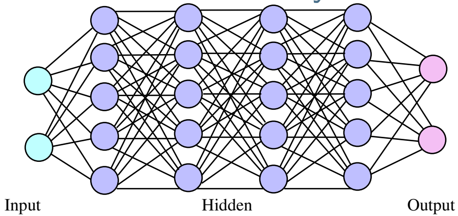
Networks with one (wide enough) hidden layer of ReLU are universal approximators for continuous functions.

Universal Approximation by ReLu networks



Green: $a + bx$. Red: individual functions $c_i(x - x_i)_+$. Black: $g(x)$.

Why should we use more hidden layers?



- Idea: characterize the expressive power by counting into how many cells we can partition $\mathbb{R}^d$ with combinations of rectifying units.
- A rectifier is a piecewise linear function. It partitions $\mathbb{R}^d$ into two open half spaces (and a border face):

$$\begin{aligned} H^+ &= \boldsymbol{x} : \boldsymbol{w}^t \boldsymbol{x} + b > 0 \in \mathbb{R}^d \\ H^- &= \boldsymbol{x} : \boldsymbol{w}^t \boldsymbol{x} + b < 0 \in \mathbb{R}^d \end{aligned}$$

- Question: by linearly combining m rectified units, into how many cells is $\mathbb{R}^d$ maximally partitioned?
- Explicit formula (Zaslavsky 1975): An arrangement of m hyperplanes in $\mathbb{R}^n$ has at most $\sum_{i=0}^{n} \binom{m}{i}$ regions.

Linear Combinations of Rectified Units and Deep Learning

Applied to ReLu networks (Montufar et al, 2014):

Theorem

A rectifier neural network with d input units and L hidden layers of width $m \geq d$ can compute functions that have $\Omega\left(\left(\frac{m}{d}\right)^{(L-1)d} m^d\right)$ linear regions.

Important insights:

- The number of linear regions of deep models grows **exponentially in L** and **polynomially in m**.
- This growth is much faster than that of **shallow networks** with the same number mL of hidden units in a single wide layer.

Implementing Deep Network Models

Modern libraries like **TensorFlow/Keras** or **PyTorch** make implementation simple:

- Libraries provide primitives for defining functions and automatically computing their derivatives.
- Only the forward model needs to be specified, gradients for backprop are computed automatically!
- GPU support.
- See PyTorch examples in the exercise class.

Subsection 2

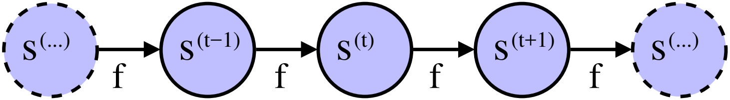
Recurrent Neural Networks

Unfolding Computational Graphs

- **Computational graph**
  $\rightsquigarrow$ formalize structure of a set of computations, e.g. mapping inputs and parameters to outputs and loss.
- Classical form of a **dynamical system:**

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

  where $\boldsymbol{s}^{(t)}$ is the **state** of the system.



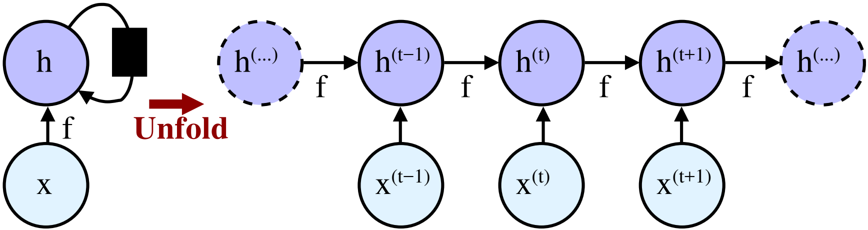
- For a finite number of time steps τ, the graph can be **unfolded** by applying the definition $\tau - 1$ times, e.g. $\boldsymbol{s}^{(3)} = f(f(\boldsymbol{s}^{(1)}))$.
- Often, a dynamical system is driven by an **external signal:**

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

Unfolding Computational Graphs

- State is the **hidden units** of the network:

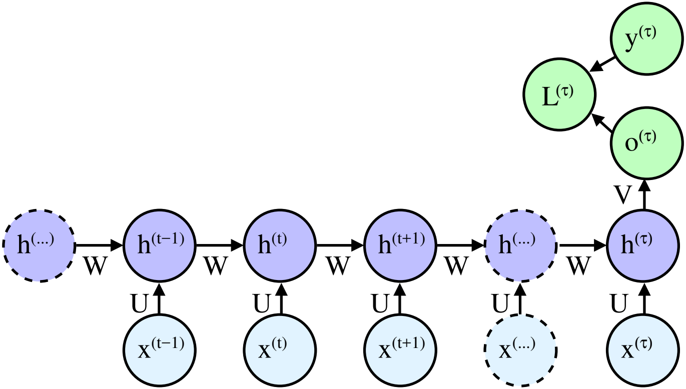
$$\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta}),$$



A RNN with no outputs. It just incorporates information about $\boldsymbol{x}$ by incorporating into $\boldsymbol{h}$. This information is passed forward through time. (Left) Circuit diagram. Black square: delay of one time step.
(Right) Unfolded computational graph.

Unfolding Computational Graphs

- The network typically learns to use the fixed length state $\boldsymbol{h}^{(t)}$ as a **lossy summary of the task-relevant aspects of $\boldsymbol{x}^{(1:t)}$.**

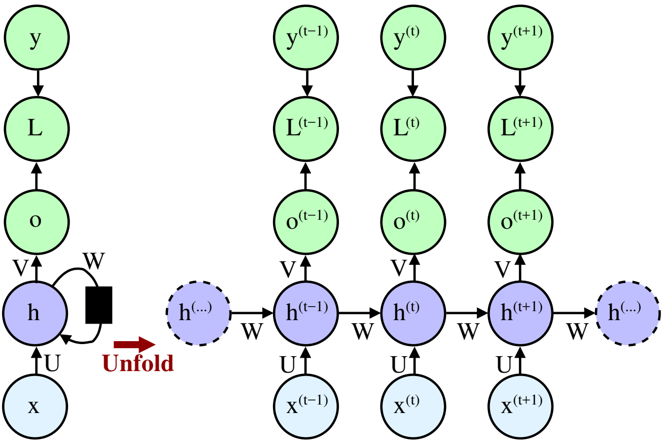


Time-unfolded RNN with a single output at the end of the sequence.

Unfolding Computational Graphs

- We can represent the unfolded recurrence after t steps with a function $g^{(t)}$ that takes the whole past sequence as input:
$$\boldsymbol{h}^{(t)} = g^{(t)}(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \ldots, \boldsymbol{x}^{(1)}) = f(\boldsymbol{h}^{(t-1)}\boldsymbol{x}^{(t)}; \boldsymbol{\theta})$$
Recurrent structure
$\rightsquigarrow$ can factorize $g^{(t)}$ into **repeated application of function f.**
- The unfolding process has two advantages:
(i) Learned model specified in terms of transition from one state to another state $\rightsquigarrow$ **always the same size.**
(ii) We can use the **same transition function f** at every time step.
- Possible to learn a **single model f** that operates on **all time steps and all sequence lengths.**
- A single shared model allows generalization to sequence lengths that **did not appear in the training set**, and requires fewer training examples.

Recurrent Neural Networks



This general RNN maps an input sequence $\boldsymbol{x}$ to the output sequence $\boldsymbol{o}$.

Universality: any function computable by a Turing machine can be computed by such a network of finite size.

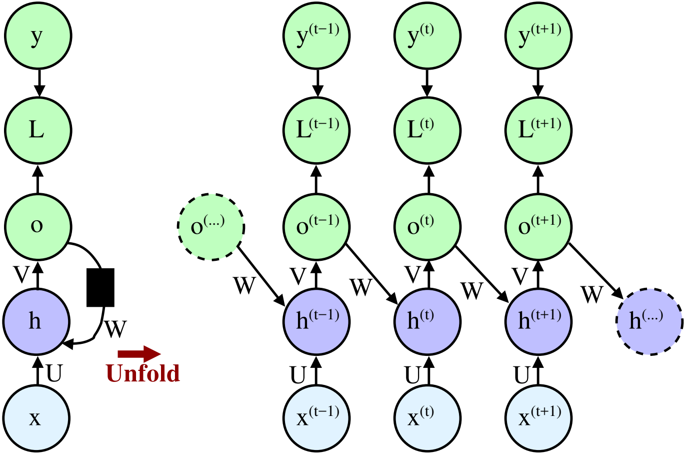
Recurrent Neural Networks

- Hyperbolic tangent activation function $\rightsquigarrow$ forward propagation:

$$
\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + W\boldsymbol{h}^{(t-1)} + U\boldsymbol{x}^{(t)}, \\
\boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}), \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + V\boldsymbol{h}^{(t)}, \\
\hat{\boldsymbol{y}}^{(t)} &= \mathrm{softmax}(\boldsymbol{o}^{(t)}).
\end{aligned}
$$

- Here, the RNN maps the input sequence to an output sequence of the same length. **Total loss = sum of the losses over all times t_i.**
- Computing the gradient is expensive: **forward propagation pass** through unrolled graph, followed by **backward propagation pass.**
- It is called **back-propagation through time (BPTT).**
- Runtime is $O(\tau)$ and **cannot be reduced by parallelization** because the forward propagation graph is inherently sequential.

Simpler RNNs

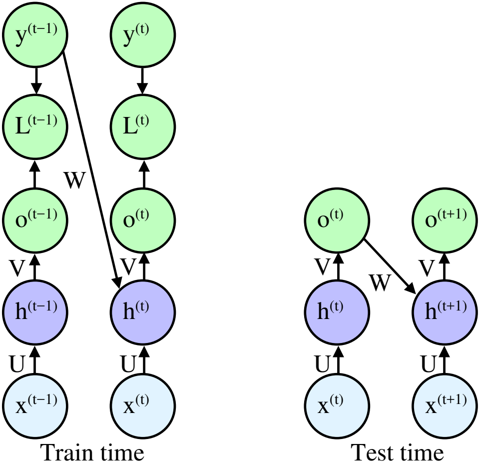


An RNN whose only recurrence is the feedback connection from the output to the hidden layer. The RNN is trained to put a specific output value into $\boldsymbol{o}$, and $\boldsymbol{o}$ is the only information it is allowed to send to the future.

Networks with Output Recurrence

- Recurrent connections only from the output at one time t to the hidden units at time $t + 1 \rightsquigarrow$ simpler, but less powerful.
- **Lacks hidden-to-hidden recurrence** $\rightsquigarrow$ requires that output units capture all relevant information about the past.
- Advantage: for any loss function based on comparing the $\boldsymbol{o}^{(t)}$ to the target $\boldsymbol{y}^{(t)}$, all the **time steps are decoupled.**
- **Training can be parallelized:**
  Gradient for each step t can be computed in isolation: no need to compute the output for the previous time step first, because training set provides the **ideal value of that output** $\rightsquigarrow$ **Teacher forcing.**

Teacher Forcing



(Left) At train time, we feed the correct output $y^{(t)}$ as input to $h^{(t+1)}$. (Right) When the model is deployed, the true output is not known. In this case, we approximate the correct output $y^{(t)}$ with the model's output $o^{(t)}$.

Sequence-to-sequence architectures

- So far: RNN maps input to output sequence of **same length.**
- **What if these lengths differ?**
  $\rightsquigarrow$ speech recognition, machine translation etc.
- Input to the RNN called the **context**. Want to produce a representation of this context, C: a vector summarizing the input sequence $X = (x^{(1)}, \dots, x^{(n_x)})$.
- Approach proposed in [Cho et al., 2014]:
  (i) **Encoder** processes the input sequence and emits the context C, as a (simple) function of its final hidden state.
  (ii) **Decoder** generates output sequence $Y = (y^{(1)}, \dots, y^{(n_y)})$.
- The two RNNs are **trained jointly** to maximize the average of $\log P(Y|X)$ over all the pairs of x and y sequences in the training set.
- The **last state h_{n_x} of the encoder RNN** is used as the representation C.

Sequence-to-sequence architectures

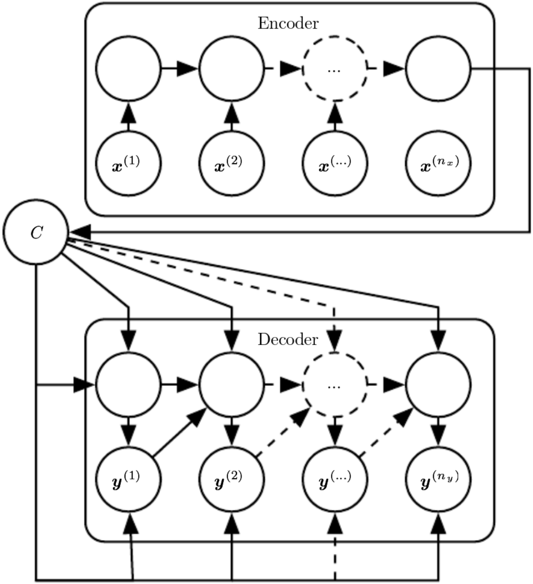
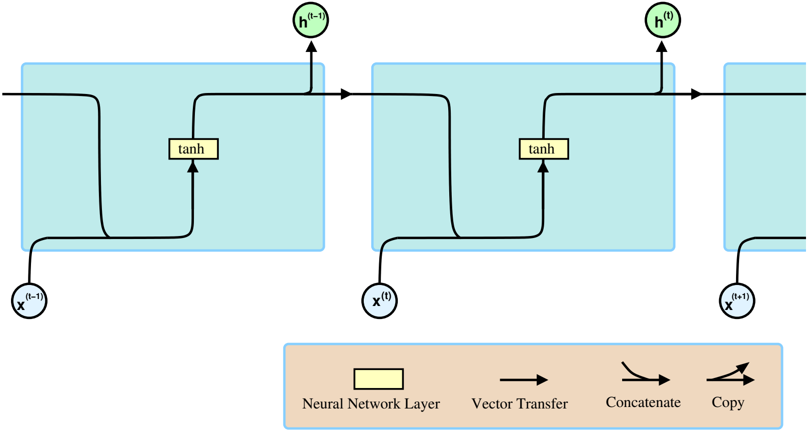


Fig 10.12 in (Goodfellow, Bengio, Courville)

Long short-term memory (LSTM) cells

- Theory: RNNs can keep track of **arbitrary long-term dependencies.**
- **Practical problem:** computations in finite-precision: $\rightsquigarrow$ **Gradients can vanish or explode.**
- RNNs using **LSTM units** partially solve this problem: LSTM units allow gradients to also **flow unchanged.** However, exploding gradients may still occur.
- Common architectures composed of a cell and three **regulators** or **gates of the inflow:** input, output and forget gate.
- Variations: gated recurrent units (GRUs) do not have an output gate.
- **Input gate** controls to which extent a new value flows into the cell
- **Forget gate** controls to which extent a value remains in the cell
- **Output gate** controls to which extent the current value is used to compute the output activation.

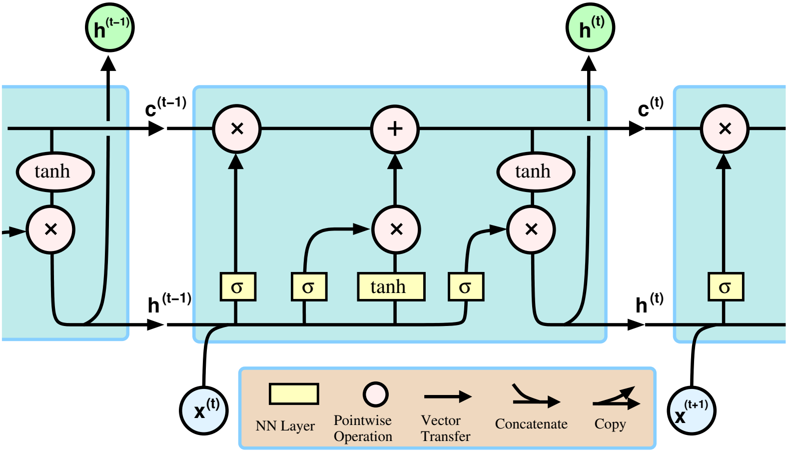
Recall: RNNs



$$\boldsymbol{h}^{(t)} = \tanh(W[\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}] + \boldsymbol{b})$$

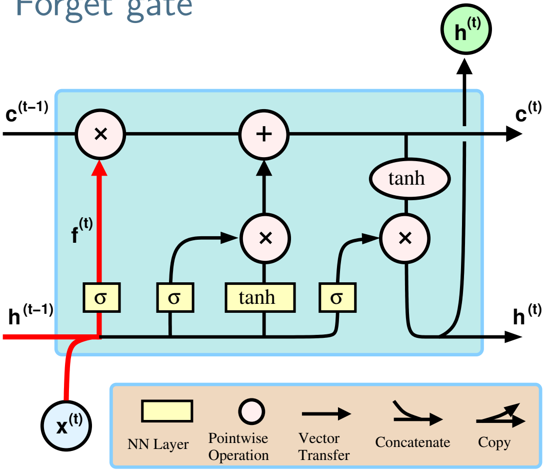
RNN cell takes current input $\boldsymbol{x}^{(t)}$ and outputs the hidden state $\boldsymbol{h}^{(t)}$
$\rightsquigarrow$ pass to the next RNN cell.

Long short-term memory (LSTM) cells



Cell states allows flow of unchanged information
⇝ helps preserving context, learning long-term dependencies.

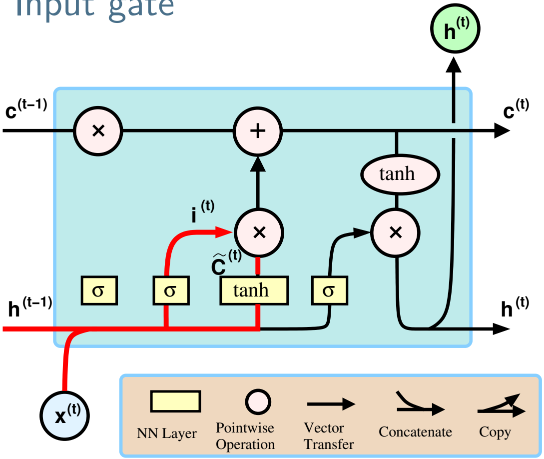
LSTM cells: Forget gate



$$\boldsymbol{f}^{(t)} = \sigma(W^f[\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}] + \boldsymbol{b}^f)$$

Forget gate alters cell state based on current input $\boldsymbol{x}^{(t)}$ and output $\boldsymbol{h}^{(t-1)}$ from previous cell.

LSTM cells: Input gate

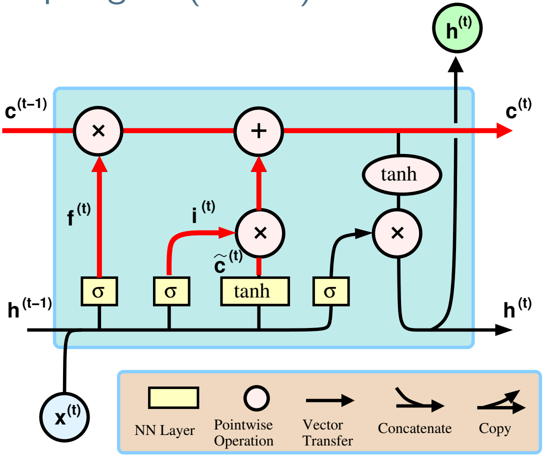


$$\boldsymbol{i}^{(t)} = \sigma(W^i[\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}] + \boldsymbol{b}^i)$$

$$\tilde{\boldsymbol{c}}^{(t)} = \tanh(W^c[\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}] + \boldsymbol{b}^c)$$

Input gate decides and computes values to be updated in the cell state.

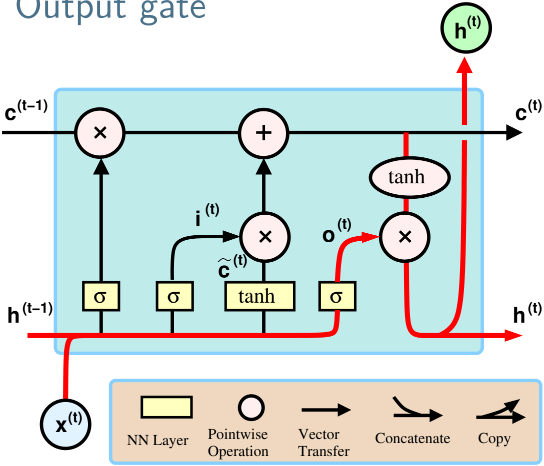
LSTM cells: Input gate (cont'd)



$$\boldsymbol{c}^{(t)} = \boldsymbol{f}^{(t)} \circ \boldsymbol{c}^{(t-1)} + \boldsymbol{i}^{(t)} \circ \tilde{\boldsymbol{c}}^{(t)}$$

Forget and input gate together update old cell state.

LSTM cells: Output gate

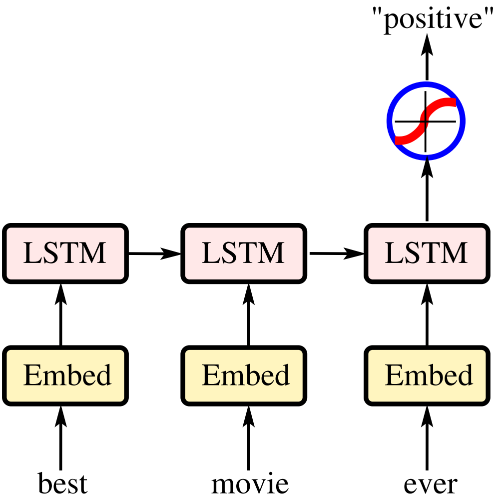


$$\boldsymbol{o}^{(t)} = \sigma(W^o[\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}] + \boldsymbol{b}^o)$$

$$\boldsymbol{h}^{(t)} = \tanh(\boldsymbol{c}^{(t)}) \circ \boldsymbol{o}^{(t)}$$

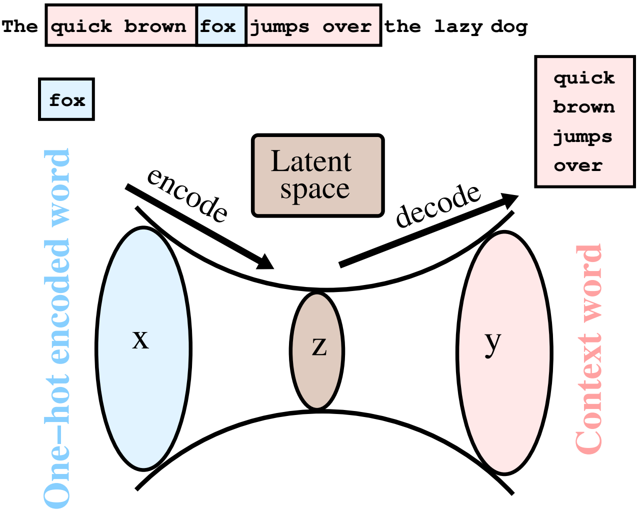
Output gate computes output from cell state to be sent to next cell.

LSTM example: movie review



Inputs: words in a movie review

Word embeddings

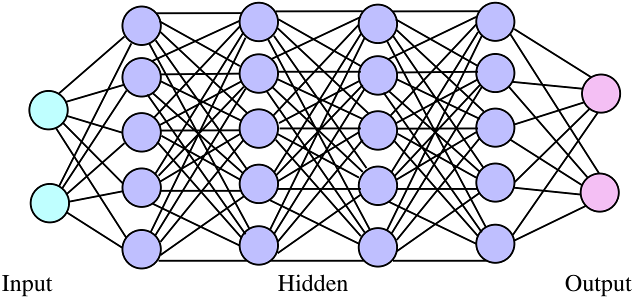


Multidimensional, distributed representation of words in a vector space.

Subsection 3

Interpretability in deep learning models

Interpretability in deep learning models



Deep neural networks are accurate but difficult to understand.
Can we directly optimise deep models for interpretability?

(M Wu, MC Hughes, S Parbhoo, M Zazzi, V Roth, F Doshi-Velez, AAAI 2018)

Existing Methods for Interpretability

Current methods try to interpret trained models.

[Selvaraju et. al. 2017]

A group of people flying kites on a beach   A man is sitting at a table with a pizza

(a) Image captioning explanations

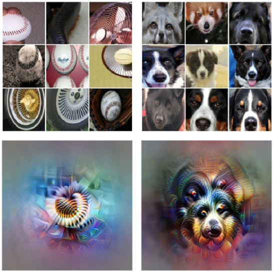
A house with a green roof   Sheep grazing in field   A house with a roof

(b) Comparison to DenseCap

[Olah et. al. 2017]

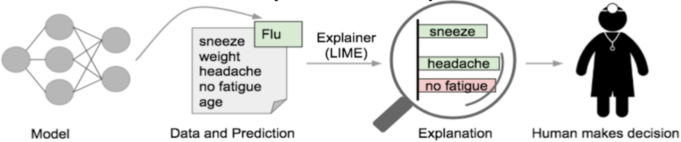
Dataset Examples show us what neurons respond to in practice

Optimization isolates the causes of behavior from mere correlations. A neuron may not be detecting what you initially thought.

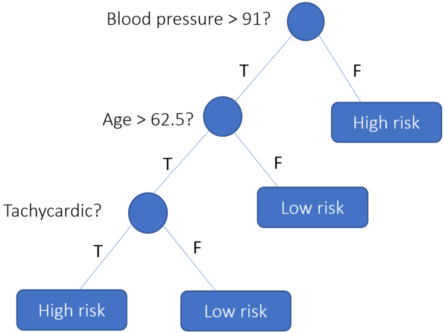


Baseball—or stripes? *mixed4a, Unit 6*   Animal faces—or snouts?

[Ribeiro et. al. 2017]



Small Trees are interpretable



- Decisions may be simulated.
- Decisions may be understood directly in terms of feature space.
- Ave. path length: cost of simulating ave. example.

But decision trees produce less accurate predictions.

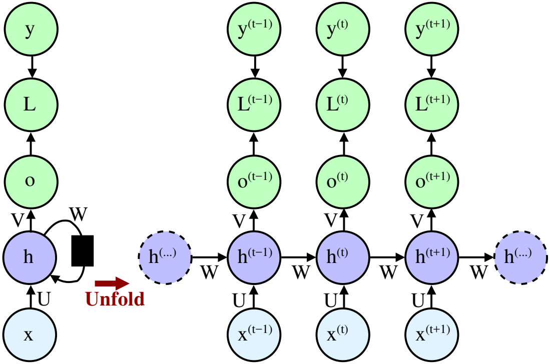
Can we optimise a neural network to be interpretable and accurate?

RNNs

- Timeseries data: N, T_n timesteps each, binary outputs.
- Train a recurrent neural network (RNN) with loss:

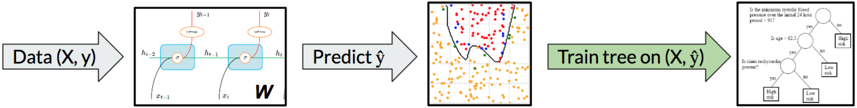
$$\lambda\psi(W) + \sum_{n=1}^{N} \sum_{t=1}^{T_n} \text{loss}(y_{nt}, \hat{y}_{nt}(x_{nt}, W))$$

where ψ is a regularizer (i.e. L1 or L2), λ is a regularization strength



Tree Regularisation for Interpretability

- Pass training data X through the RNN to make predictions $\hat{\boldsymbol{y}}$.
- Train DT on $X, \hat{\boldsymbol{y}}$ to try to match the RNN predictions.

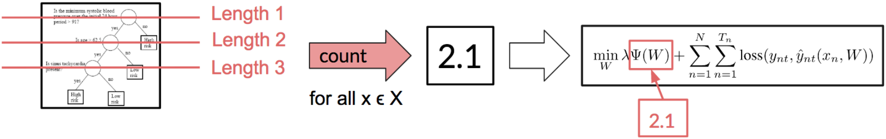


At any point in the optimization, approximate partially trained RNN with simple DT.

Tree Regularisation for Interpretability

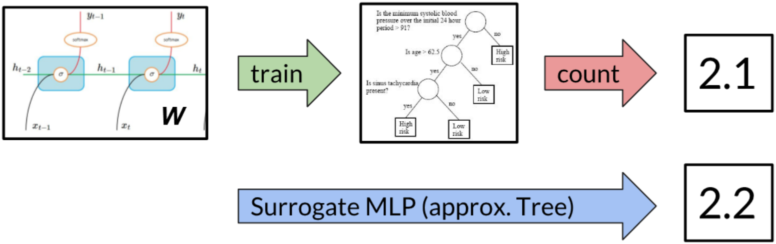
- Use average path length of DT to constrain predictions
- Interpretation: cost for a human to simulate the average example.
- Redefine loss function:

$$\lambda \sum_{n=1}^{N} \sum_{n=1}^{T_n} \textbf{pathlength}(x_{nt}, \hat{y}_{nt}) + \sum_{n=1}^{N} \sum_{n=1}^{T_n} \text{loss}(y_{nt}, \hat{y}_{nt}(x_{nt}, W))$$



Optimizing a Surrogate of the Tree

But DTs aren't differentiable ⇝ use a surrogate network to mimic the tree:

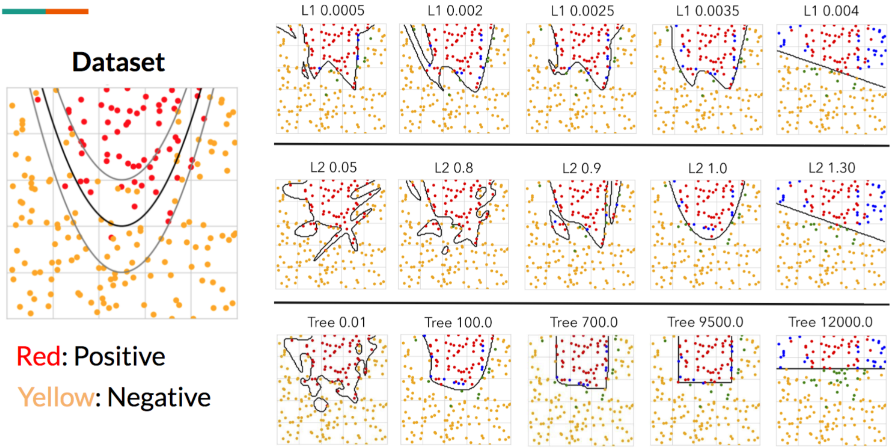


Given fixed [Surrogate MLP →], optimize **W** via gradient descent.

Given fixed **W**, we can find the best [Surrogate MLP →].

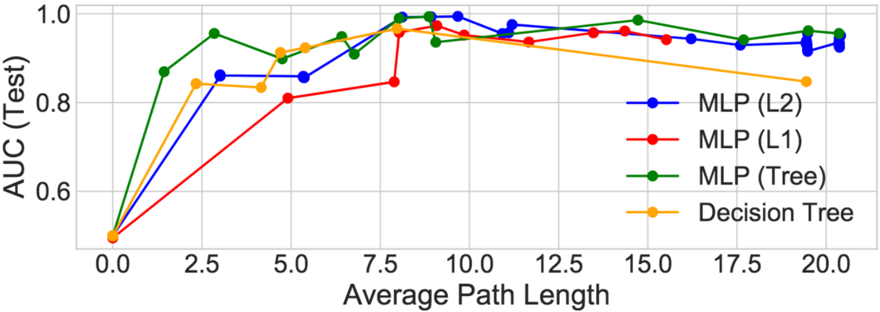
Toy dataset

- Parabolic decision function $y = 5 * (x - 0.5)^2 + 0.4$
- Points above parabola are positive, points below negative.
- Tree regularization produces axis aligned boundaries.



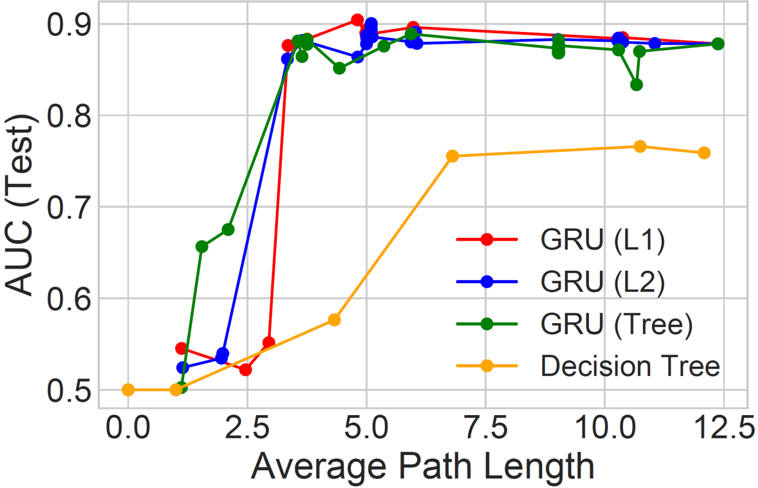
Results: High Accuracy Predictions

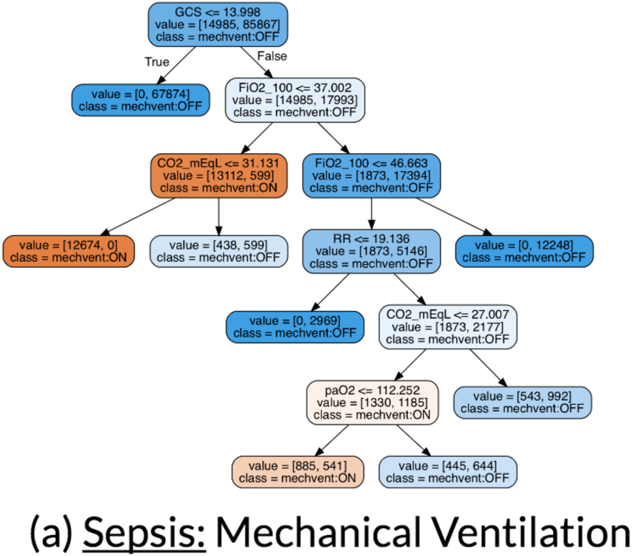
Better performance in high regularization (human-simulatable) regimes:



Real-word Data: Sepsis (Johnson et. al. 2016)

Time-series data for 11k septic intensive-care-unit (ICU) patients.
35 hourly features: respiration rate (RR), blood oxygen levels (paO2) etc.
Binary outcome: if a ventilation was used.





Important features (FiO2, RR, CO2, and paO2) are medically valid

Real-word Data: EuResist (Zazzi et. al. 2012)

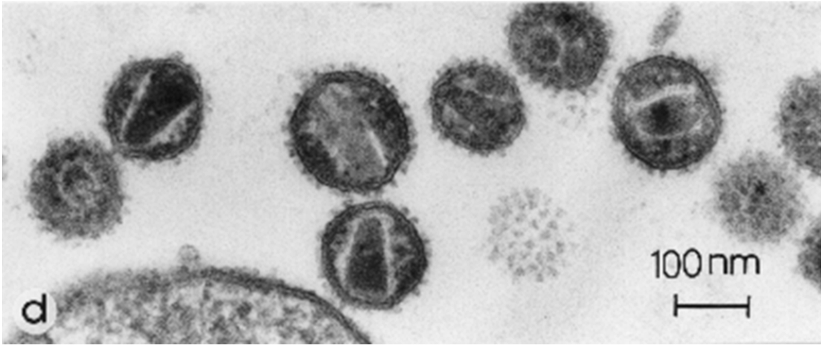
Time-series data for 50k patients diagnosed with HIV.

Time steps: 4-6 month intervals (hospital visits).

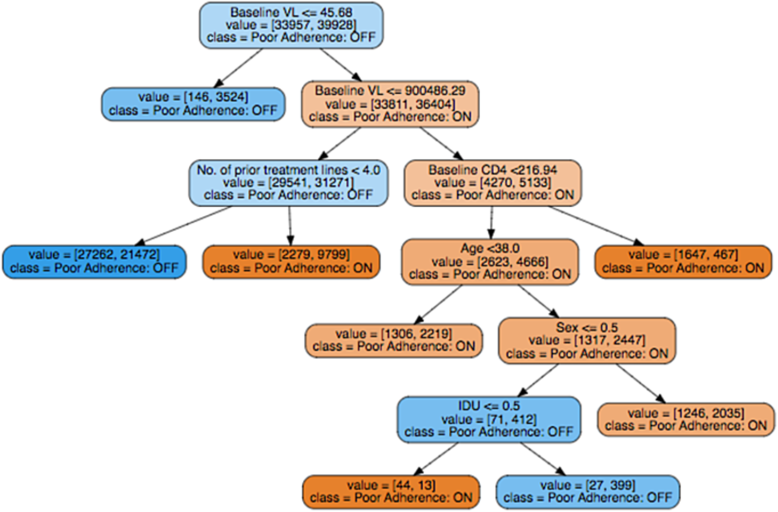
40 input features (blood counts, viral load, viral genotype etc.)

15 output features (viral load reduction, adherence issues, etc.)

The average sequence length is 14 steps.



HR Gelderblom *et al.*, Virology 1987



(Langford et al. 2007, Socas et al. 2011): **high baseline viral loads**
$\rightsquigarrow$ faster disease progression $\rightsquigarrow$ need multiple drug cocktails
$\rightsquigarrow$ harder for patients to adhere to prescription.

Summary of Tree Regularisation

Regularise deep models such that they have
high-accuracy and low complexity.
Axis-aligned decision boundaries that are
easy to interpret and explain decisions.
Decision trees that make faithful predictions and can be used to
personalise therapies.