

Neural MT: Mathematics and Algorithms

Dimitar Shterionov

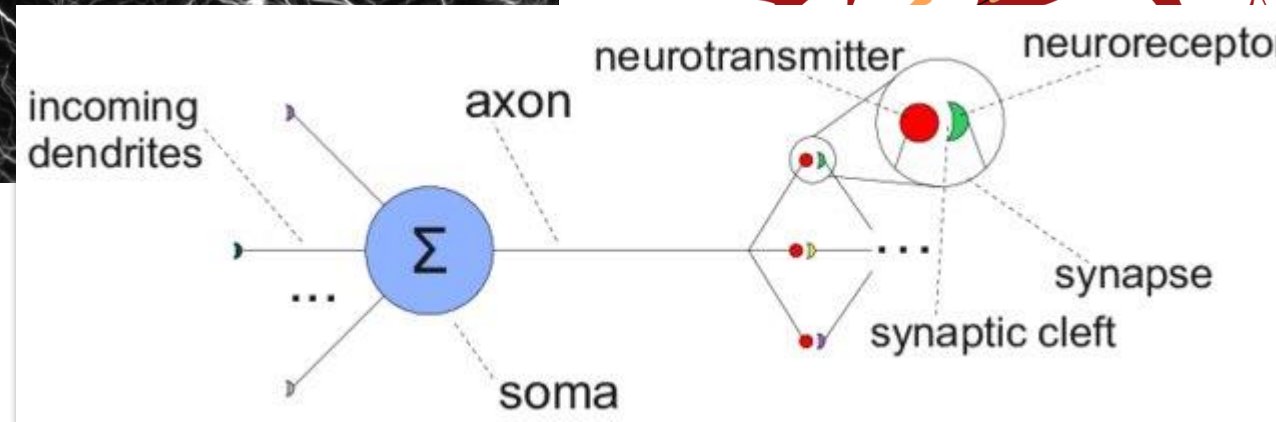
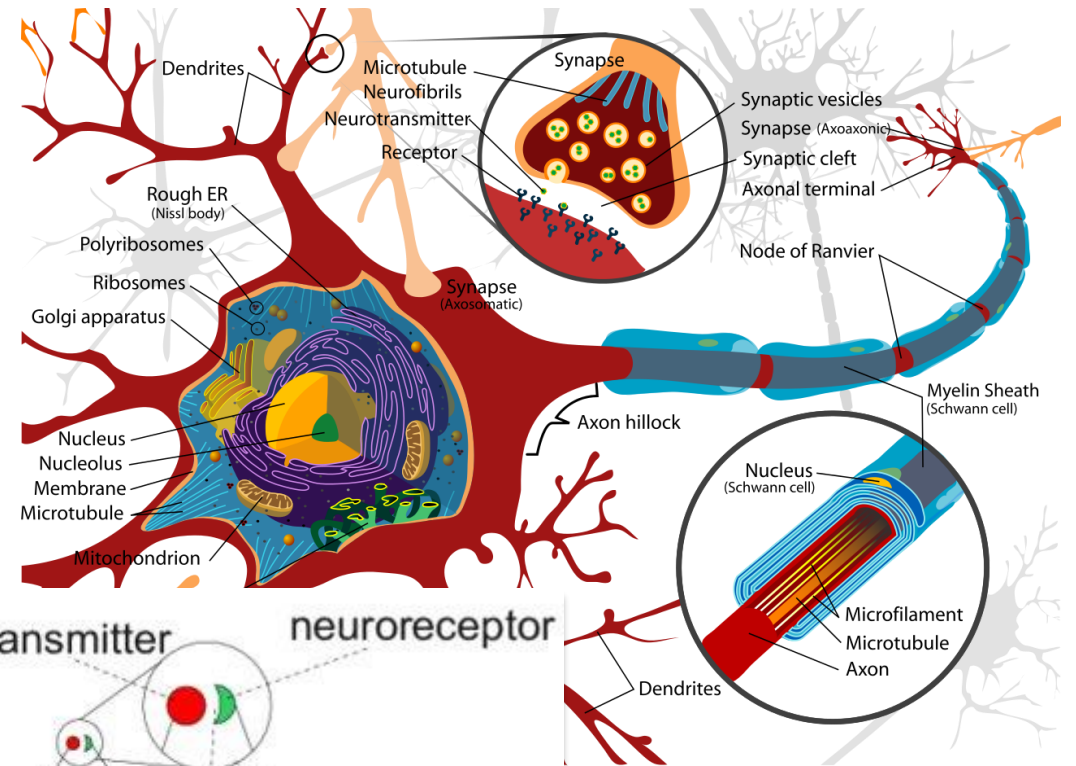
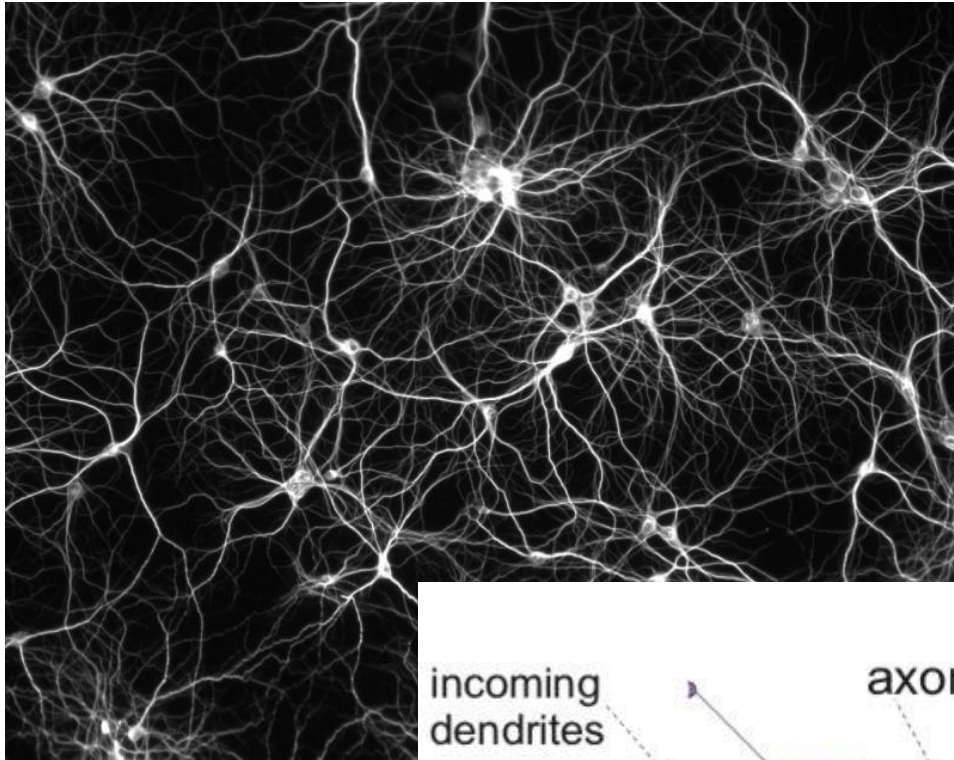
ADAPT Centre, Dublin City University

11/04/2019

Outline

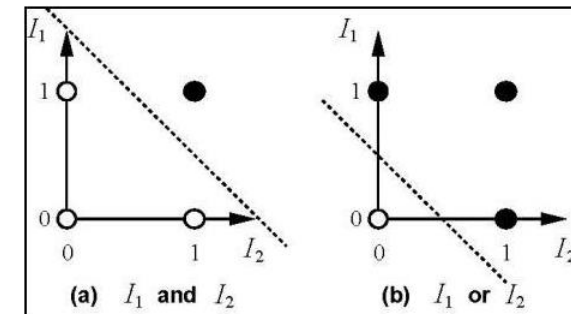
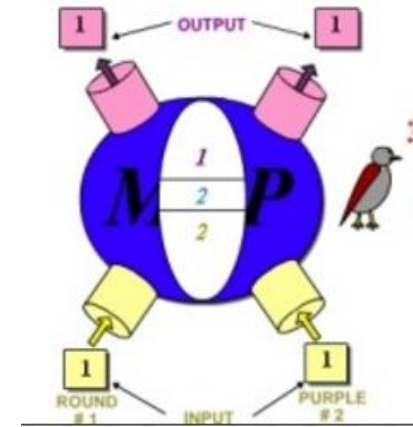
- A brief history
- Basic mathematics in Neural Networks (NN)
- Neural Machine Translation (NMT)
- Advanced NMT

History



History

- 1943: McCulloch and Pitts model neural networks based on their understanding of neurology.
 - Neurons embed simple logic functions:
 - a or b
 - a and b
- 1950s:
 - Farley and Clark
 - IBM group that tries to model biological behaviour
 - Consult neuro-scientists at McGill, whenever stuck
 - Rochester, Holland, Haibit and Duda



History

- Perceptron (Rosenblatt 1958)

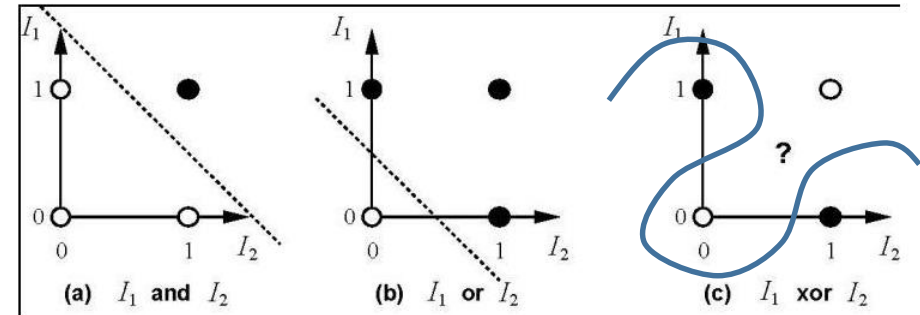
- Three layer system:

- Input nodes
 - Output node
 - Association layer

- Can learn to connect or associate a given input to a random output unit

- Minsky and Papert (1969)

- Marvin Lee Minsky: Pioneer of Artificial Intelligence, Turing Award (1969)
 - Showed that a single layer perceptron cannot learn the XOR of two binary inputs
 - Lead to loss of interest (and funding) in the field



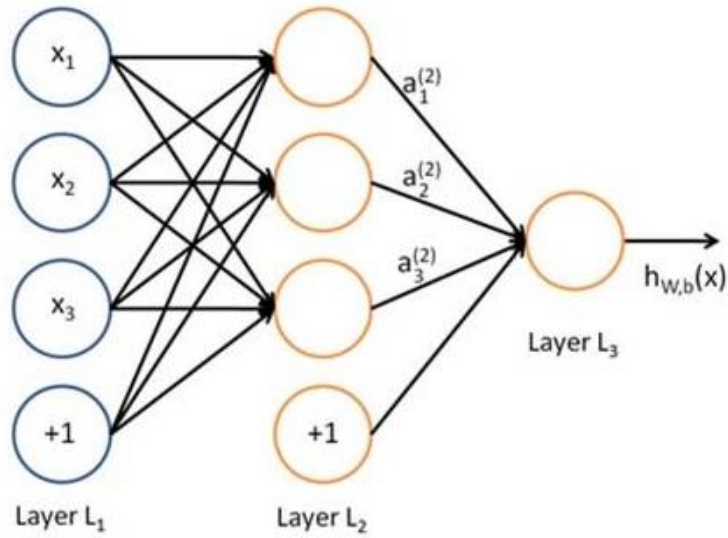
History

- Back-propagation learning method (Werbos 1974)
 - Three layers of neurons
 - Input, Output, Hidden
 - Better learning rule for generic three layer networks
 - Regenerates interest in the 1980s
- 1990s:
 - Successful applications in many fields
 - But only three layer networks
 - Hard to train the network with more and deep layers (gradient vanishing)

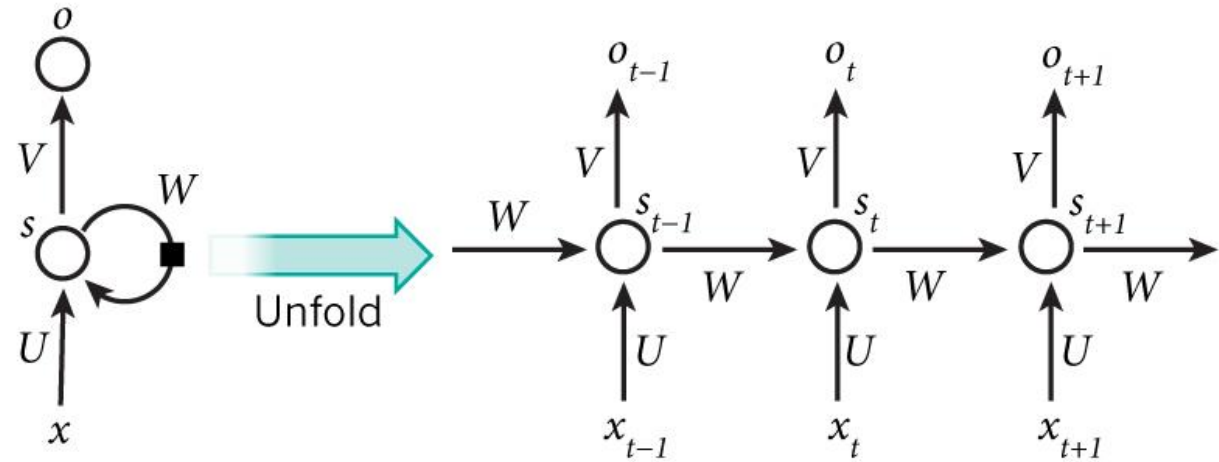
History

- Deep learning (2006 to now)
 - Geoffrey E. Hinton et al.: A Fast Learning Algorithm for Deep Belief Nets (2006)
 - pre-training method to alleviate the gradient vanishing and local optimum problems for deep neural networks
 - the number of hidden layers was up to 7
 - Activation functions: ReLU and variants, maxout etc.
 - New cells: long short-term memory (LSTM), Gated Recurrent Unit (GRU)
 - New or evolved architectures: convolutional neural network (CNN), recurrent neural network (RNN), highway network, deep residual learning/network (ResNet)
 - Deep ResNet: 152 layers

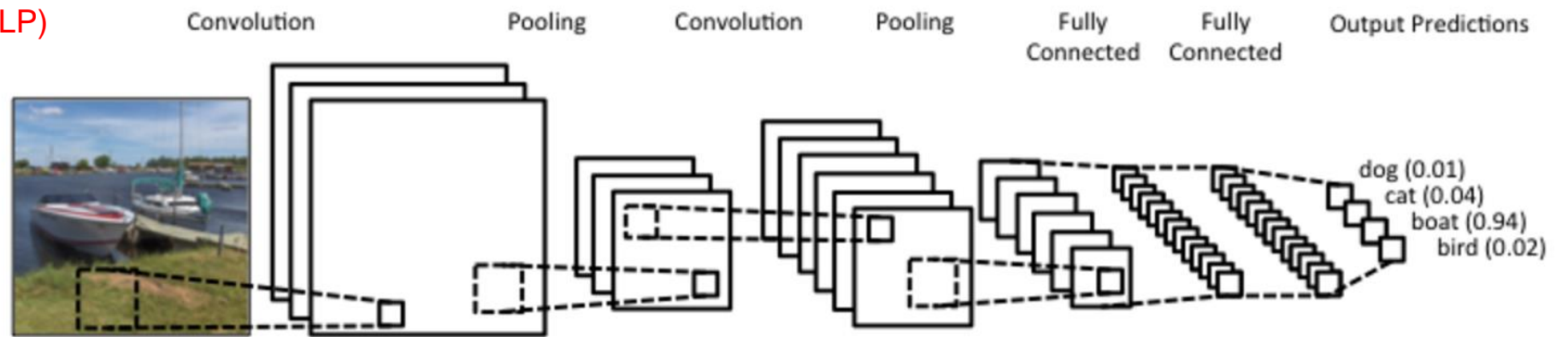
Different Architectures



Multi-layer Perceptron (MLP)



Recurrent Neural Network

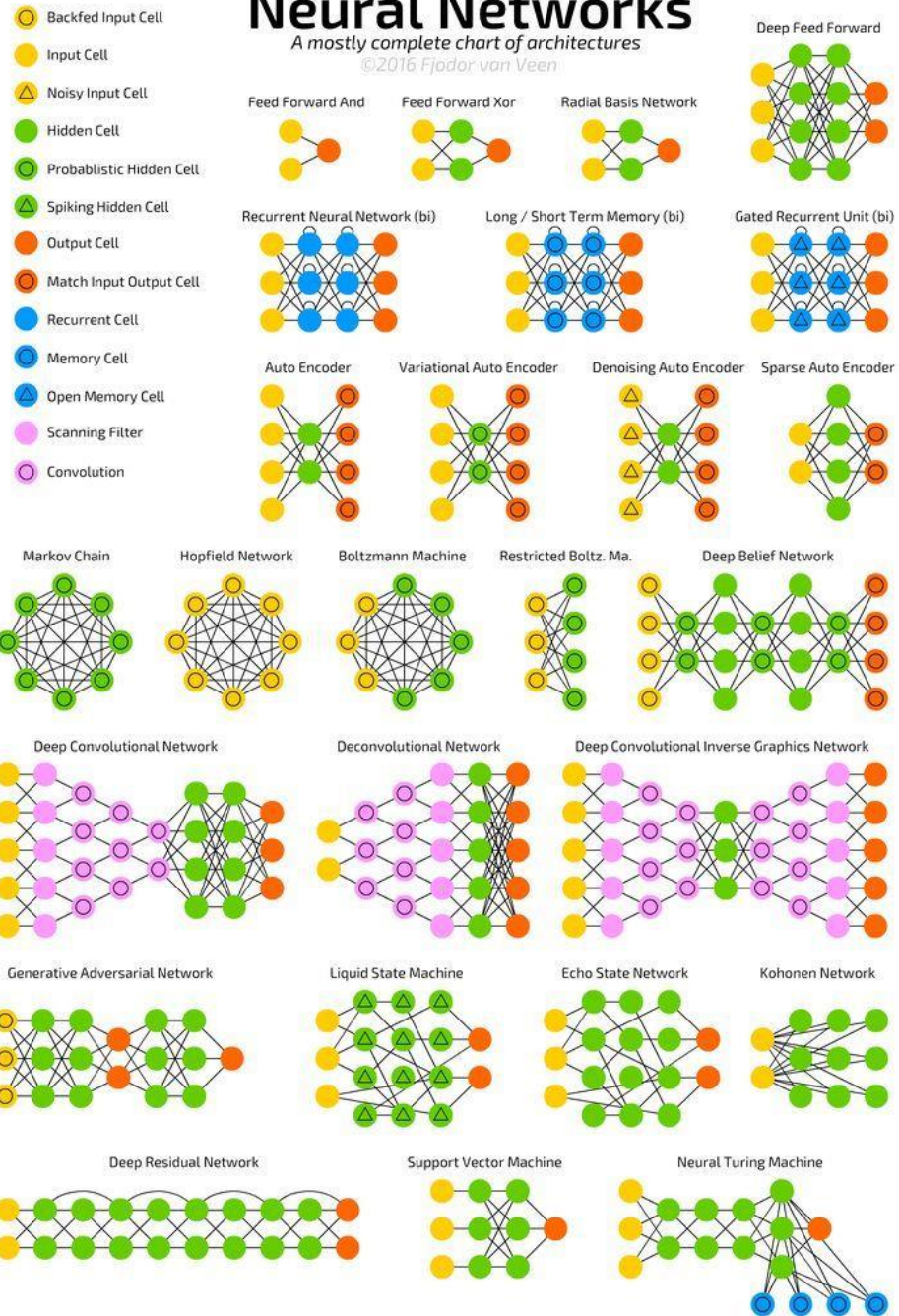


Convolutional Neural Network

Neural Networks

A mostly complete chart of architectures

©2016 Fjodor van Veen



Applications

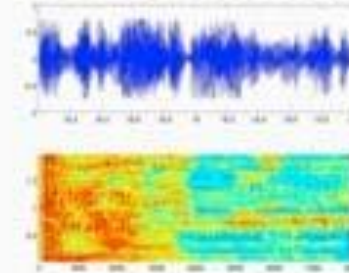
Images & Video



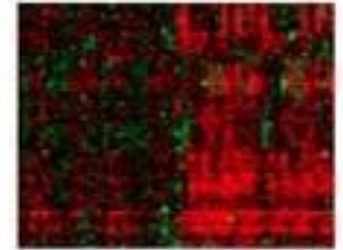
Text & Language



Speech & Audio



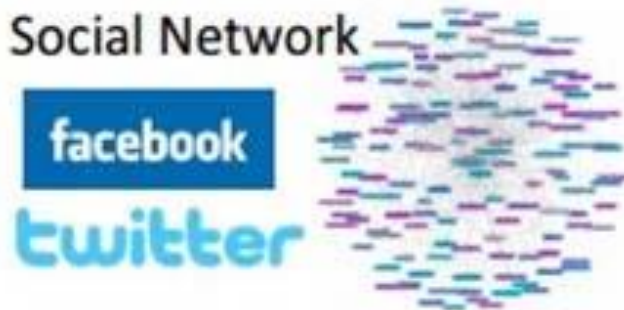
Gene Expression



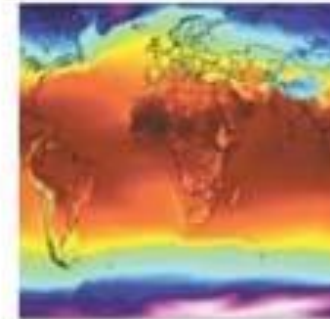
Product Recommendation



Relational Data/
Social Network



Climate Change



Geological Data



Outline

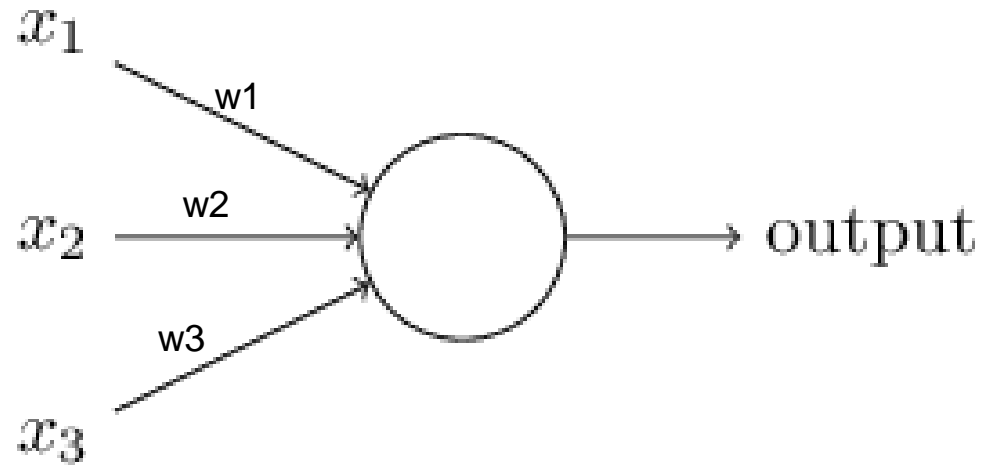
- A brief history
- Basic mathematics in Neural Networks (NN)
- Neural Machine Translation (NMT)
- Advanced NMT

Basic Mathematics in Neural Networks

- Artificial Neuron
- Activation Function
- Architecture
- Gradients
- Backpropagation (BP)
- Cost Function
- Overfitting, Regularization, Weight Initialization, Learning rate, early stopping
- Gradient Vanishing Problem

Perceptron: Artificial Neuron

- Was developed in the 1950s and 1960s by the scientist Frank Rosenblatt



- A perceptron takes several binary **inputs**, x_1, x_2, \dots
- Produces a single binary output
- Rosenblatt introduced **weights**, w_1, w_2, \dots , real numbers expressing the **importance** of the respective inputs to the output.


How a perceptron works

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$$

- The neuron's output, **0 or 1**, is determined by whether the weighted sum $\sum_j w_j x_j$ is less than or greater than some *threshold value*.
- The threshold is a real number which is a parameter of the neuron.

Vectorisation

- Input: x_1 , x_2 and x_3 .
- Weights: w_1 , w_2 and w_3 .
- Operation: summation over products.

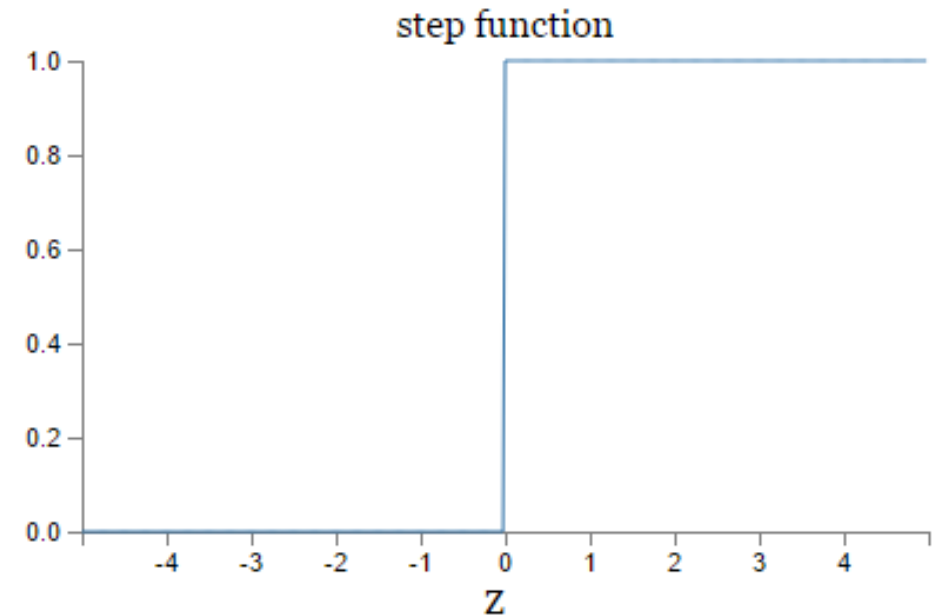

$$\mathbf{x} = [x_1, x_2, x_3],$$
$$\mathbf{w} = [w_1, w_2, w_3]$$

- Performing an operations to an entire set of values at once.
- Sets are seen as vectors.
- Reduces the complexity.

Simplification of Perceptrons

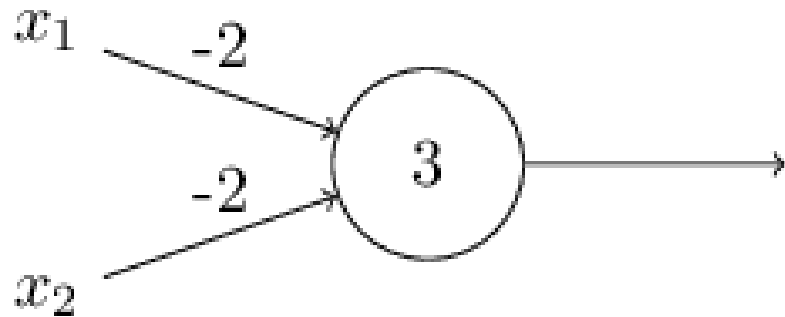
$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

- **Dot product**: $\mathbf{w} \cdot \mathbf{x} \equiv \sum_j w_j x_j$
- The perceptron's **bias**, $b \equiv -\text{threshold}$
- $\sigma(\mathbf{z}) = 0$, if $\mathbf{z} \leq 0$ or $\sigma(\mathbf{z}) = 1$ is $\mathbf{z} > 0$,
where $z = \mathbf{w} \cdot \mathbf{x} + b$
- σ – activation function;
step-like shape.



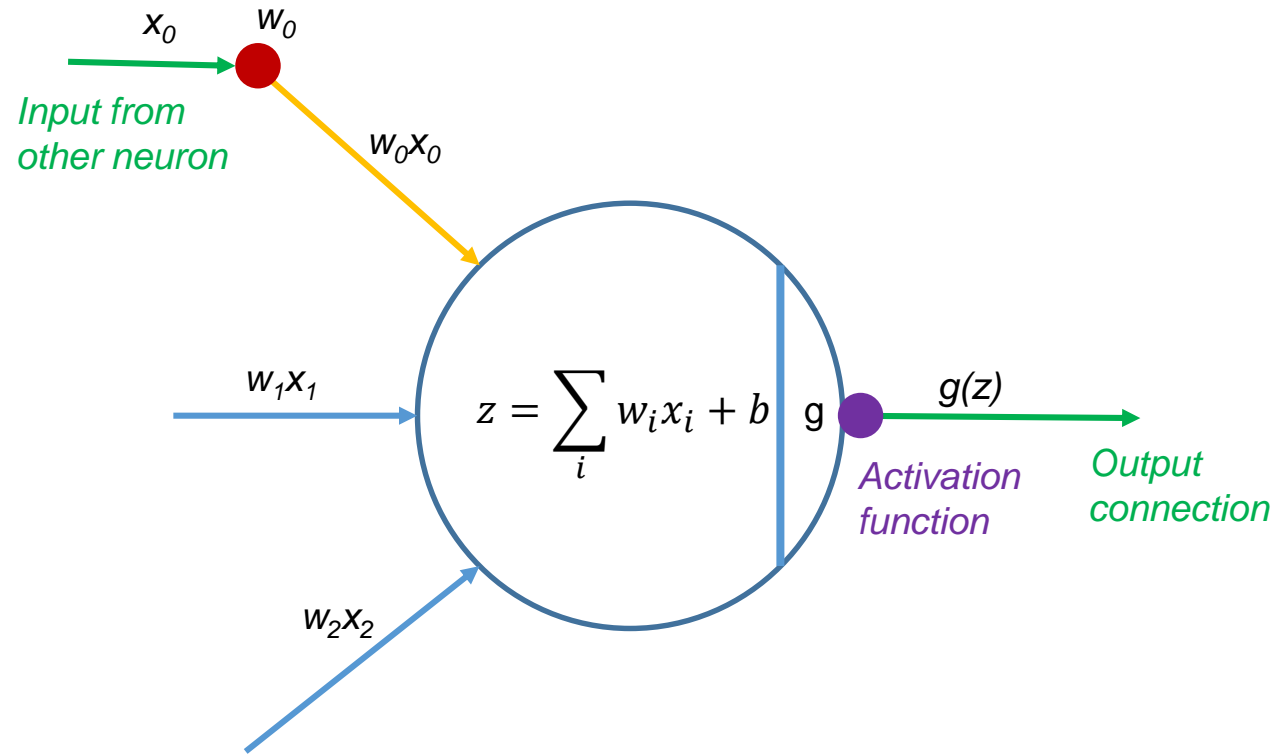
Applications of Perceptrons: logical functions

- Try inputs: 00, 01, 10 and 11
- What is this logical function?

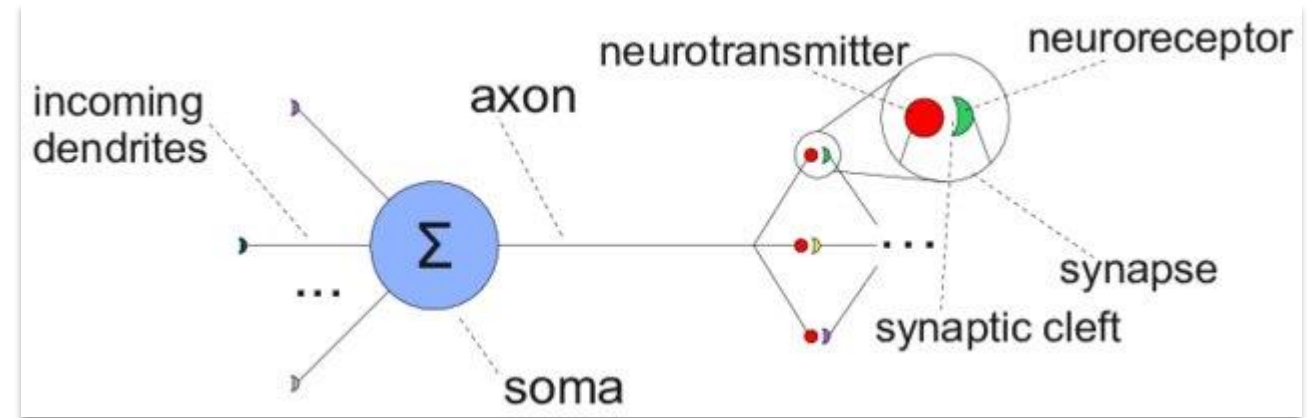
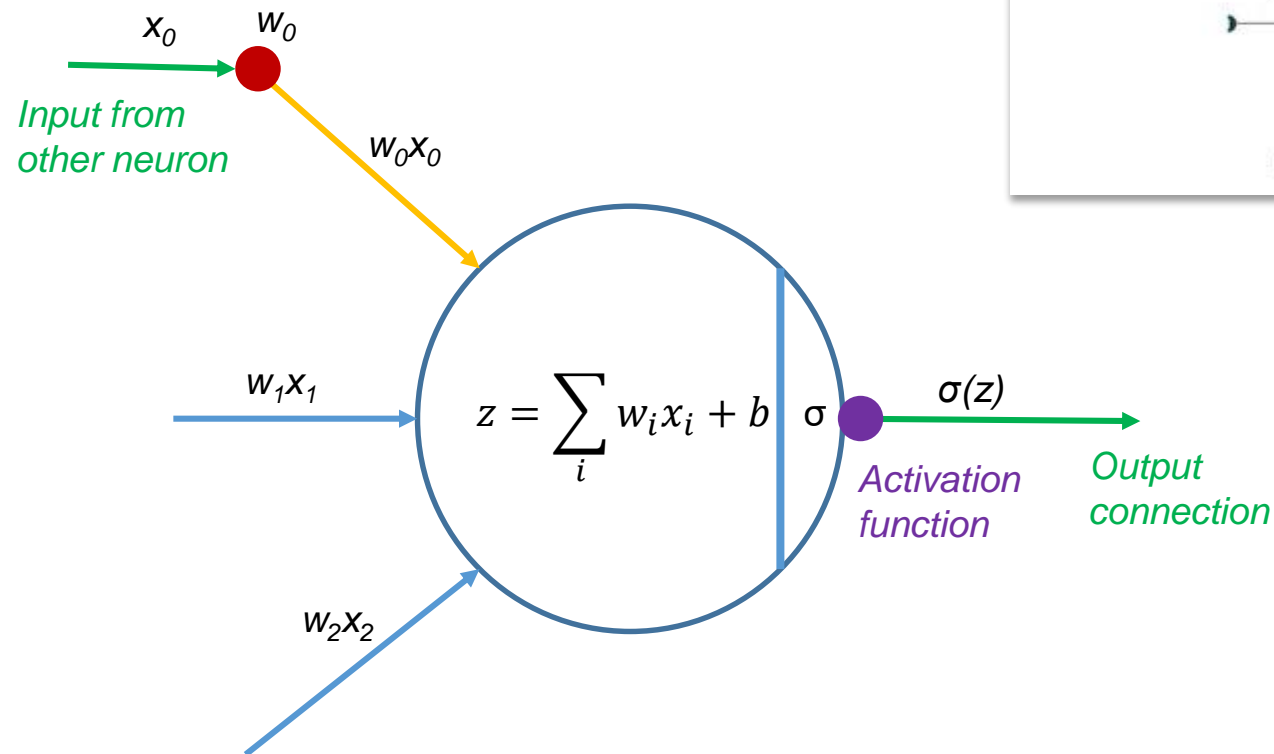


NAND Gate

General artificial neuron



General artificial neuron



- x_i takes on any values between 0 and 1
- The output is not 0 or 1. Instead, it's $\sigma(z) = \sigma(w \cdot x + b)$
- Different activation function => different performance

From a Perceptron to a Sigmoid Neuron

- Problems of Perceptrons:
 - Hard to learn - hard to tune the weights for an expected small change in outputs due to a small change in the weights or bias
 - We expect a gradual modification of the weights and biases so that the network gets closer to the desired behaviour
- **Sigmoid** Neurons:
 - Small changes in their weights and bias cause only a small change in their output
 - They can learn!

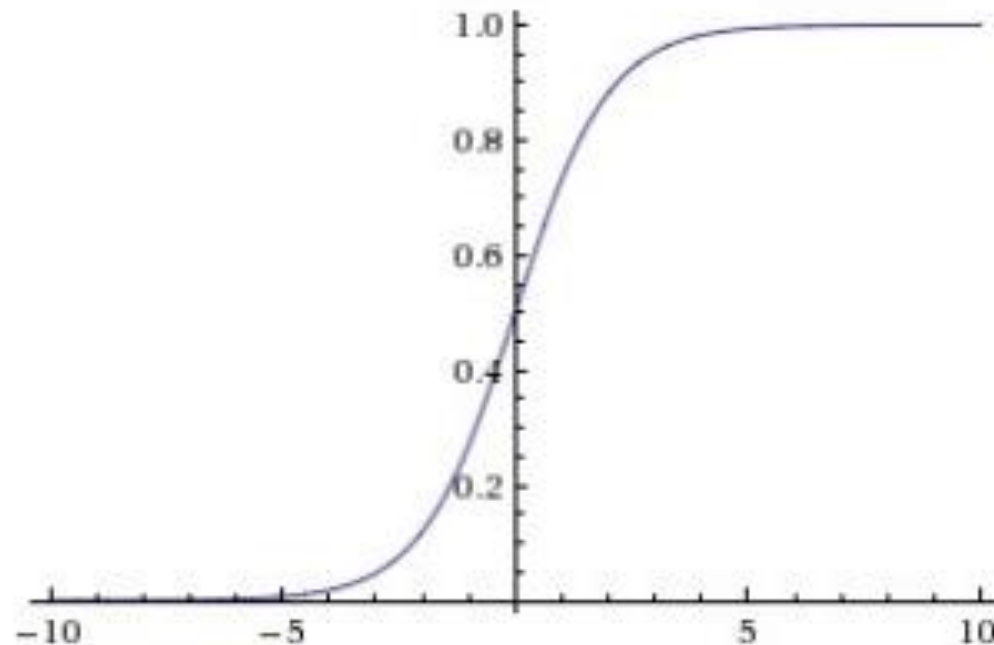
Activation Function

- The activation function of a node defines the output of that node given an input or set of inputs.
- It is also called transfer function
- Properties:
 - Nonlinear: a three layer-NN can be proven to be a universal function approximator
 - Continuously differentiable: necessary for enabling gradient-based optimization methods
 - Monotonic: the error surface associated with a single hidden layer model is guaranteed to be convex.

Activation Function: Sigmoid Function

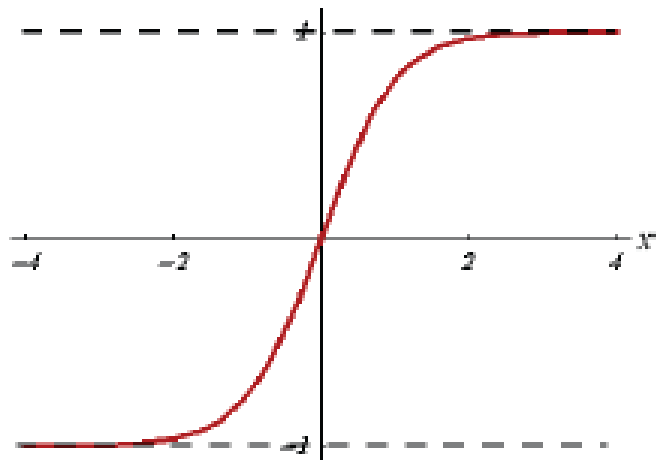
$$\sigma(z) \equiv \frac{1}{1 + e^{-z}} = \frac{1}{1 + \exp(-\sum_j w_j x_j - b)}.$$

Shape:

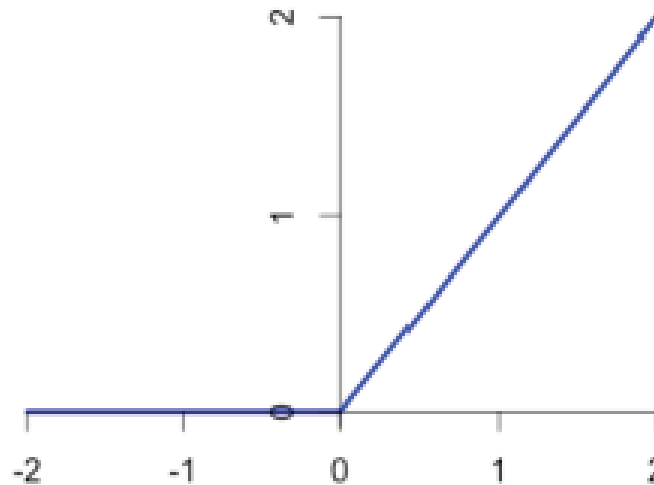


Other Commonly-used Activation Functions

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$






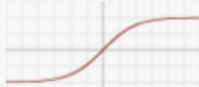
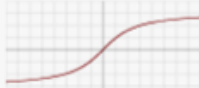




$$\text{relu}(x) = \max(0, x)$$



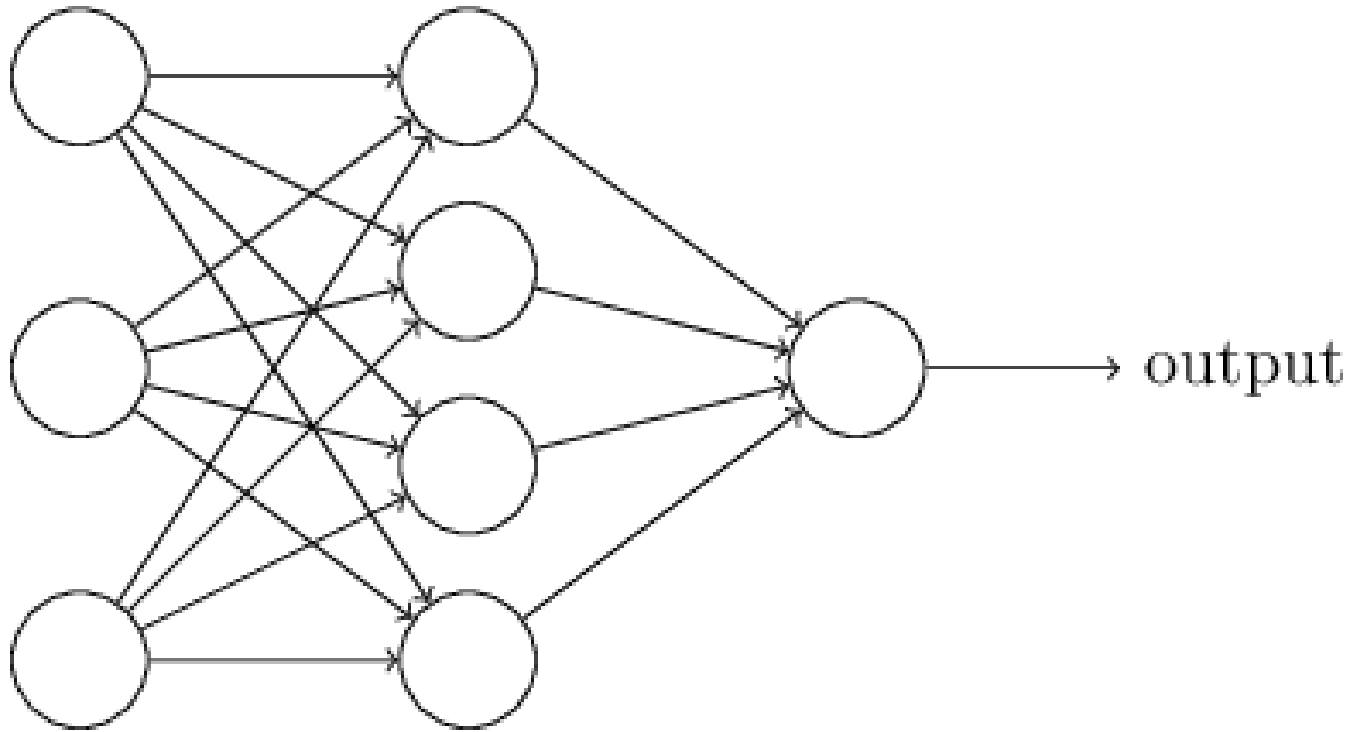
$$\text{softmax}(x)_i = \frac{e^{x_i}}{\sum e^{x_i}}$$

Useful for modeling probability (in classification task)

Activation functions list

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
Tanh		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parametric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$

From a Neuron to Neural Networks: Architecture

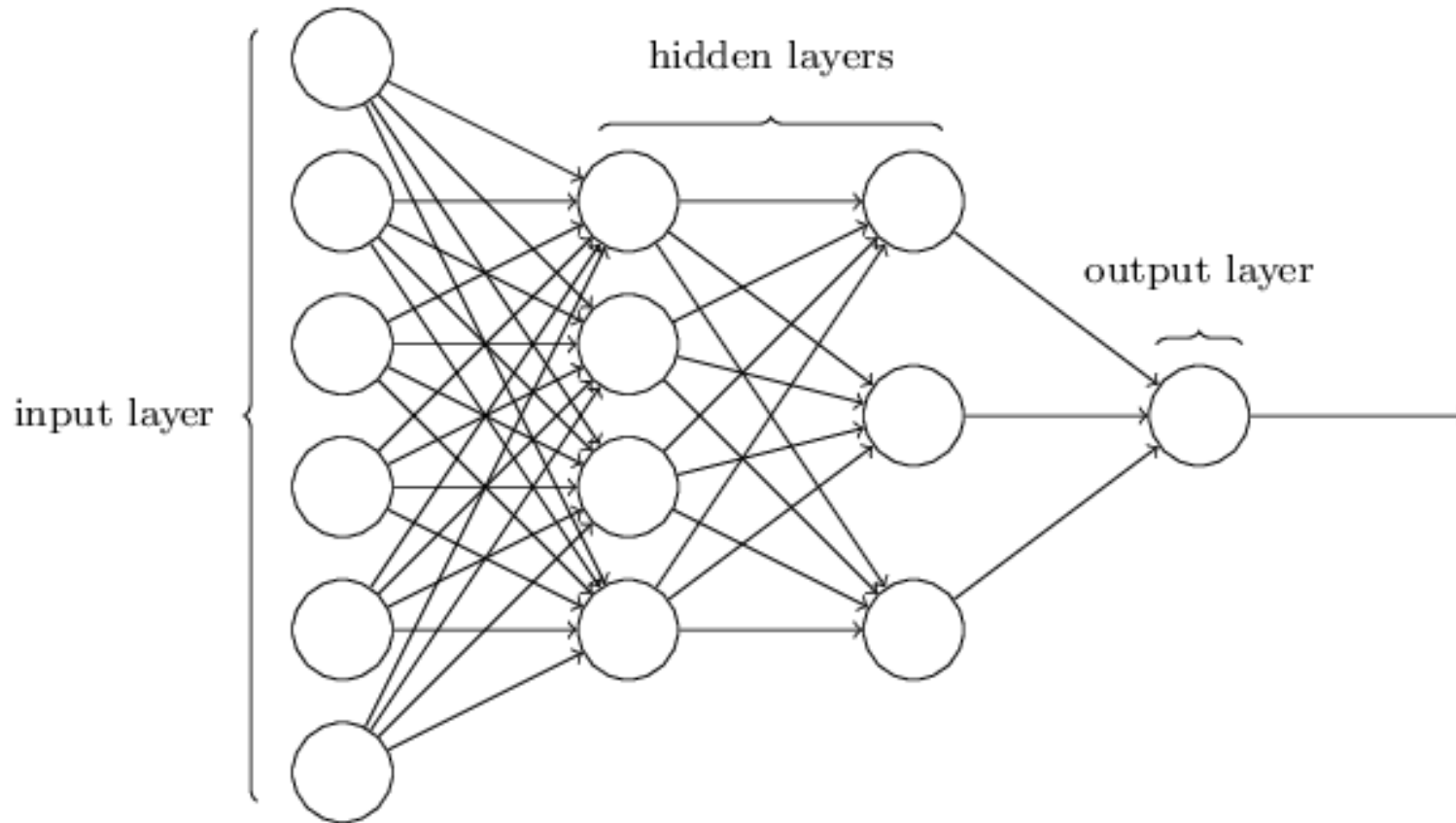


Input layer hidden layer output layer

- **Input layer**: input neurons
- **Output layer**: output neurons
- **Hidden layer**: in the middle
- **Neurons**: fully connected

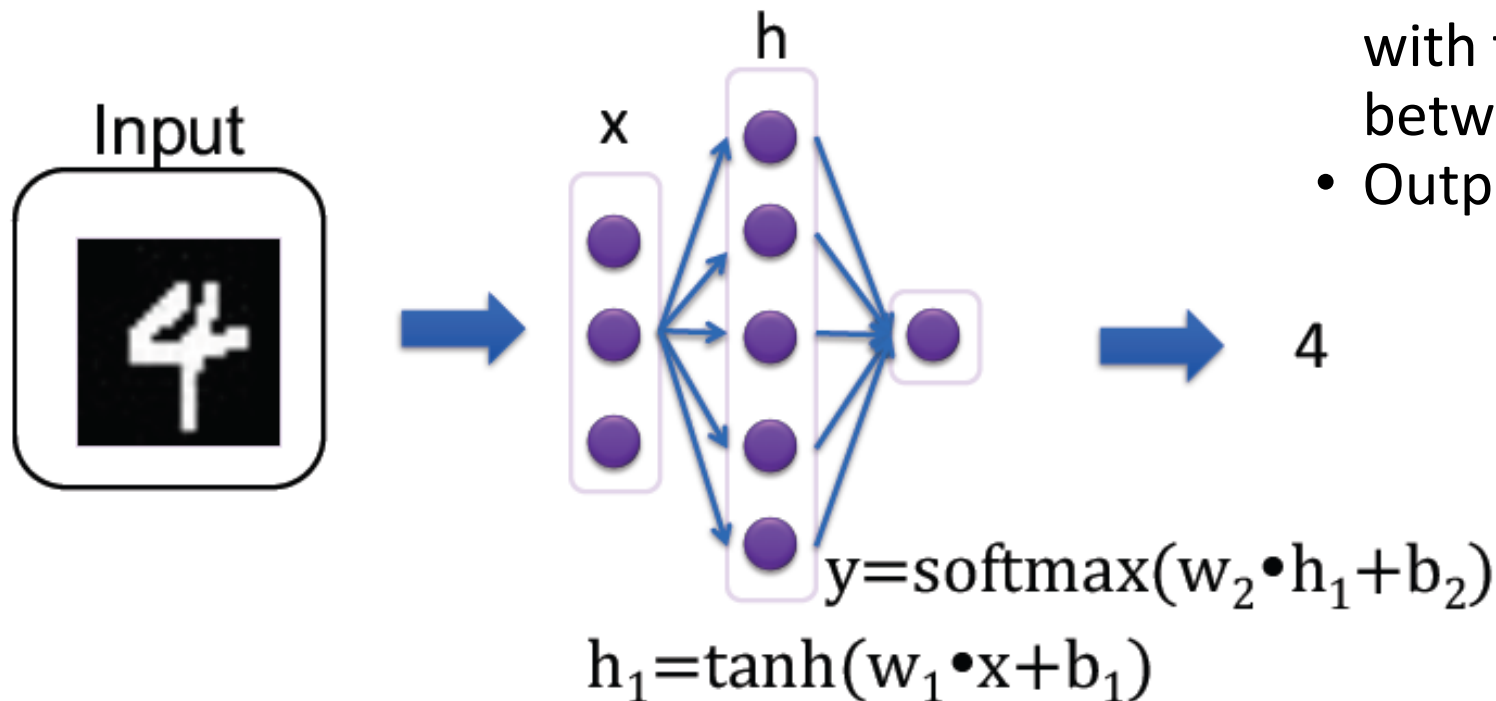
Multi-layer Perceptron: MLP

[Feed-forward Neural Networks]



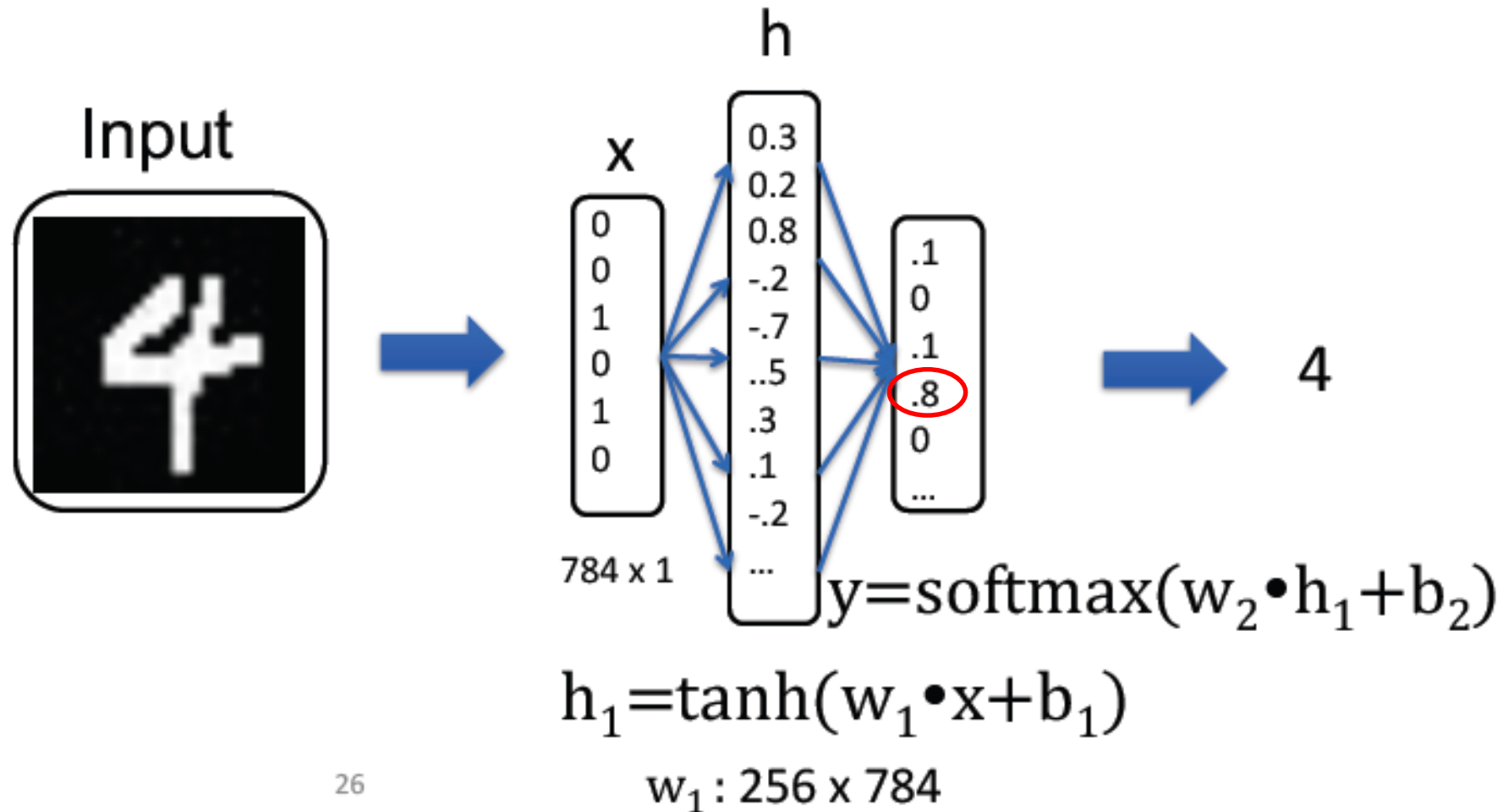
- There are no loops in the network
- Information is always fed forward, never fed back
- Neurons are fully connected

Examples: Handwritten digits recognition using Neural Nets

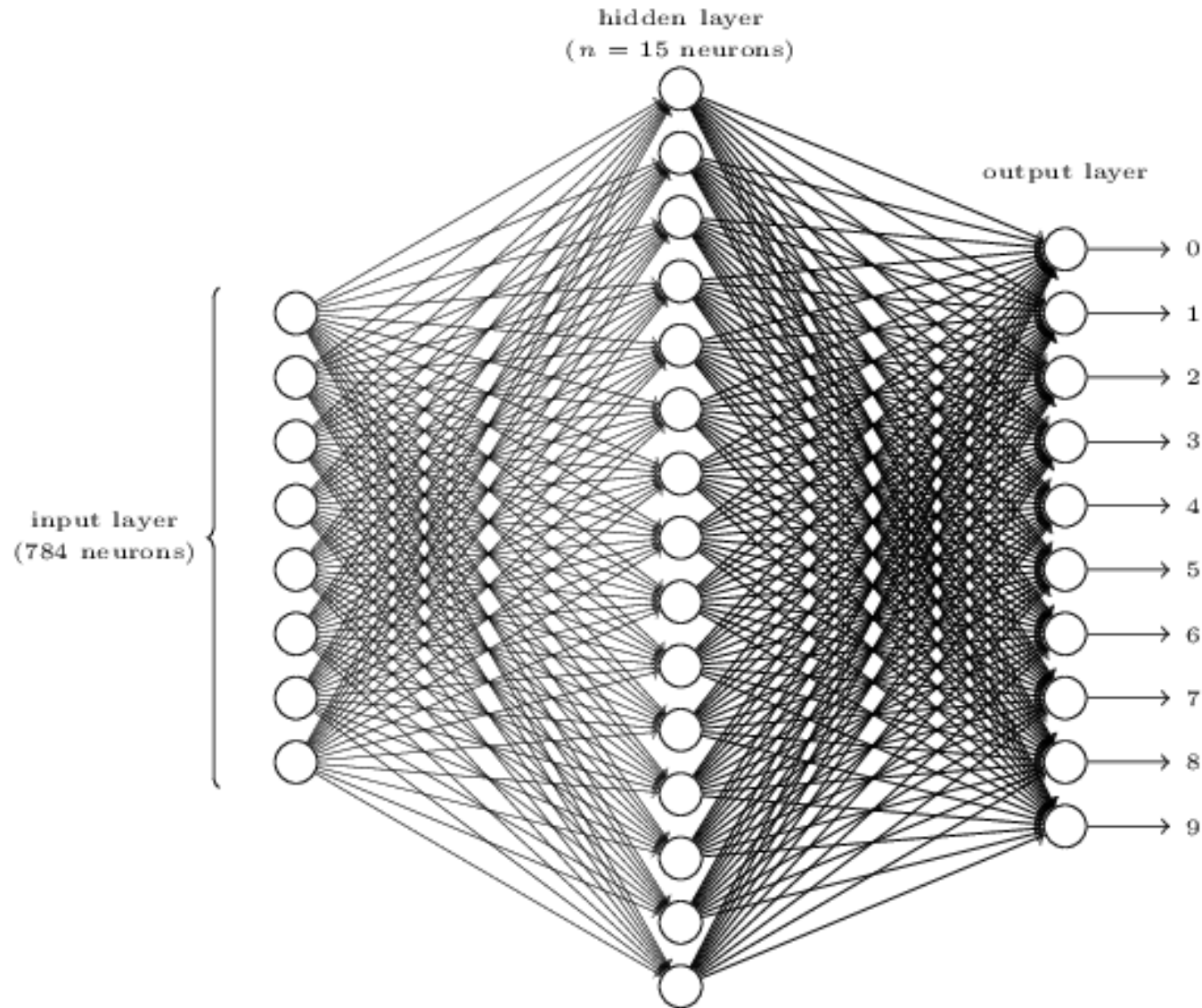


- Encode the intensities of the image pixels into the input neurons
- If the image is a 28×28 grey scale image, then we'd have $784 = 28 \times 28$ input neurons, with the intensities scaled appropriately between 0 and 1
- Output layer: <0.5 is not '4' and ≥ 0.5 is '4'

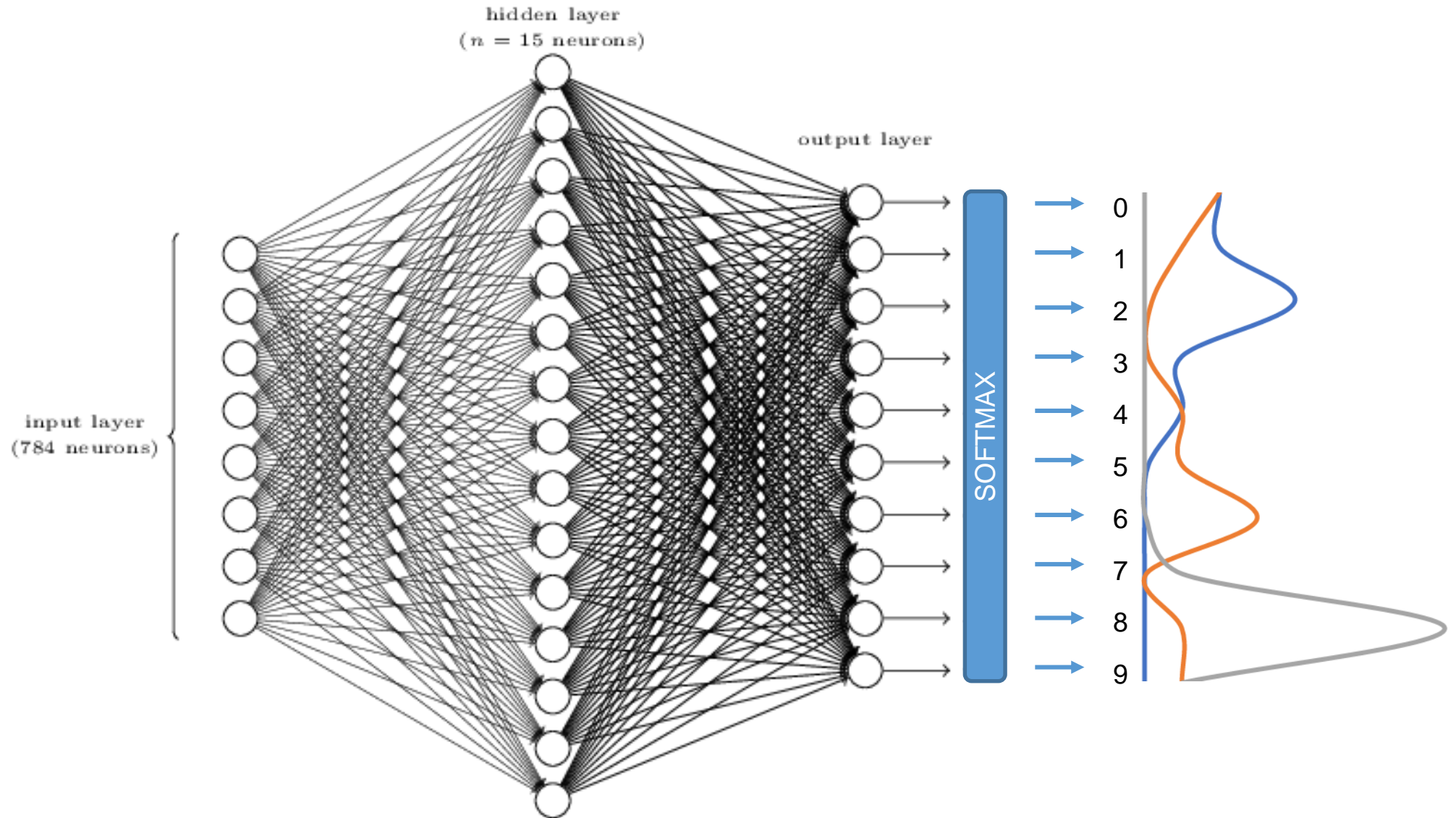
Numerical Example



A Real Neural Nets for Handwritten Digits



A Real Neural Nets for Handwritten Digits + a softmax layer



Learning for NNs: Cost Function

- We need an **algorithm** that
- can find **proper weights** and **biases** so that we have correct outputs for the training examples
- **Input**: \mathbf{x}
- **Expected output**: $y(\mathbf{x})$
- **Output of network**: a
- **Objective/Loss/Cost Function**: the aim of our training algorithm will be to minimize the cost

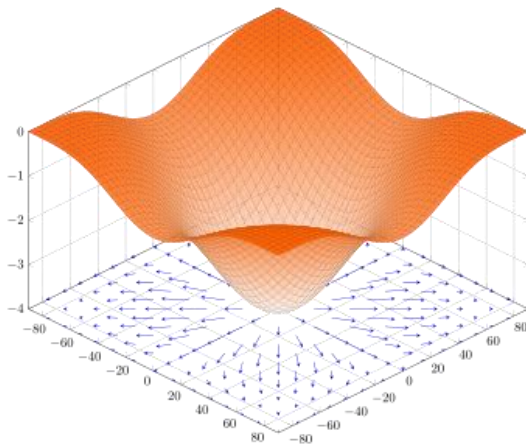
$$C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2.$$

Quadratic cost
function or mean
squared error
(MSE)

Minimum and gradient

•Gradient:

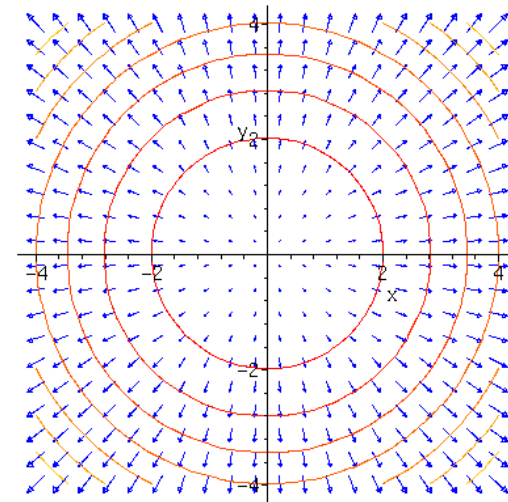
- A way of 'packing together' partial derivatives of a function
- A multi-variable generalization of the derivative
- the gradient is a vector-valued function, as opposed to a derivative, which is scalar-valued
- denoted as ∇f where ∇ (the nabla symbol) denotes the vector differential operator, del.



$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n} \right)$$

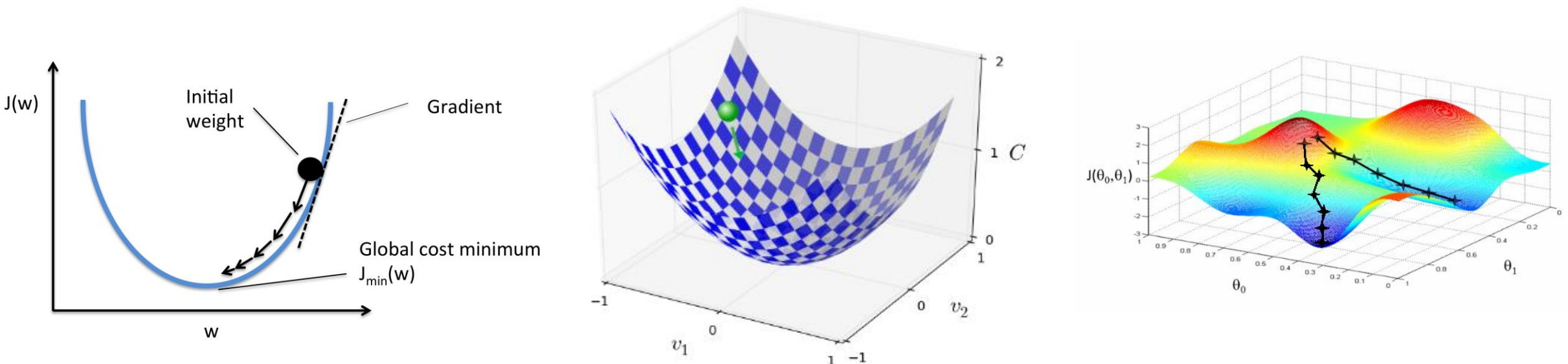
•Function optimisation:

- The gradient points at the direction of steepest ascent.
- The length of the gradient tells you the steepness of the direction.
- Maximum – follow the direction of the gradient, minimum – follow the opposite direction.
- Algorithm: *gradient descent*



Gradient Descent

- **Goal**: training a neural network is to find weights and biases which minimize the quadratic cost function $C(w, b)$.
- “**Minimize**” indicates that C achieves its **global minimum**.
- To illustrate how to minimize the cost C , let's imagine C as a function of just two variables $v_1, v_2 : C(v_1, v_2)$ and the shape of C is as:



Gradient Descent

- If we move a small amount Δv_1 in the v_1 direction, and a small amount Δv_2 in the v_2 direction, Calculus tells us that C changes as:

$$\Delta C \approx \frac{\partial C}{\partial v_1} \Delta v_1 + \frac{\partial C}{\partial v_2} \Delta v_2.$$

- The goal is to choose Δv_1 and Δv_2 to make ΔC negative
- Define:
 - $\Delta v \equiv (\Delta v_1, \Delta v_2)^T$ the vector of changes
 - Gradient of C : *the vector of partial derivatives*

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2} \right)^T.$$

Gradient Descent

- ΔC can be rewritten as

$$\Delta C \approx \nabla C \cdot \Delta v.$$

- How to choose Δv so as to make ΔC negative?
- Suppose we choose:

$$\Delta v = -\eta \nabla C,$$

η is a small, positive parameter (known as the learning rate).

- Then we have

$$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2.$$

Gradient Descent

- We can update v following the above ideas:

$$v \rightarrow v' = v - \eta \nabla C.$$

- The way the gradient descent algorithm works is to repeatedly compute the gradient ∇C , and then to move in the opposite direction to the global minimum.
- Gradient descent can be viewed as a way of taking small steps in the direction which does the most to immediately decrease C .

Learning Rate

- To make gradient descent work correctly, we need to choose the **learning rate** η to be **small enough** that $\Delta C \approx \nabla C \cdot \Delta v$ is a good approximation.
- However, we don't want η to be too small, since that will make the changes Δv tiny, and thus the gradient descent algorithm will work very slowly.
- In practical implementations, η is often varied.

From Two Variables to Many Variables

$$C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2.$$

- C is a function of m variables, v_1, \dots, v_m
- Then the change ΔC in C produced by a small change $\Delta v = (\Delta v_1, \dots, \Delta v_m)^T$ is

$$\Delta C \approx \nabla C \cdot \Delta v,$$

where the gradient ∇C is the vector

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \dots, \frac{\partial C}{\partial v_m} \right)^T.$$

Gradient Descent for Neural Networks

- Use gradient descent to find the weights w_k and biases b_l which minimize the cost in Cost Function
- Repeatedly update weights and biases as

$$w_k \rightarrow w'_k = w_k - \eta \frac{\partial C}{\partial w_k}$$

$$b_l \rightarrow b'_l = b_l - \eta \frac{\partial C}{\partial b_l}.$$

Learning Problems in Gradient Descent

- Suppose we have n training examples
- For each example x , the quadratic (MSE) cost is:

$$C_x \equiv \frac{\|y(x) - a\|^2}{2}$$

- The cost over all training examples is:

$$C = \frac{1}{n} \sum_x C_x$$

- The gradient would be:

$$\nabla C = \frac{1}{n} \sum_x \nabla C_x.$$

If n is very large, then this will take a long time and learning is very slowly.

Stochastic Gradient Descent (SGD)

- Idea:
 - Estimate the gradient ∇C by computing ∇C_x for a small sample of randomly chosen training inputs - mini-batch
 - By averaging over this small sample we can quickly get a good estimate of the true gradient ∇C

$$\frac{\sum_{j=1}^m \nabla C_{x_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C,$$

- m is the number of randomly chosen training inputs

SGD for Neural Nets

$$w_k \rightarrow w'_k = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k}$$

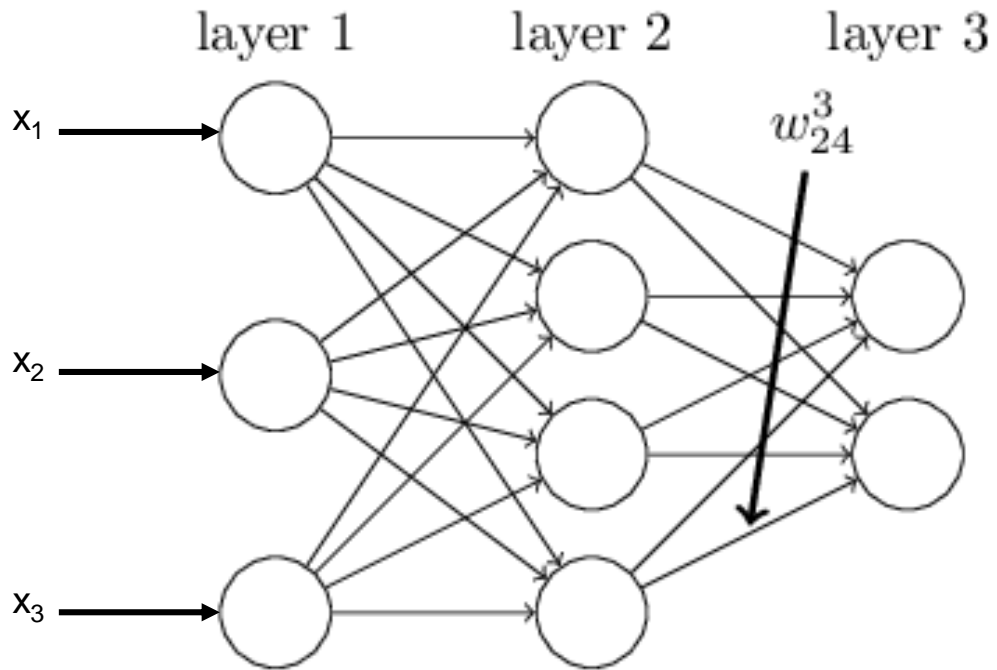
$$b_l \rightarrow b'_l = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l},$$

- The sums are over all the training examples X_j in the current mini-batch
- Then we pick out another randomly chosen mini-batch and train with those
- And so on, until we've exhausted the training inputs, which is said to complete an **epoch** of training.

Summary of SGD

- Significantly speed up the estimate of the gradient
- There will be statistical fluctuations
- However, it is moving in a general direction that will help decrease C , so we don't need an exact computation of the gradient
- It is a commonly used and powerful technique for learning in neural networks
- It is the basis for most of the learning techniques

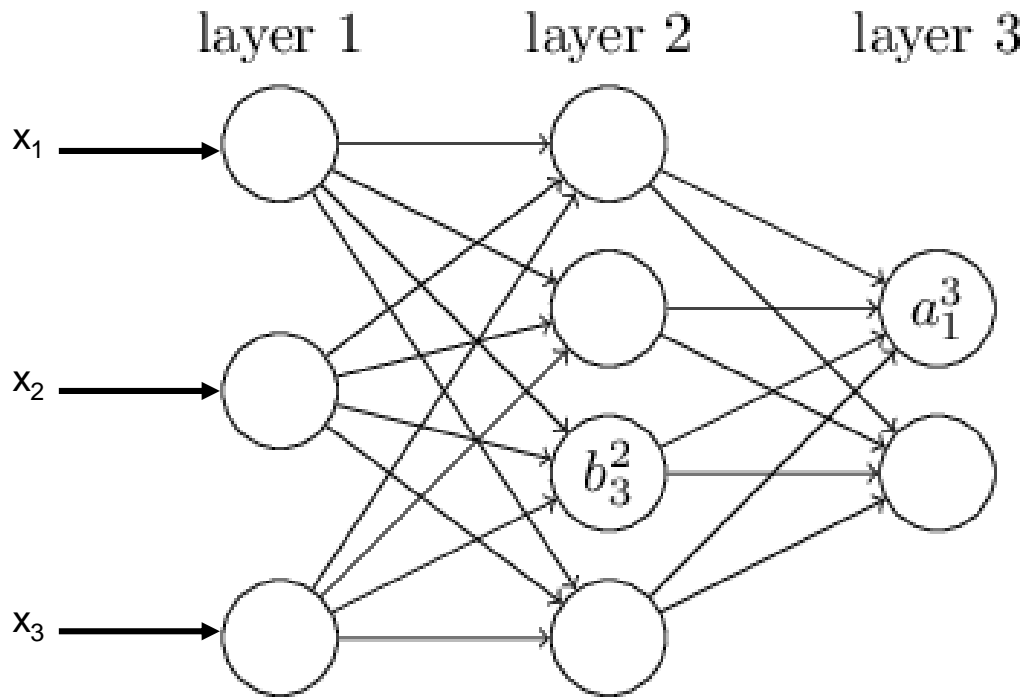
Some Notations: weights



w_{jk}^l is the weight from the k^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer

- w, b denote weights and biases
- l denotes layers
- j, k denote neurons
- w_{jk}^l denotes the weight for the connection from the k^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer.
- x_i – the i^{th} input

Some Notations: bias and activation



- b_j^l is the bias of the j^{th} neuron in the l^{th} layer.
- a_j^l is the activation of the j^{th} neuron in the l^{th} layer.
- For the forward propagation, we have

$$a_j^l = \sigma \left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l \right),$$

- The sum is over all neurons k in the $(l-1)^{\text{th}}$ layer.

Some Notations: Vectorized Form

- Weight matrix w^l for layer l
 - the weights connecting to the l^{th} layer of neurons
 - the entry in the j^{th} row and k^{th} column is w_{jk}^l
- Bias vector b^l for layer l
 - The entry in the j^{th} position is b_j^l
 - one component for each neuron in the l^{th} layer
- Activation vector a^l for layer l
 - The entry in the j^{th} position is a_j^l
 - one component for each neuron in the l^{th} layer
- Then we have the vectorized form

$$a^l = \sigma(w^l a^{l-1} + b^l)$$

More Notations: weighted Input

- Intermediate quantity: weighted input

$$z^l = w^l a^{l-1} + b^l$$

z^l is a matrix

- z_j^l is the weighted input to the activation function for neuron j in layer l :

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l,$$

- So we have $a^l = \sigma(z^l)$

More Notations: error

- intermediate quantity: error δ_j^l
 - the error in the j^{th} neuron in the l^{th} layer.
 - The error will be related to the partial derivatives $\partial C / \partial w_{jk}^l$ and $\partial C / \partial b_j^l$.
 - The error is defined as

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}$$

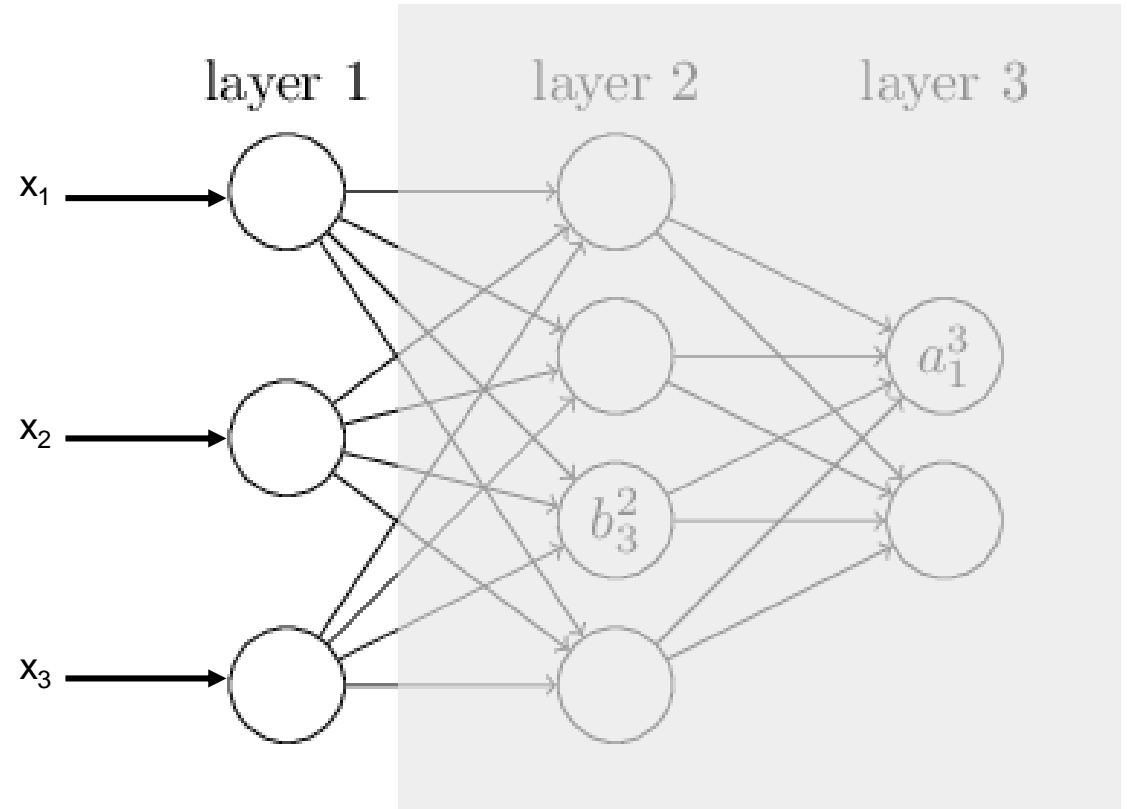
- In vector form, we have δ^l to denote the vector of errors associated with layer l .

Learning cycle in NNs

- Forward propagation
 - Propagate input through the network
- Backward propagation
 - Compute loss
 - Compute gradients
- Update weights based on (S)GD

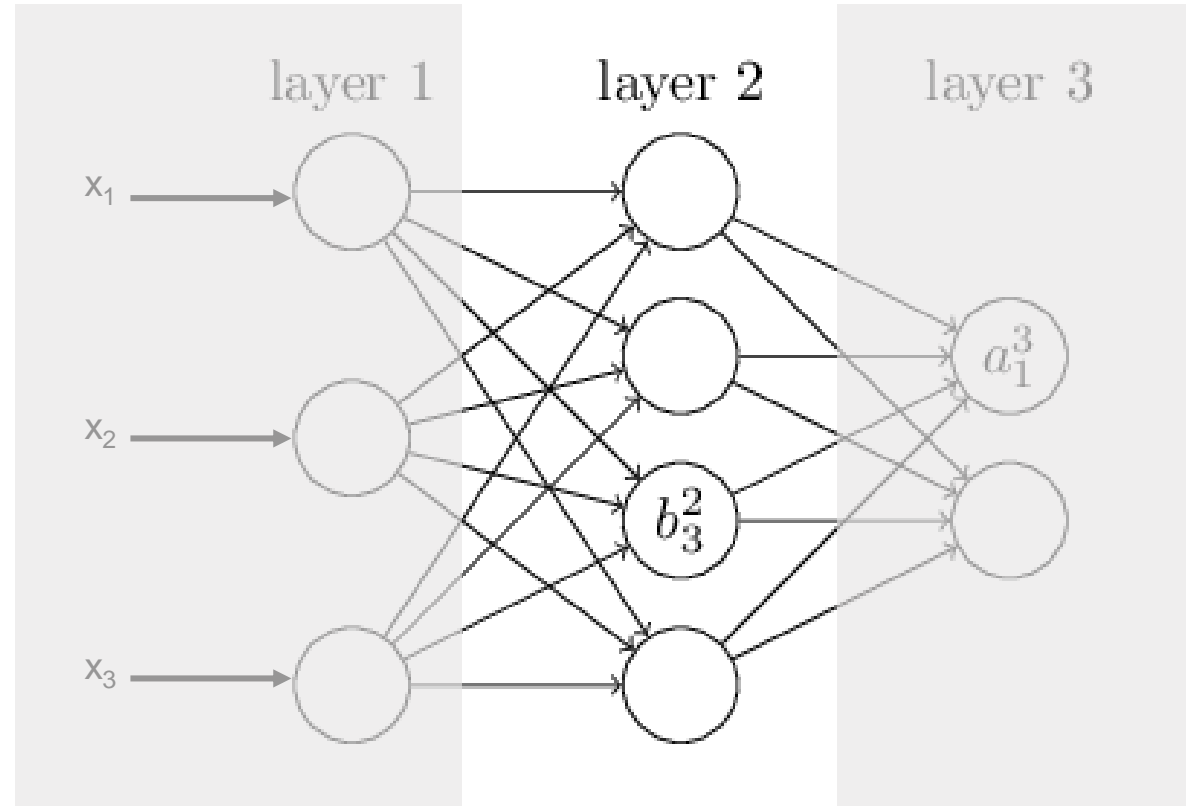
Forward propagation

- Propagate input through the network



Forward propagation

- Propagate input through the network

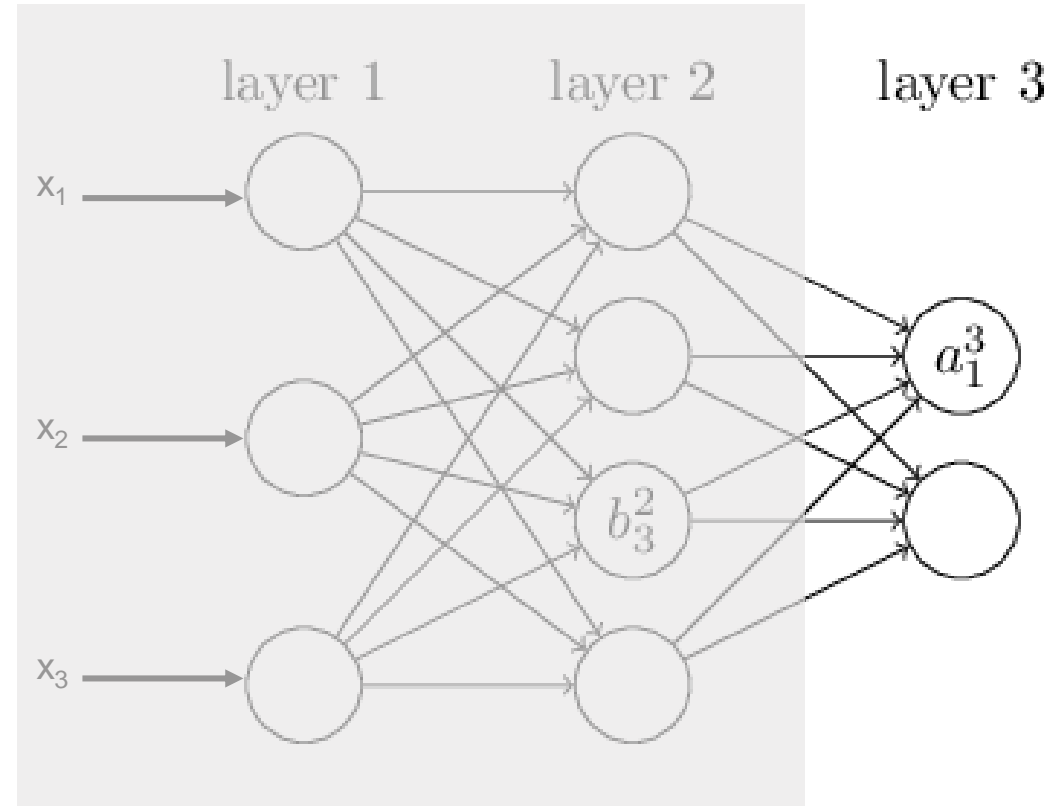


Forward propagation

- Propagate input through the network

$$z^l = w^l a^{l-1} + b^l$$

$$a^l = \sigma(z^l)$$



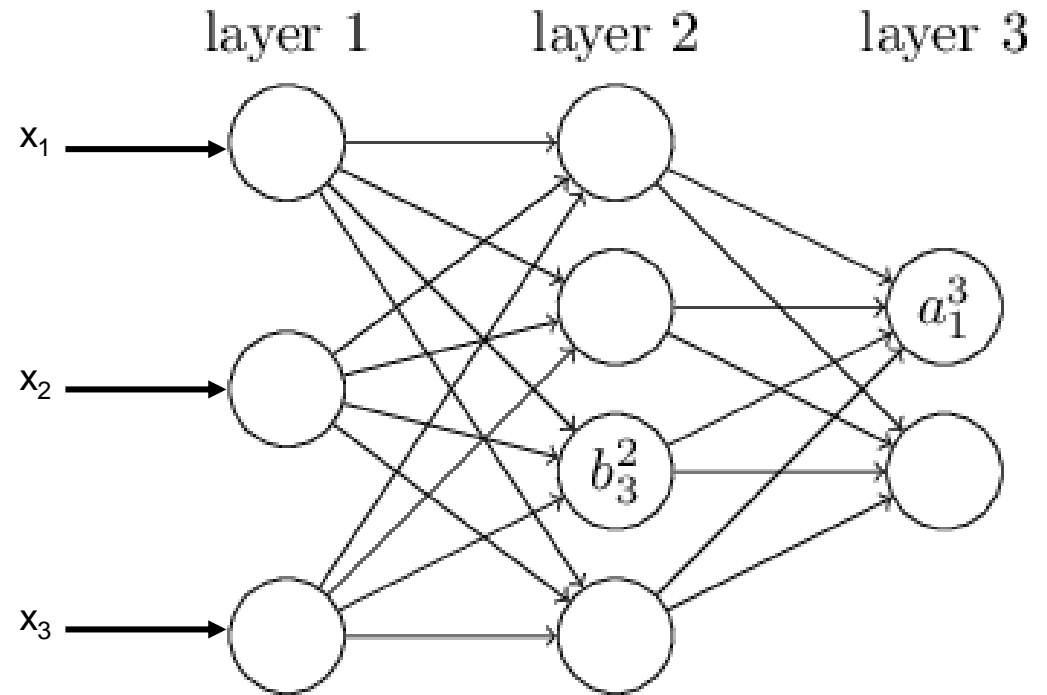
Forward propagation

- Propagate input through the network

```
for l in range(levels+1):
```

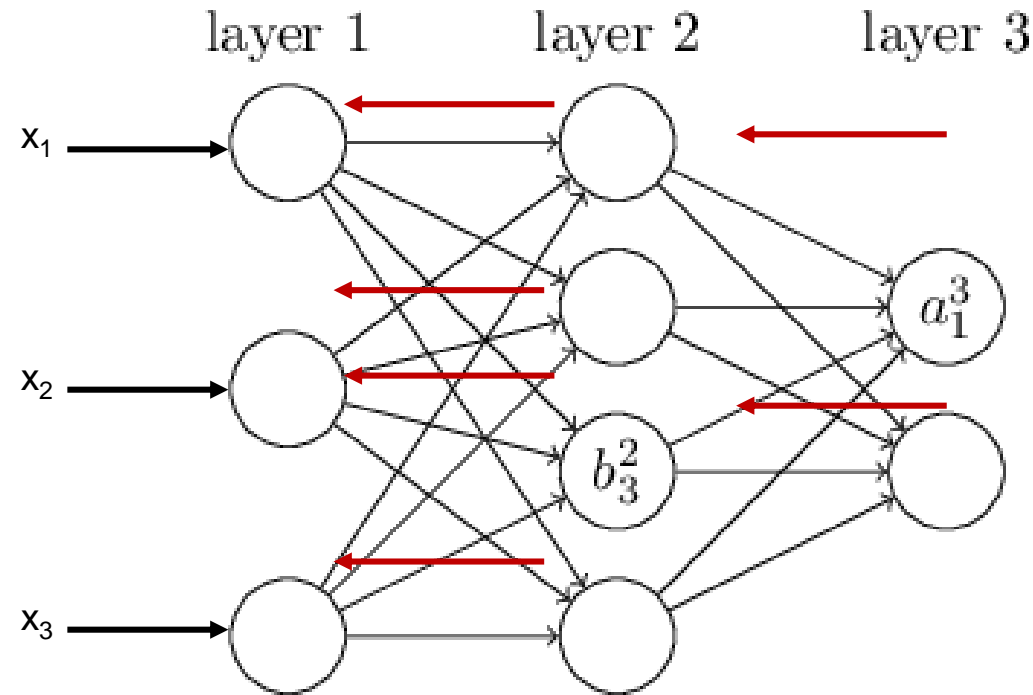
$$z^l = w^l a^{l-1} + b^l$$

$$a^l = \sigma(z^l)$$



Backpropagation:

A fast algorithm for computing gradients



Backpropagation:

A fast algorithm for computing gradients

- Was originally introduced in the 1970s
- But its importance wasn't fully appreciated until a famous [1986 paper](#) by David Rumelhart, Geoffrey Hinton, and Ronald Williams.
- That paper describes several neural networks where backpropagation works far faster than earlier approaches to learning, making it possible to use neural nets to solve problems which had previously been insoluble.
- Today, the backpropagation algorithm is the workhorse of learning in neural networks.

The equations of backpropagation

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \quad (\text{BP1})$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \quad (\text{BP2})$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \quad (\text{BP3})$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \quad (\text{BP4})$$

Backpropagation Algorithm

1. **Input x :** Set the corresponding activation a^1 for the input layer.
2. **Feedforward:** For each $l = 2, 3, \dots, L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$.
3. **Output error δ^L :** Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$.
4. **Backpropagate the error:** For each $l = L - 1, L - 2, \dots, 2$ compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$.
5. **Output:** The gradient of the cost function is given by $\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l$ and $\frac{\partial C}{\partial b_j^l} = \delta_j^l$.

BP Algorithm for SGD (mini-batch)

1. **Input a set of training examples**

2. **For each training example x :** Set the corresponding input activation $a^{x,1}$, and perform the following steps:

- **Feedforward:** For each $l = 2, 3, \dots, L$ compute

$$z^{x,l} = w^l a^{x,l-1} + b^l \text{ and } a^{x,l} = \sigma(z^{x,l}).$$

- **Output error $\delta^{x,L}$:** Compute the vector

$$\delta^{x,L} = \nabla_a C_x \odot \sigma'(z^{x,L}).$$

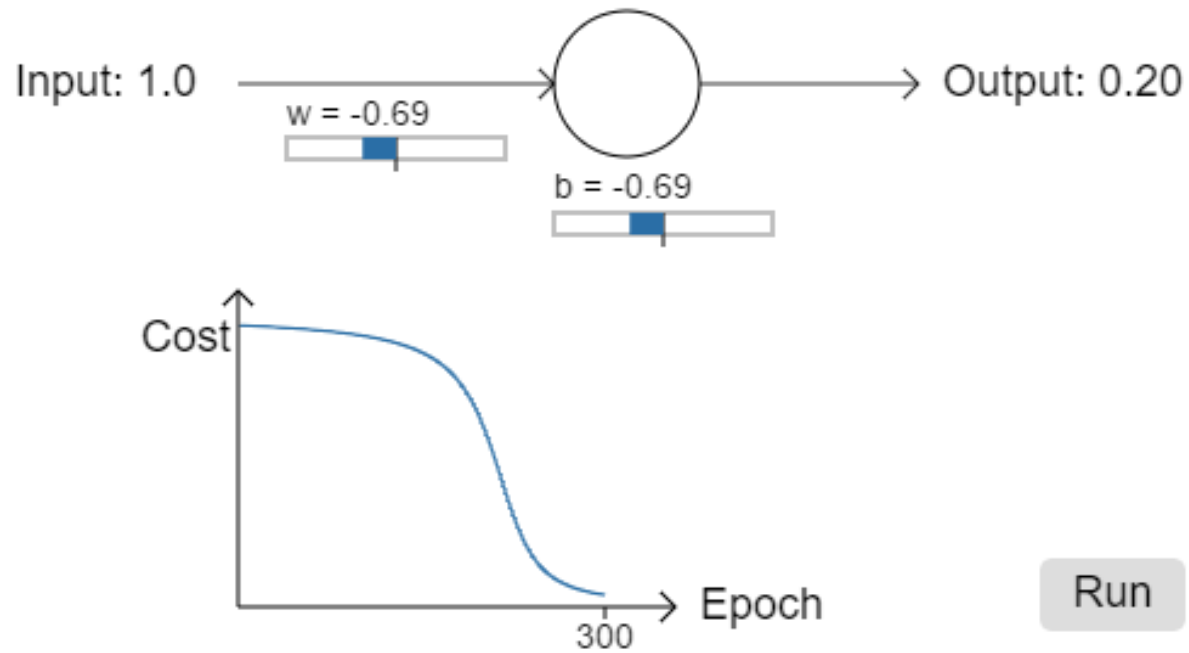
- **Backpropagate the error:** For each

$l = L - 1, L - 2, \dots, 2$ compute

$$\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l}).$$

3. **Gradient descent:** For each $l = L, L - 1, \dots, 2$ update the weights according to the rule $w^l \rightarrow w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$, and the biases according to the rule $b^l \rightarrow b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$.

Revisit: Cost Function



- quadratic cost function

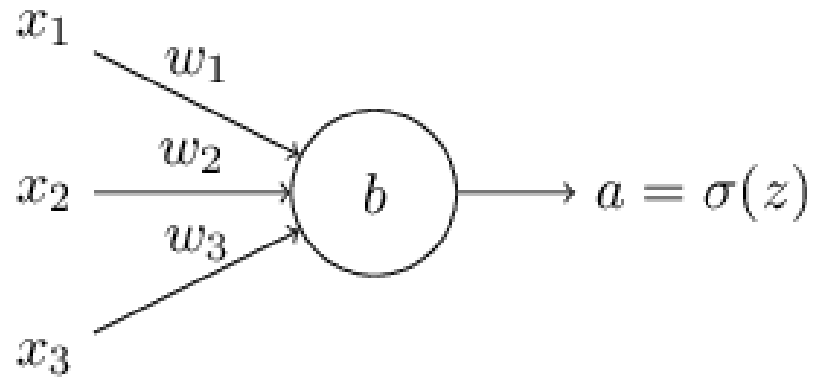
- learning is very slow, why?
- In this example, we have

$$\frac{\partial C}{\partial w} = (a - y)\sigma'(z)x = a\sigma'(z)$$

$$\frac{\partial C}{\partial b} = (a - y)\sigma'(z) = a\sigma'(z),$$

- when the neuron's output is close to 1, the curve of sigmoid gets very flat, and so $\sigma'(z)$ gets very small.

Cost Function: Cross-Entropy



$$z = \sum_j w_j x_j + b$$

- We define the cross-entropy cost function for this neuron by

$$C = -\frac{1}{n} \sum_x [y \ln a + (1 - y) \ln(1 - a)] ,$$

where n is the total number of items of training data, the sum is over all training inputs, x , and y is the corresponding desired output (in this toy example, $y = 0/y = 1$).

Cross-Entropy

- Two properties:
 - non-negative ($C > 0$)
 - if the neuron's actual output is close to the desired output for all training inputs, x , then the cross-entropy will be close to zero
- Cross entropy can avoid the problem of learning slowing down, why?
 - We substitute $a=\sigma(z)$, then we compute the partial derivative with respect to weights and apply the chain rule as:

$$C = -\frac{1}{n} \sum_x [y \ln a + (1 - y) \ln(1 - a)],$$

$$\begin{aligned} \frac{\partial C}{\partial w_j} &= -\frac{1}{n} \sum_x \left(\frac{y}{\sigma(z)} - \frac{(1 - y)}{1 - \sigma(z)} \right) \frac{\partial \sigma}{\partial w_j} \\ &= -\frac{1}{n} \sum_x \left(\frac{y}{\sigma(z)} - \frac{(1 - y)}{1 - \sigma(z)} \right) \sigma'(z) x_j. \\ &= \frac{1}{n} \sum_x \frac{\sigma'(z) x_j}{\sigma(z)(1 - \sigma(z))} (\sigma(z) - y). \end{aligned}$$

Cross-Entropy

- sigmoid function: $\sigma(z) = 1/(1 + e^{-z})$
- and its derivative is: $\sigma'(z) = \sigma(z)(1 - \sigma(z))$.

- Then we have

$$\frac{\partial C}{\partial w_j} = \frac{1}{n} \sum_x x_j (\sigma(z) - y).$$

- the rate at which the weight learns is controlled by $\sigma(z)-y$, i.e., by the error in the output.
- The larger the error, the faster the neuron will learn.
- Similarly, for the bias, the partial derivative is

$$\frac{\partial C}{\partial b} = \frac{1}{n} \sum_x (\sigma(z) - y).$$

Cross Entropy in Practice

- many-neuron multi-layer networks

$$C = -\frac{1}{n} \sum_x \sum_j \left[y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L) \right] .$$

Softmax function

- It defines a new type of output layer for our neural networks to address the learning slowdown problem
- the activation a_j^L of the j^{th} output neuron is

$$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}},$$

in the denominator we sum over all the output neurons.

$$\sum_j a_j^L = \frac{\sum_j e^{z_j^L}}{\sum_k e^{z_k^L}} = 1.$$

the output from the softmax layer can be thought of as a probability distribution

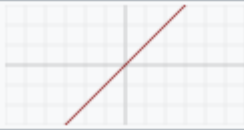


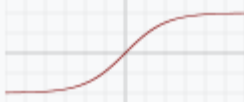



Log-likelihood Cost Function

- Definition: $C \equiv -\ln a_y^L.$
- How can log-likelihood avoid the learning slowdown problem?

$$\frac{\partial C}{\partial b_j^L} = a_j^L - y_j$$
$$\frac{\partial C}{\partial w_{jk}^L} = a_k^{L-1} (a_j^L - y_j)$$

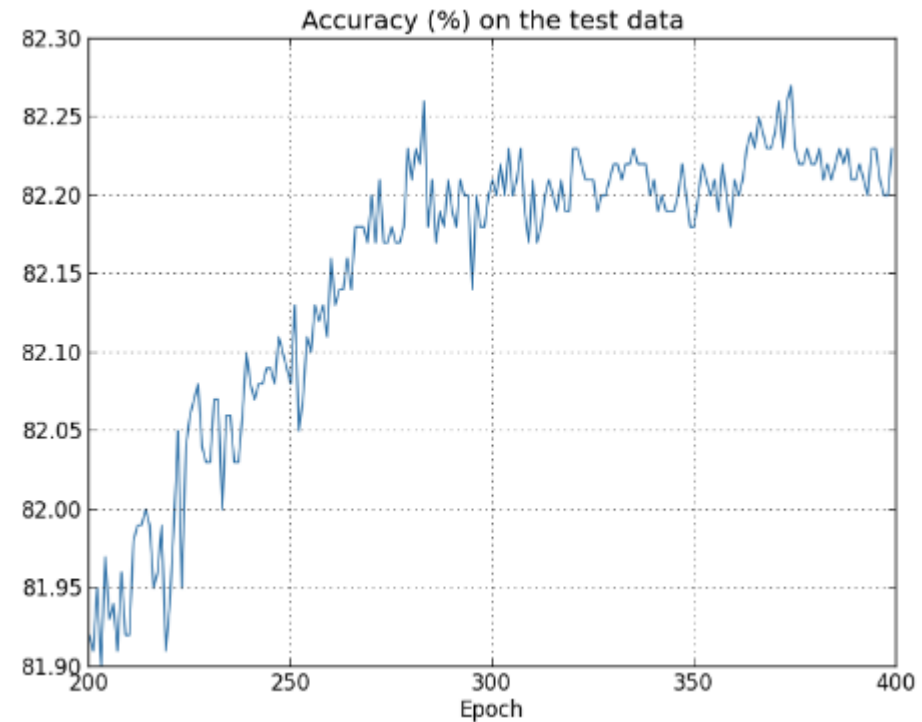
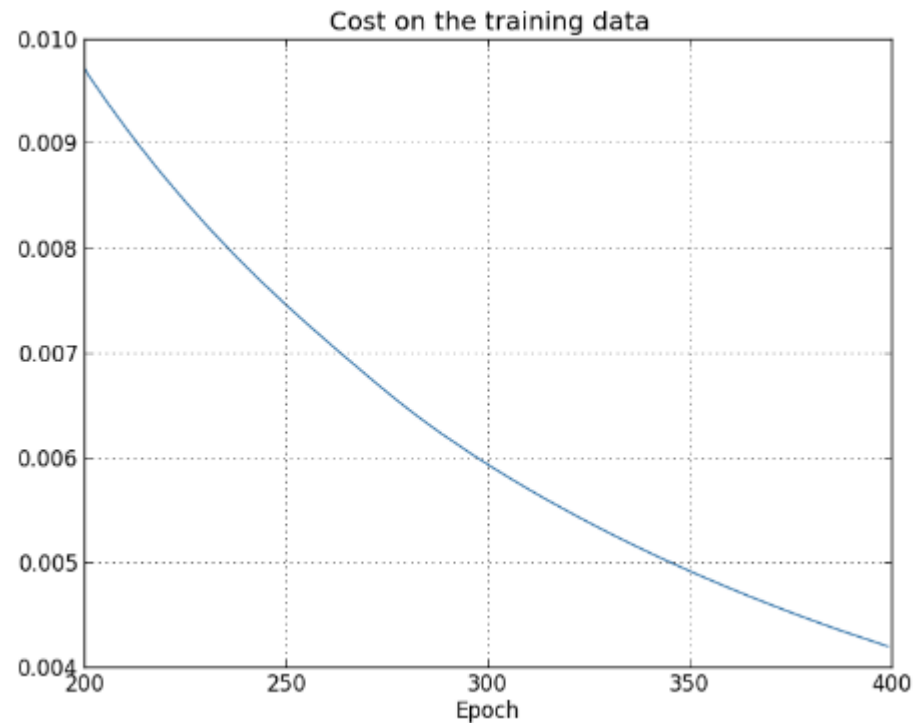
Derivative of commonly-used cost function

- derivative with respect to x

Name ⇄	Plot ⇄	Equation ⇄	Derivative (with respect to x) ⇄	Range
Identity		$f(x) = x$	$f'(x) = 1$	$(-\infty, \infty)$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$	$\{0, 1\}$
Logistic (a.k.a. Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$	$(0, 1)$
TanH		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$	$(-1, 1)$
Rectified linear unit (ReLU) ^[9]		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$[0, \infty)$
Leaky rectified linear unit (Leaky ReLU) ^[10]		$f(x) = \begin{cases} 0.01x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0.01 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$(-\infty, \infty)$
Parameteric rectified linear unit (PReLU) ^[11]		$f(\alpha, x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(\alpha, x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$(-\infty, \infty)$

Overfitting

- Overfitting: the model will work well for the existing data, but will fail to generalize to new situations.



- the network is *overfitting* or *overtraining* beyond epoch 280.

Regularization

- In general, one of the best ways of reducing overfitting is to increase the size of the training data.
- Unfortunately, training data can be expensive or difficult to acquire, so this is not always a practical option.
- Regularization: known as weight decay or L2 regularization

L2 Regularization

- add an extra term to the cost function
- the term is called the *regularization term*, such as the following regularized cross-entropy:

$$C = -\frac{1}{n} \sum_{x_j} \left[y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L) \right] + \frac{\lambda}{2n} \sum_w w^2.$$

- The first term is just the usual expression for the cross-entropy.
- The second term is the sum of the squares of all the weights in the network, and is scaled by a factor $\lambda/2n$, where $\lambda > 0$ is known as the regularization parameter, and n is, as usual, the size of our training set.

L2 Regularization

$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2,$$

- The effect of regularization is to make the network prefer to learn small weights, all other things being equal.
- Regularization can be viewed as a way of compromising between finding small weights and minimizing the original cost function.
- SGD in a regularized NN:

$$\begin{aligned}\frac{\partial C}{\partial w} &= \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} w \\ \frac{\partial C}{\partial b} &= \frac{\partial C_0}{\partial b}.\end{aligned}$$

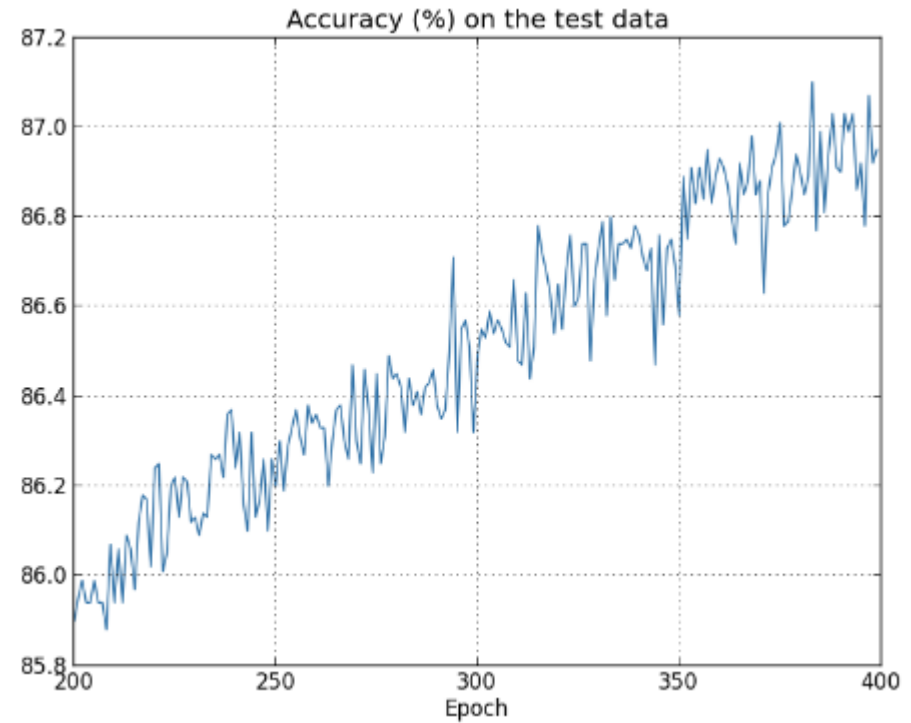
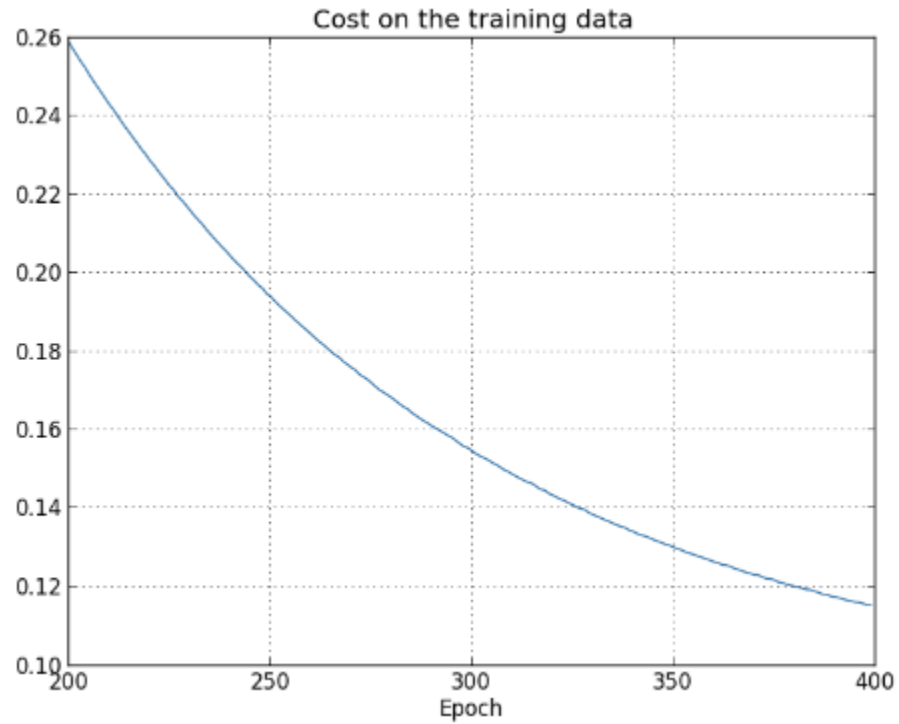


$$b \rightarrow b - \eta \frac{\partial C_0}{\partial b}.$$

$$\begin{aligned}w &\rightarrow w - \eta \frac{\partial C_0}{\partial w} - \frac{\eta \lambda}{n} w \\ &= \left(1 - \frac{\eta \lambda}{n}\right) w - \eta \frac{\partial C_0}{\partial w}.\end{aligned}$$

weight decay

Regularized Neural Nets



Why Regularization works?

- Smaller weights make the model, in some sense, less complex, and so provide a simpler and more powerful explanation for the data, and should thus be preferred.
- For neural nets, the smallness of the weights means that the behaviour of the network won't change too much if we change a few random inputs here and there. → insensitive to local noise in the data
- A regularized network learns to respond to types of evidence which are seen often across the training set.
- Regularized networks are constrained to build relatively simple models based on patterns seen often in the training data, and are resistant to learning peculiarities of the noise in the training data.

Other techniques for regularization

- L1 regularization:

$$C = C_0 + \frac{\lambda}{n} \sum_w |w|.$$

- weights updates with regularization:

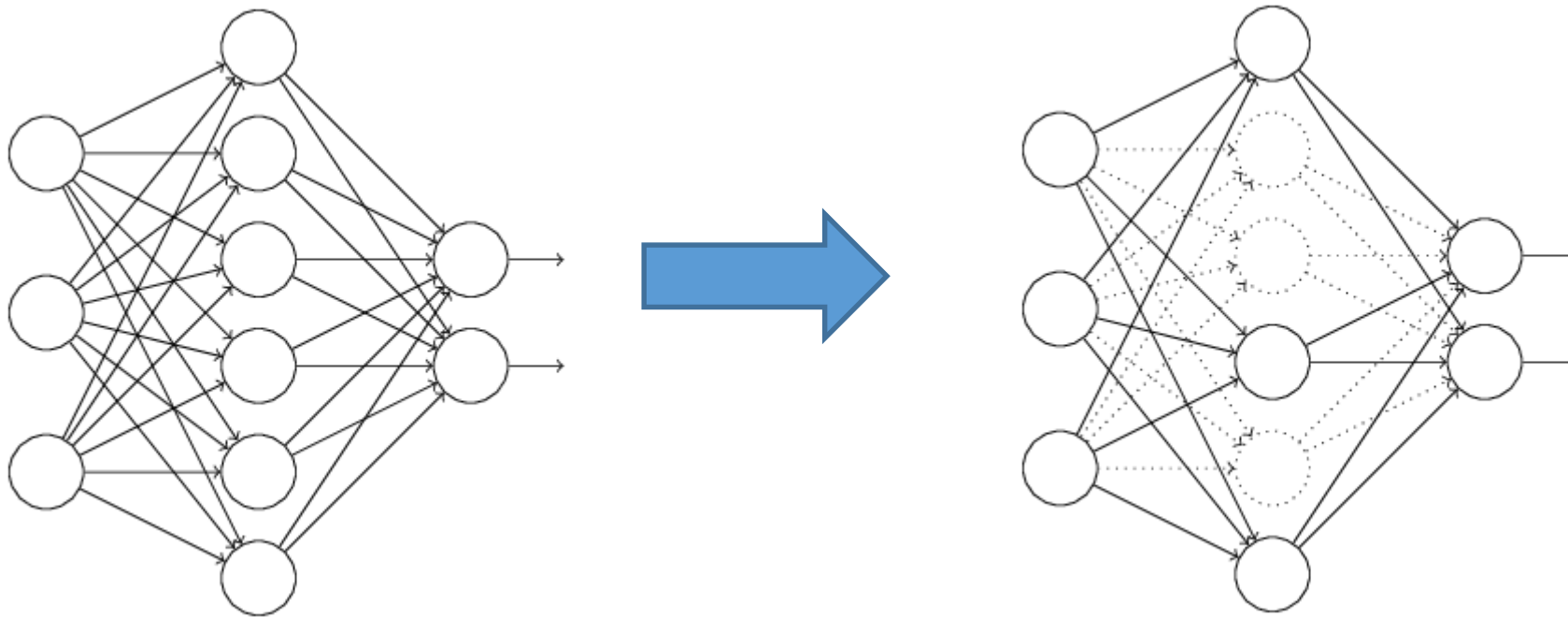
$$w \rightarrow w' = w - \frac{\eta\lambda}{n} \text{sgn}(w) - \eta \frac{\partial C_0}{\partial w},$$

the weights shrink by a constant amount toward 0.

- The net result is that L1 regularization tends to concentrate the weight of the network in a relatively small number of high-importance connections, while the other weights are driven toward zero.

Other techniques for regularization

- Dropout: it is a radically different technique for regularization.
- Dropout doesn't rely on modifying the cost function.
- Instead, in dropout we modify the network itself.



How Dropout Works

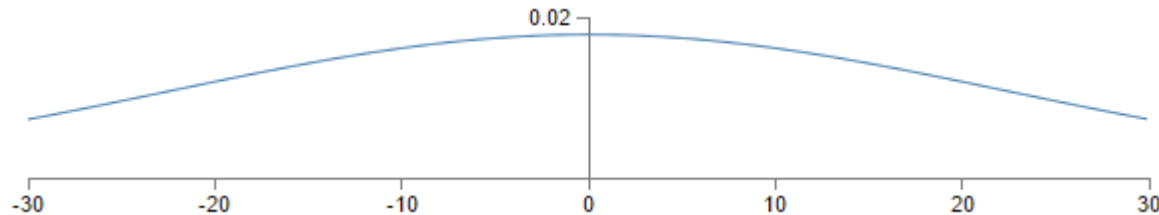
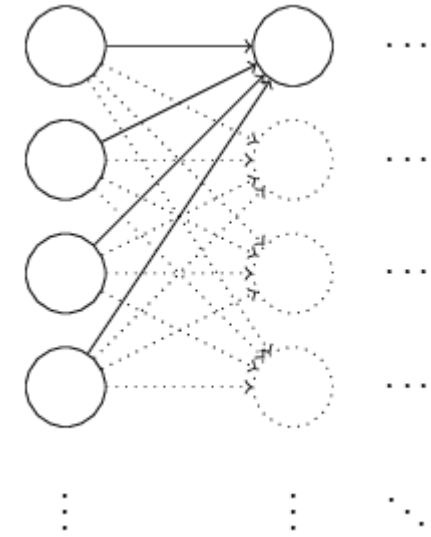
- Nullify random weights -> dropout neurons
- Forward-propagate the input through the modified network, and then backpropagate the result
- After doing this over a mini-batch of examples, we update the appropriate weights and biases.
- Restore the dropout neurons, then choose a new random subset of hidden neurons to delete => estimate the gradient for a different mini-batch, and update the weights and biases
- By repeating this process over and over, our network will learn a set of weights and biases spread over the whole network and not concentrated.

Why dropout works

- Heuristically, when we dropout different sets of neurons, it's rather like we're training different neural networks.
- The dropout procedure is like averaging the effects of a very large number of different networks.
- The different networks will overfit in different ways, and so, hopefully, the net effect of dropout will be to reduce overfitting.

Weight Initialization

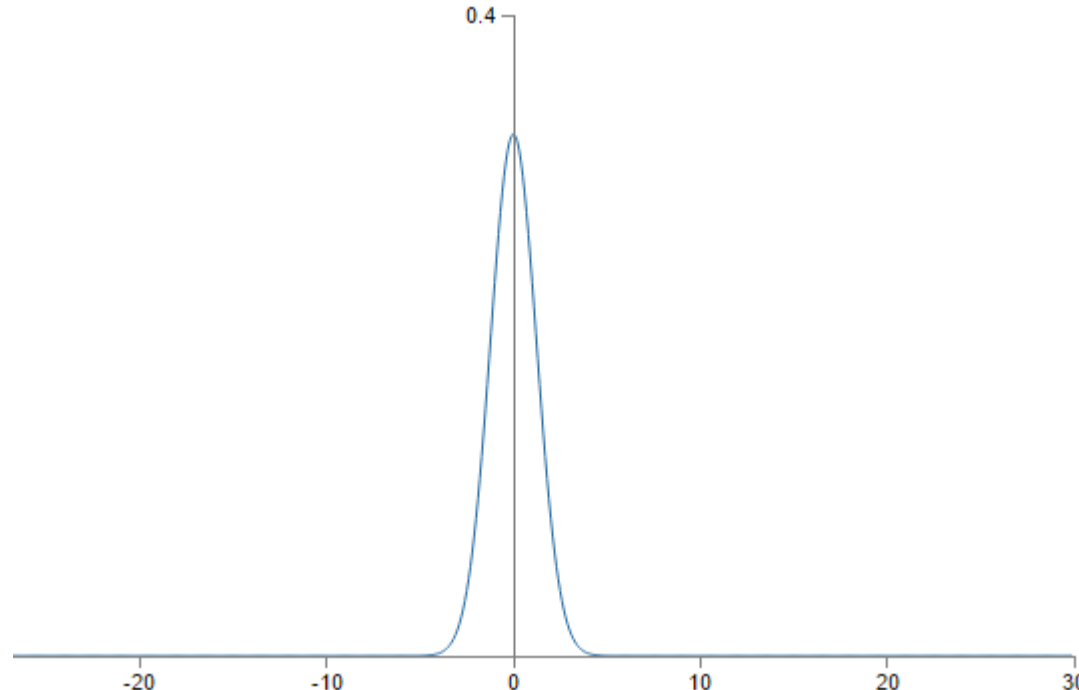
- Why weight initialization is important?
- A toy example:
 - 1000 input neurons, half is on, and half is off
 - using normalized Gaussian distribution to initialize weights
 - the weighted sum $z = \sum_j w_j x_j + b$
 - so z is a sum over a total of 501 normalized Gaussian random variables, accounting for the 500 weight terms and the 1 extra bias term.
 - z is itself distributed as a Gaussian with mean zero and standard deviation $\sqrt{501} \approx 22.4$



- if the weights are initialized using normalized Gaussians, then activations will often be very close to 0 or 1, and learning will proceed very slowly.

Better Initializations

- initialize the weights as Gaussian random variables with mean 0 and standard deviation $1/\sqrt{n_{in}}$.
- the bias as a Gaussian with mean 0 and standard deviation 1
- the weighted sum $z = \sum_j w_j x_j + b$ will be a Gaussian random variable with mean 0, but it'll be much more sharply peaked than it was before. The standard deviation is $\sqrt{3/2} = 1.22$



Such a neuron is much less likely to saturate, and correspondingly much less likely to have problems with a learning slowdown.

Revisit: Learning Rate

- Learning rate: η
 - It's often advantageous to decay the learning rate.

$$w_k \rightarrow w'_k = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k}$$

$$b_l \rightarrow b'_l = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l},$$

- Rule of Thumb
 - At the beginning, it's best to use a large learning rate that causes the weights to change quickly.
 - Later, we can reduce the learning rate as we make more fine-tuned adjustments to our weights.
- Three decay methods:
 - Step decay: Reduce the learning rate by some factor every few epochs.
 - Exponential decay: $\alpha = \alpha_0 e^{-kt}$, where α_0 , k are hyperparameters and t is the iteration number
 - 1/t decay: $\alpha = \alpha_0 / (1 + kt)$, where α_0 , k are hyperparameters and t is the iteration number

Early Stopping

- Use early stopping to determine the number of training epochs
- early stopping means that at the end of each epoch we should compute the performance on the validation data.
- When that stops improving, terminate.
- Early stopping also automatically prevents us from overfitting.
- In practice, a better rule is to terminate if the best performance doesn't improve for quite some time, e.g. the last ten epochs.

Advanced Gradient Descent Optimization Algorithms

- Momentum
- Nesterov accelerated gradient
- Adagrad
- Adadelata
- RMSprop
- Adam

Outline

- A brief history
- Basic mathematics in Neural Networks (NN)
- Neural Machine Translation (NMT)
- Advanced NMT

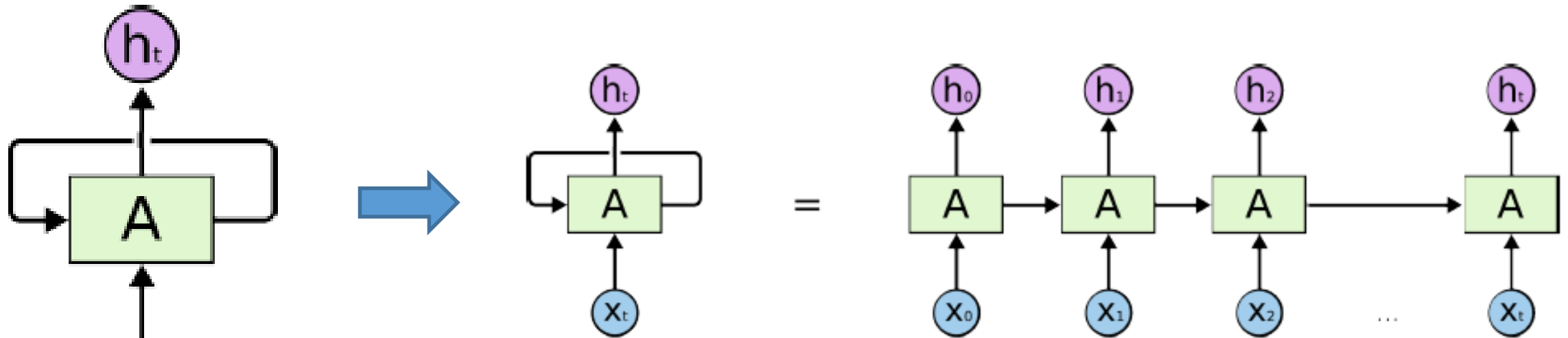
What is RNN?

- Question: Why is it difficult to process sentences using feed-forward NN?
 - We assume that all inputs (and outputs) are independent of each other.
 - It's a bad idea for sequential data.
- Idea behind RNN:
 - We need a NN which can make use of sequential information.
 - We need a NN which can learn or know the context of one word.

Informal Definition of RNN

- It's a kind of neural network
- It can process the sequence
- *Recurrent* indicates that it performs the same task for every element of a sequence, with the output being depended on the previous computations.
- We could say that it has a “memory” which captures information about what has been calculated so far.
- They are networks with loops in them, allowing information to persist.

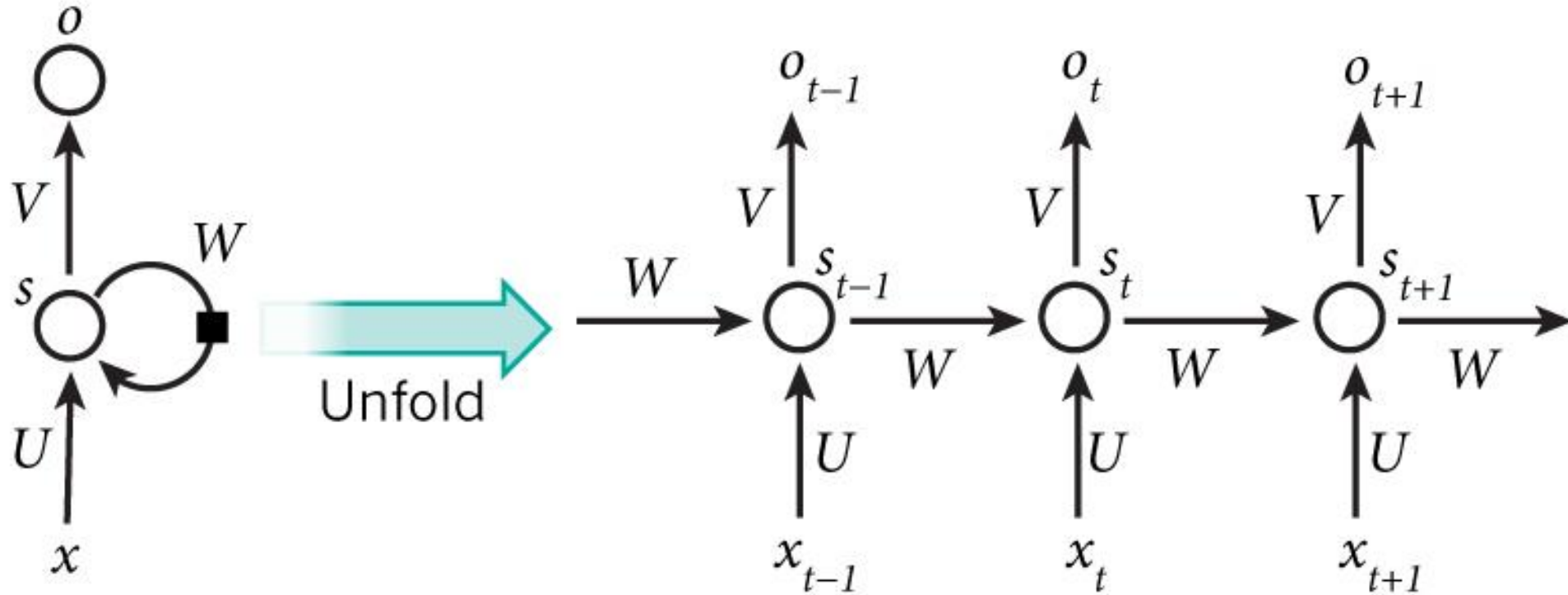
What a RNN looks like?



Recurrent Neural Networks
have loops.

A recurrent neural network can be thought of as multiple
copies of the same network, each passing a message to a
successor.

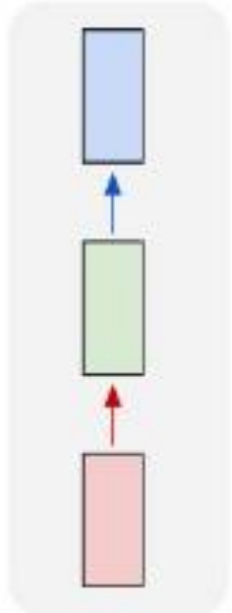
Architecture



A recurrent neural network and the unfolding in time of the computation involved in its forward computation. Source: Nature

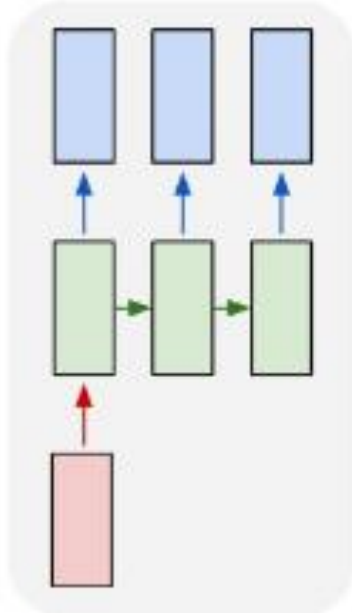
What can RNNs Do?

one to one



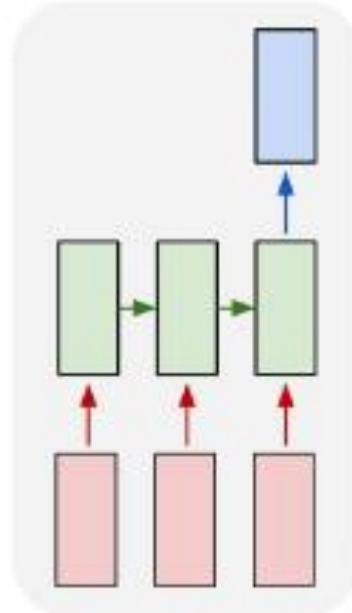
Vanilla NN
(FF):
Image
classification

one to many



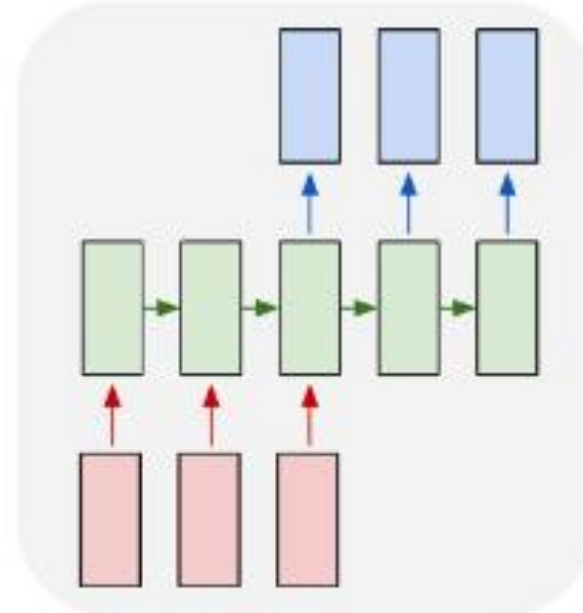
Sequence
Output: image
captioning

many to one



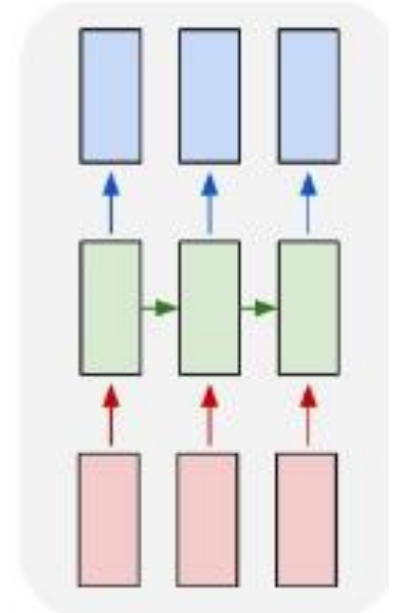
Sequence Input:
sentiment
analysis, text
classification

many to many



Sequence Input and
sequence Output: machine
translation, language
modelling

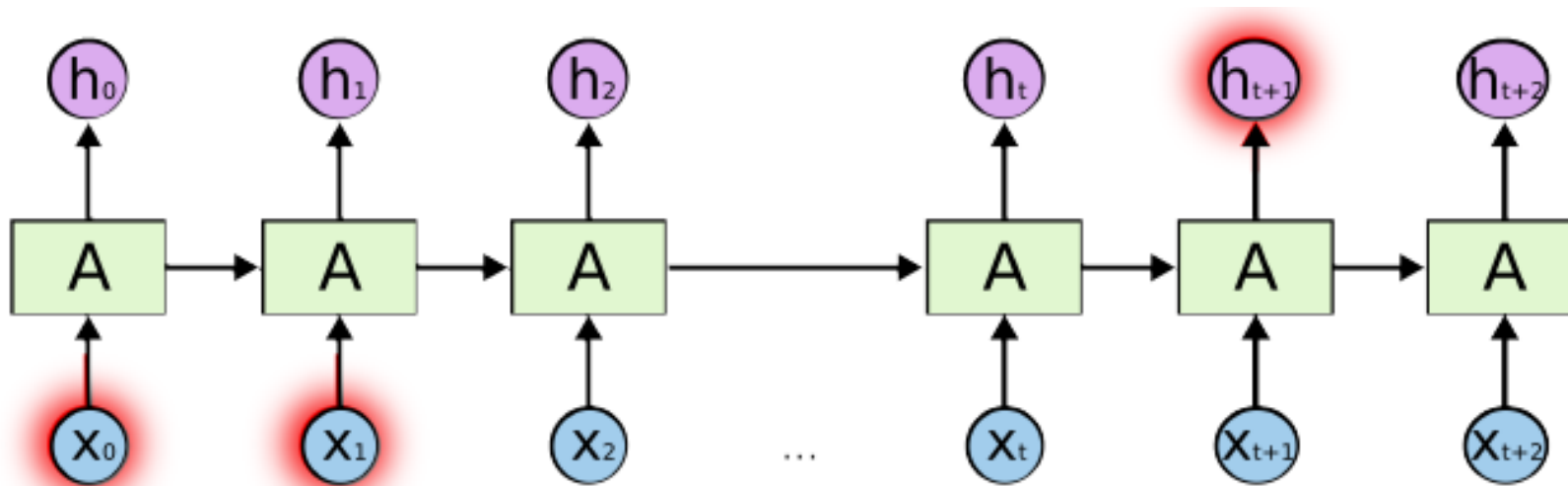
many to many



Synchronised
Sequence Input:
video
classification

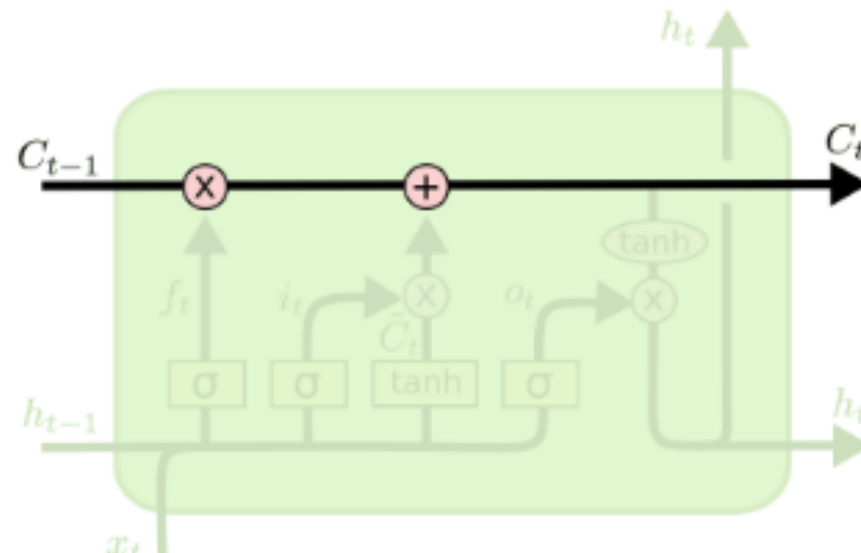
Special Unit in RNN: GRU/LSTM RNN

- Motivation
 - The vanishing gradient problem prevents standard RNNs from learning **long-term dependencies**.
 - LSTMs were designed to combat vanishing gradients through a **gating** mechanism.



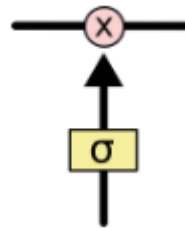
LSTMs

- A special kind of RNN, capable of learning long-term dependencies.
- Were introduced by Hochreiter & Schmidhuber (1997)
- The key to LSTMs is the cell state, which is kind of like a conveyor belt.
- It runs straight down the entire chain, with only some minor linear interactions.
- The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called **gates**.



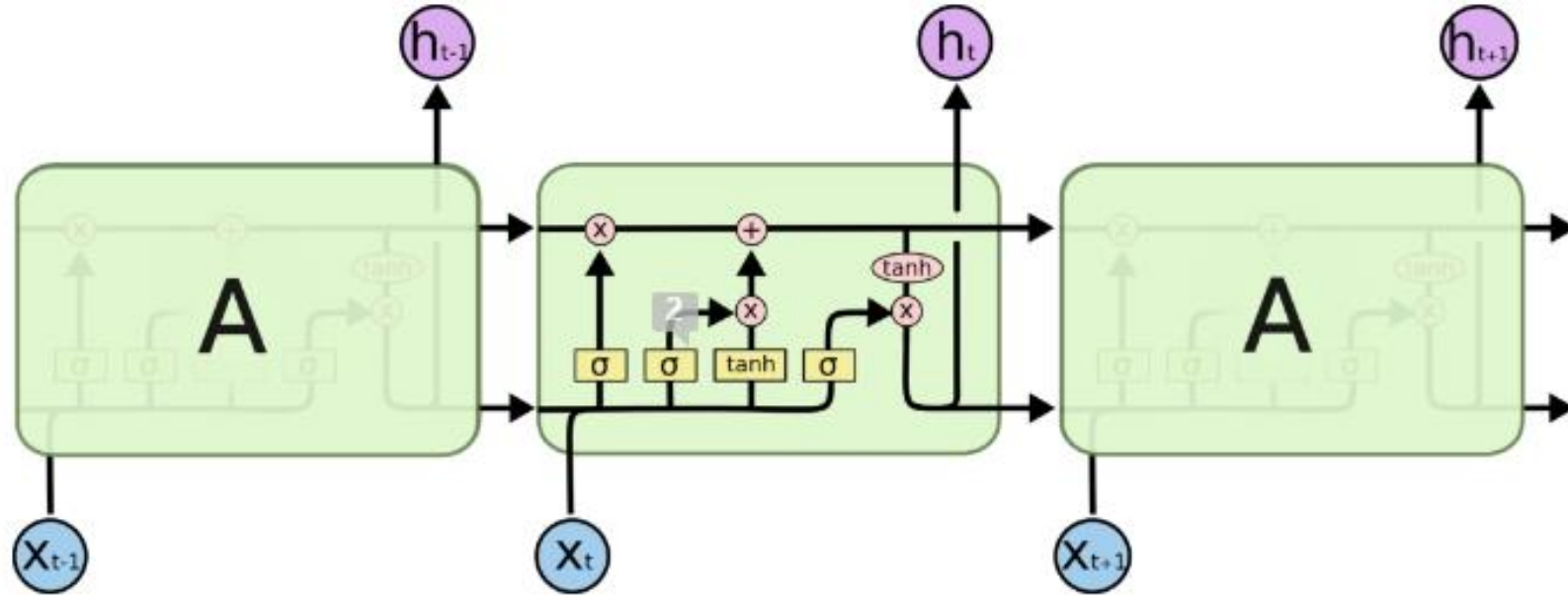
LSTMs

- Gates:
 - Gates are a way to optionally let information through.
 - They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.



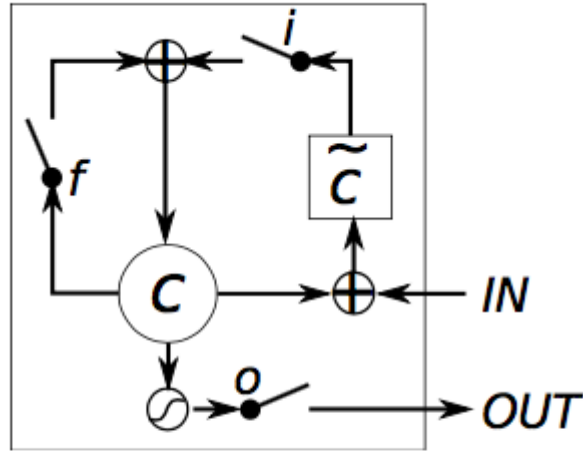
- The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through.
- A value of zero means “let nothing through,” while a value of one means “let everything through!”
- An LSTM has three of these gates, to protect and control the cell state.

LSTM RNN

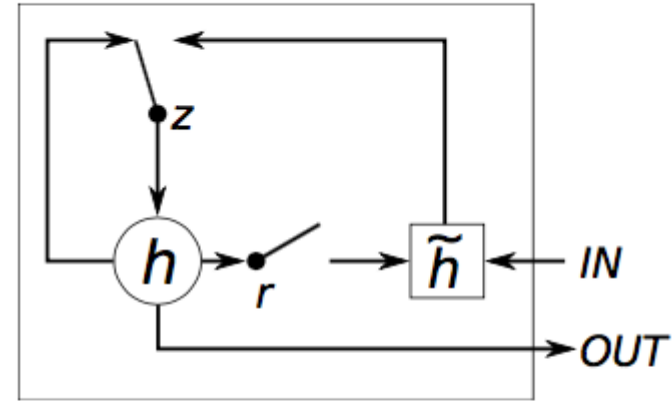


- The repeating module in an LSTM contains four interacting layers.

LSTM and GRU



(a) Long Short-Term Memory

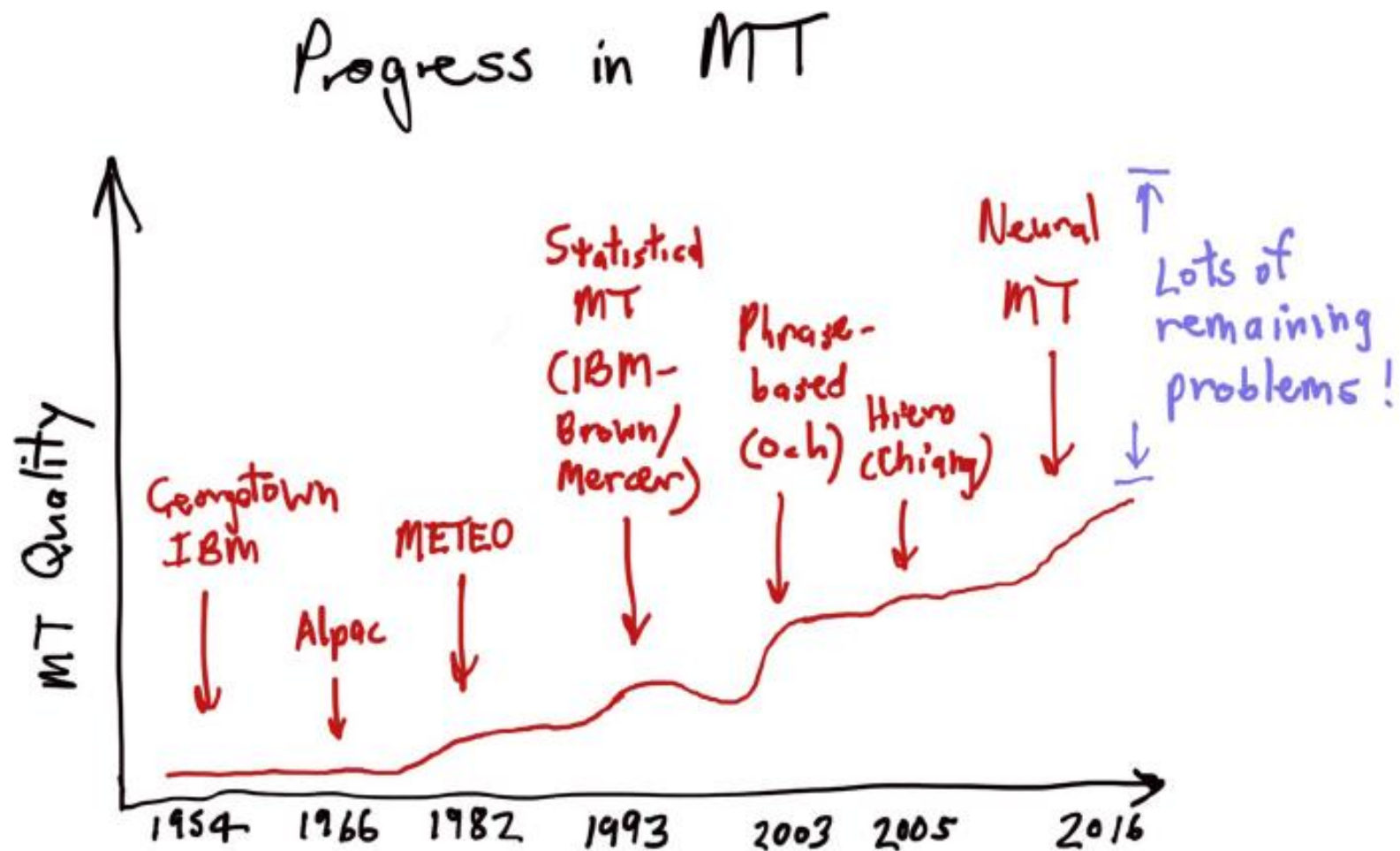


(b) Gated Recurrent Unit

Figure 1: Illustration of (a) LSTM and (b) gated recurrent units. (a) i , f and o are the input, forget and output gates, respectively. c and \tilde{c} denote the memory cell and the new memory cell content. (b) r and z are the reset and update gates, and h and \tilde{h} are the activation and the candidate activation.

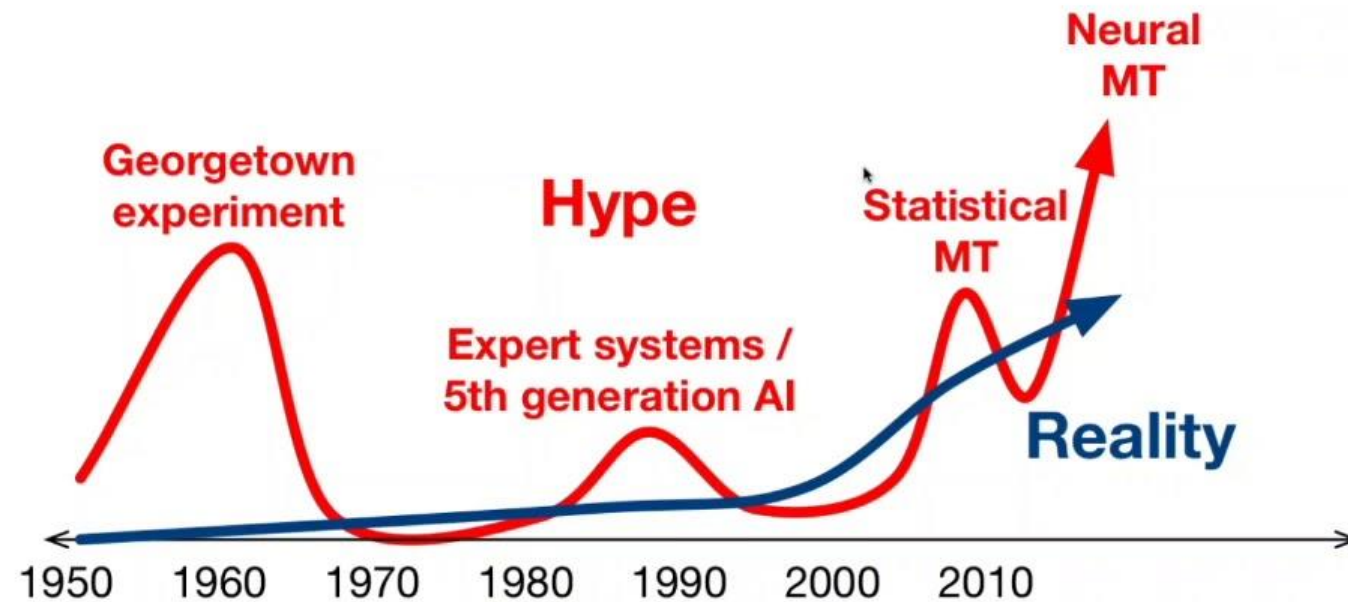
Neural Machine Translation

Progress in MT



Progress in MT

Hype and Reality

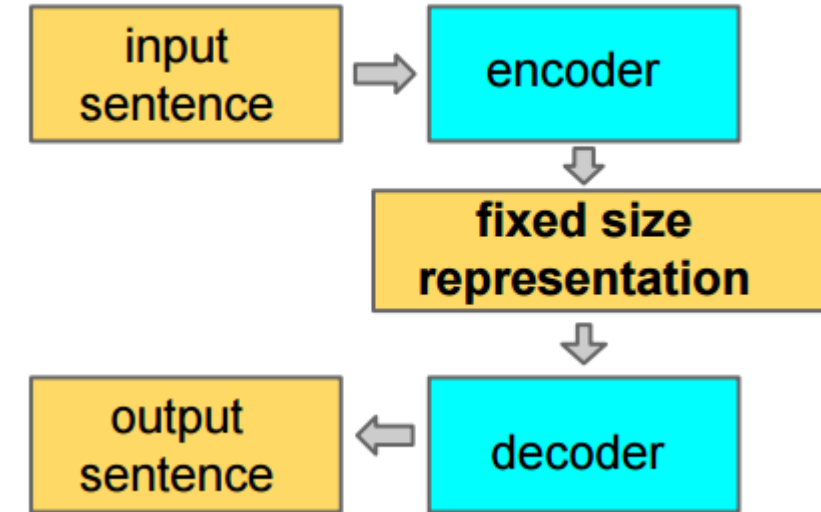
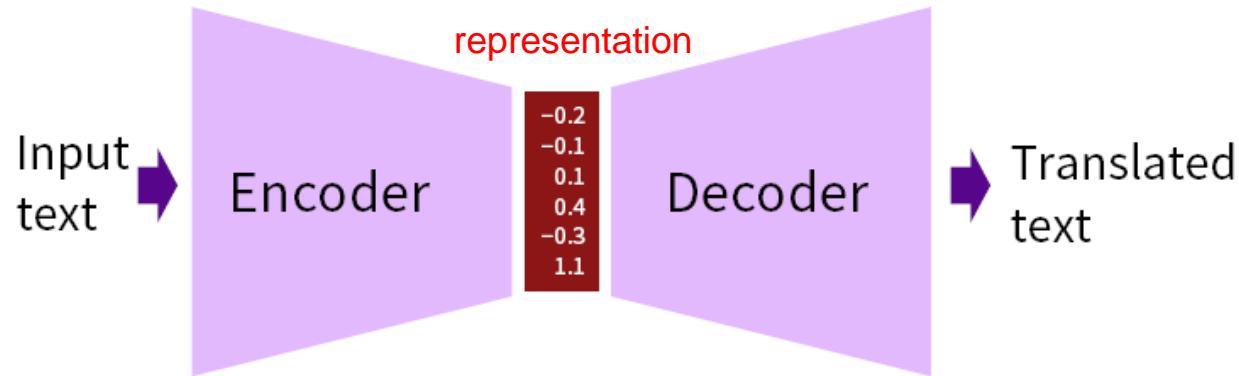


What is neural MT (NMT)?

- Neural Machine Translation is the approach of modelling the entire MT process via one big artificial neural network.
- NMT models use deep learning and representation learning.
- All parts of the neural translation model are trained jointly (end-to-end) to maximize the translation performance.
- Probability equation for NMT:

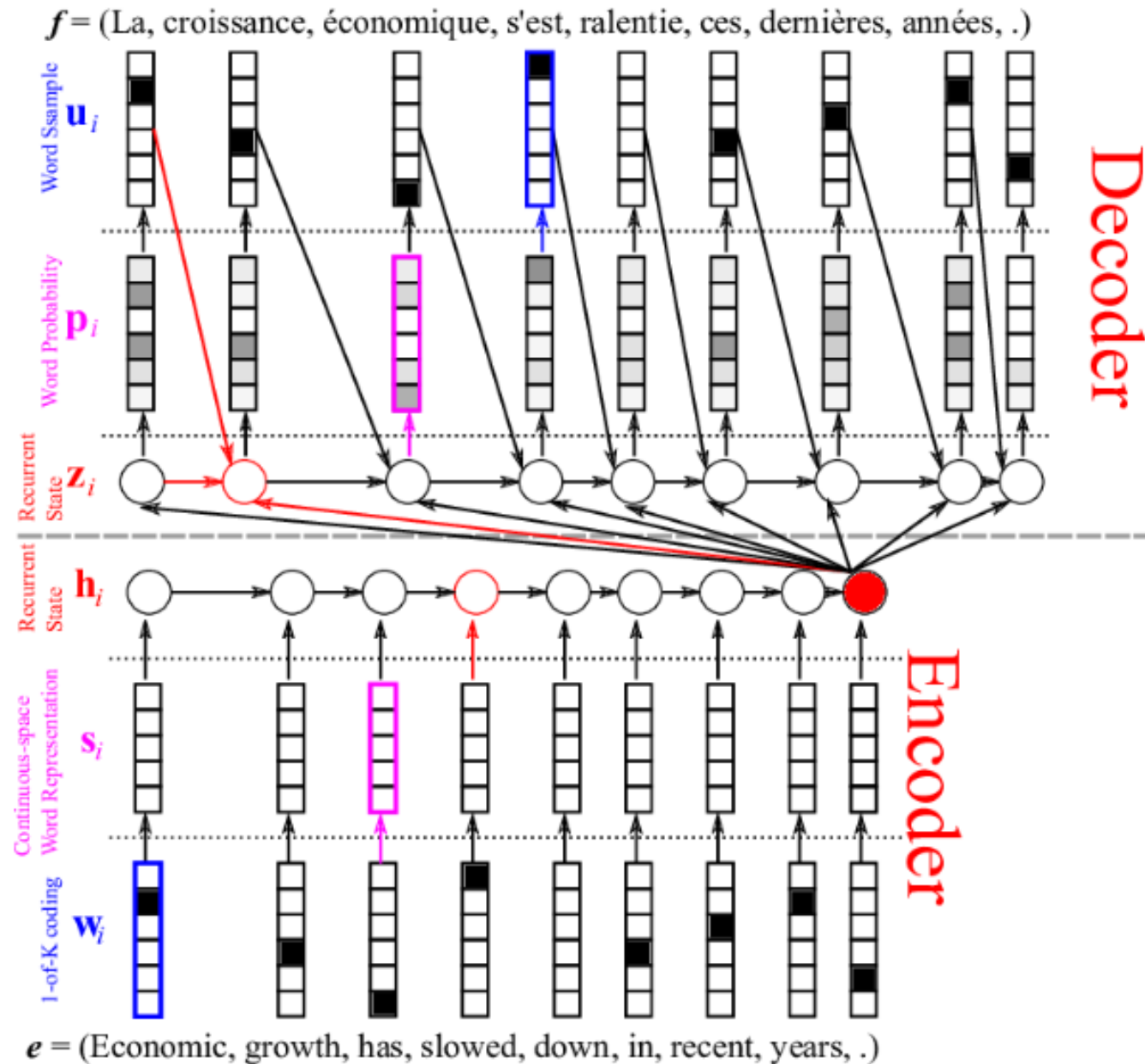
$$p(\mathbf{y}|\mathbf{x}) = \prod_{i=1}^n p(y_i | y_{<i}, x_{\leq m})$$

Neural encoder-decoder architectures

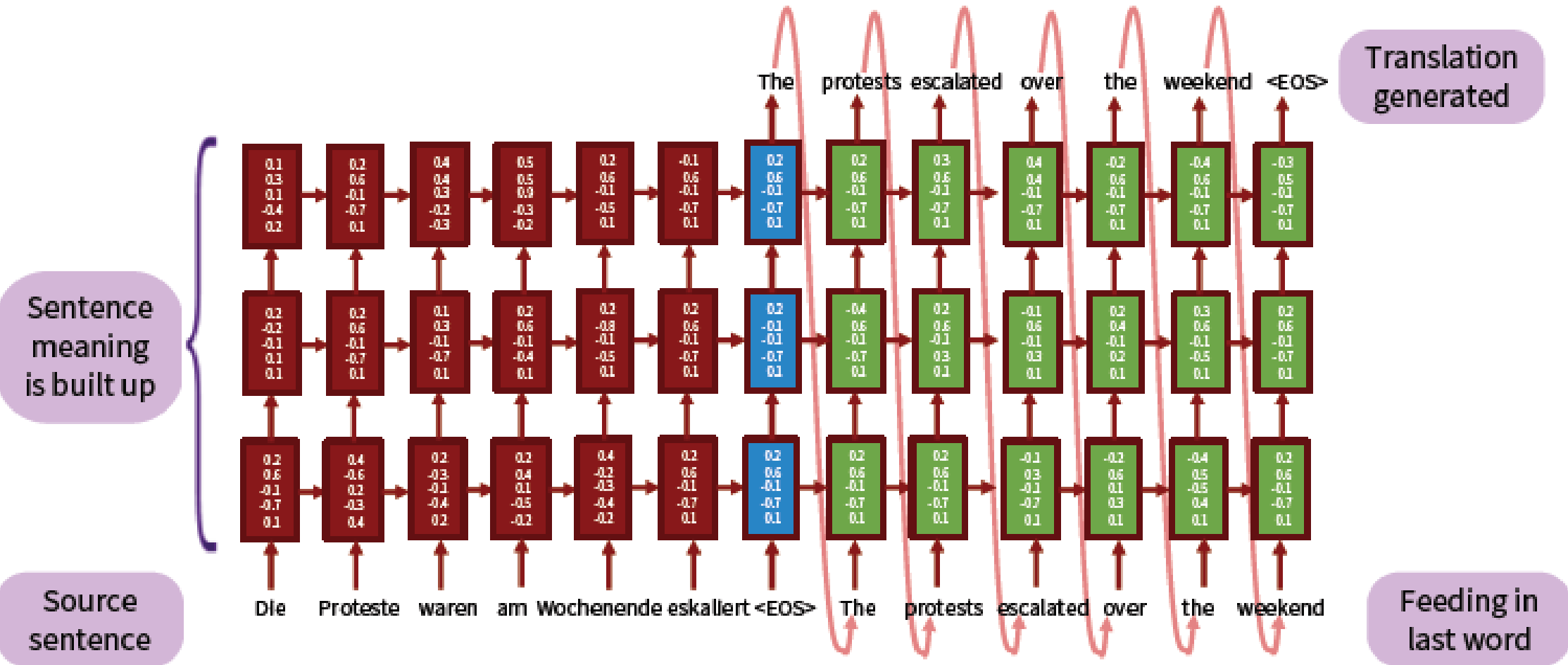


- two RNNs (LSTM or GRU):
 - encoder reads input and produces hidden state representations
 - decoder produces output, based on last encoder hidden state
- encoder and decoder are learned jointly
 - supervision signal from parallel text is backpropagated

Neural encoder-decoder architectures



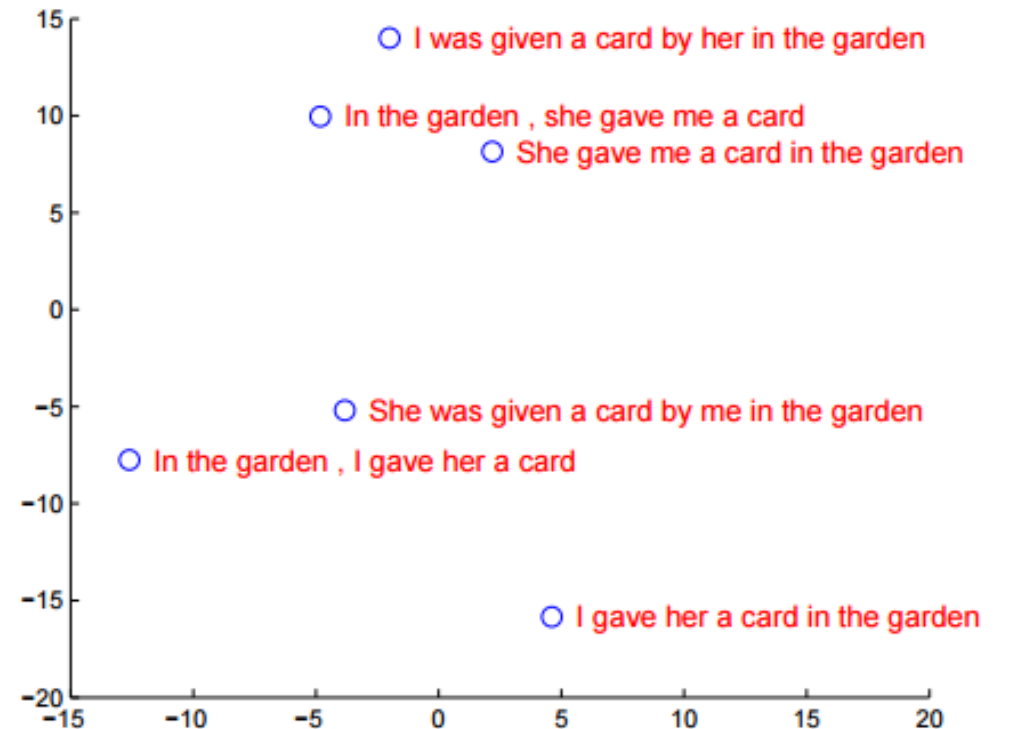
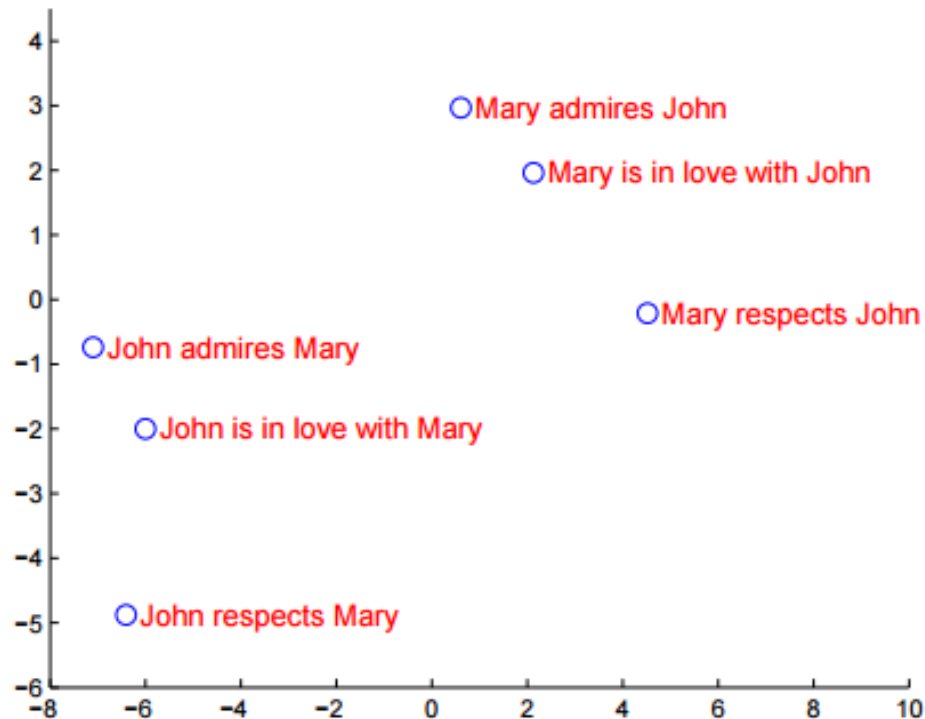
Neural encoder-decoder architectures



A deep recurrent neural network

Summary Vector (Context Vector)

- last encoder hidden-state “summarizes” source sentence
- with multilingual training, we can potentially learn language-independent meaning representation

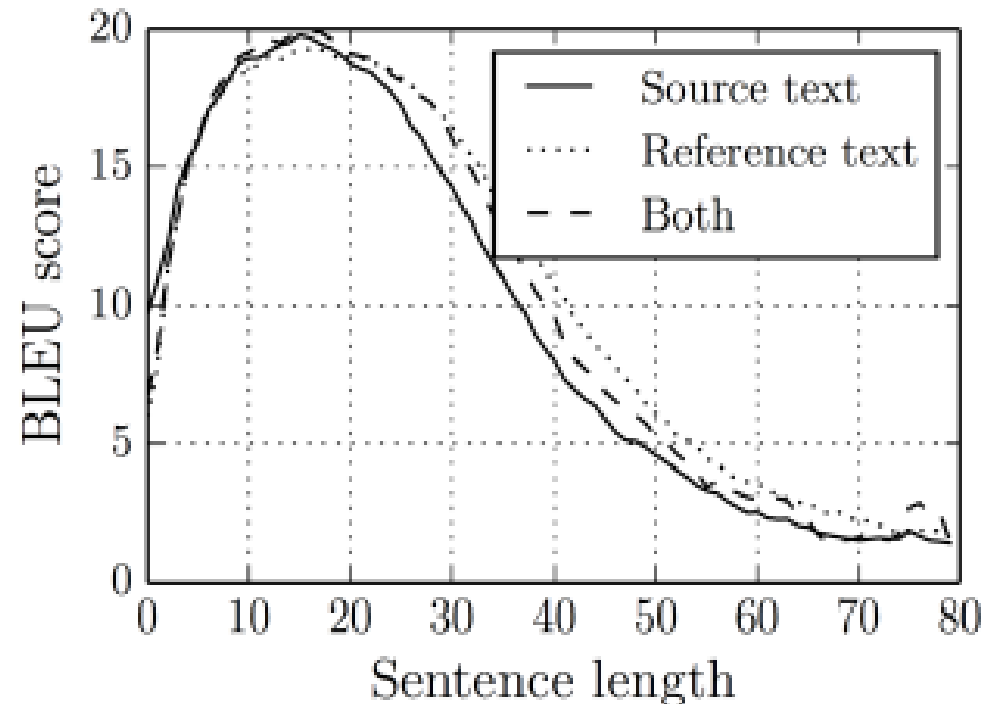


Outline

- A brief history
- Basic mathematics in Neural Networks (NN)
- Neural Machine Translation (NMT)
- **Advanced NMT**

Summary vector as information bottleneck

- can fixed-size vector represent meaning of arbitrarily long sentence?
- empirically, quality decreases for long sentences
- reversing source sentence brings some improvement



Attentional encoder-decoder

- encoder
 - goal: avoid bottleneck of summary vector
 - Tell decoder what is now being translated:
 - use bidirectional RNN, and concatenate forward and backward states
 - annotation vector h_i
 - represent source sentence as a vector of n annotations
 - variable-length representation

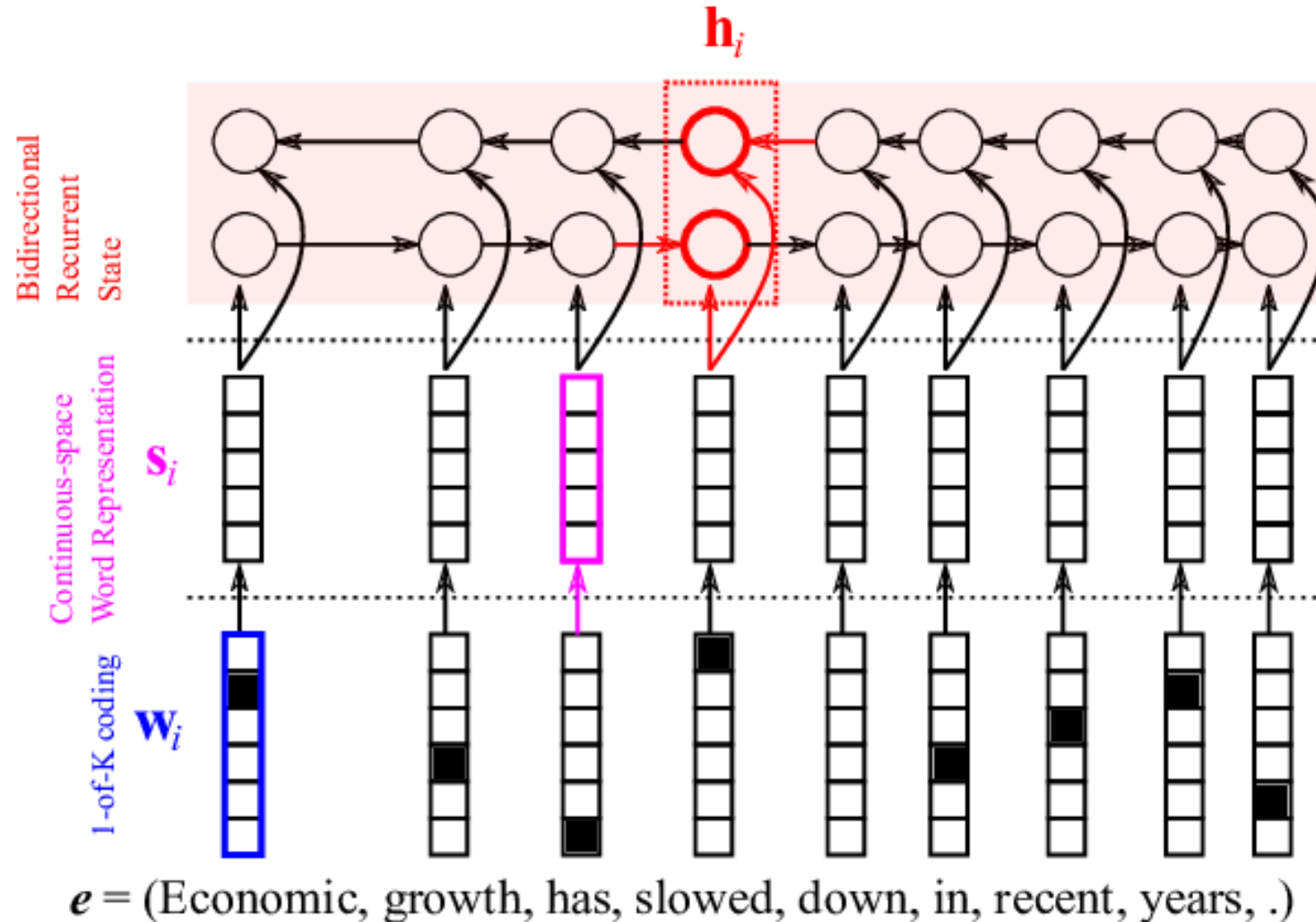
The agreement on European Economic Area was signed in August 1992.

L'accord sur ???

L'accord sur l'Espace économique européen a été signé en ???

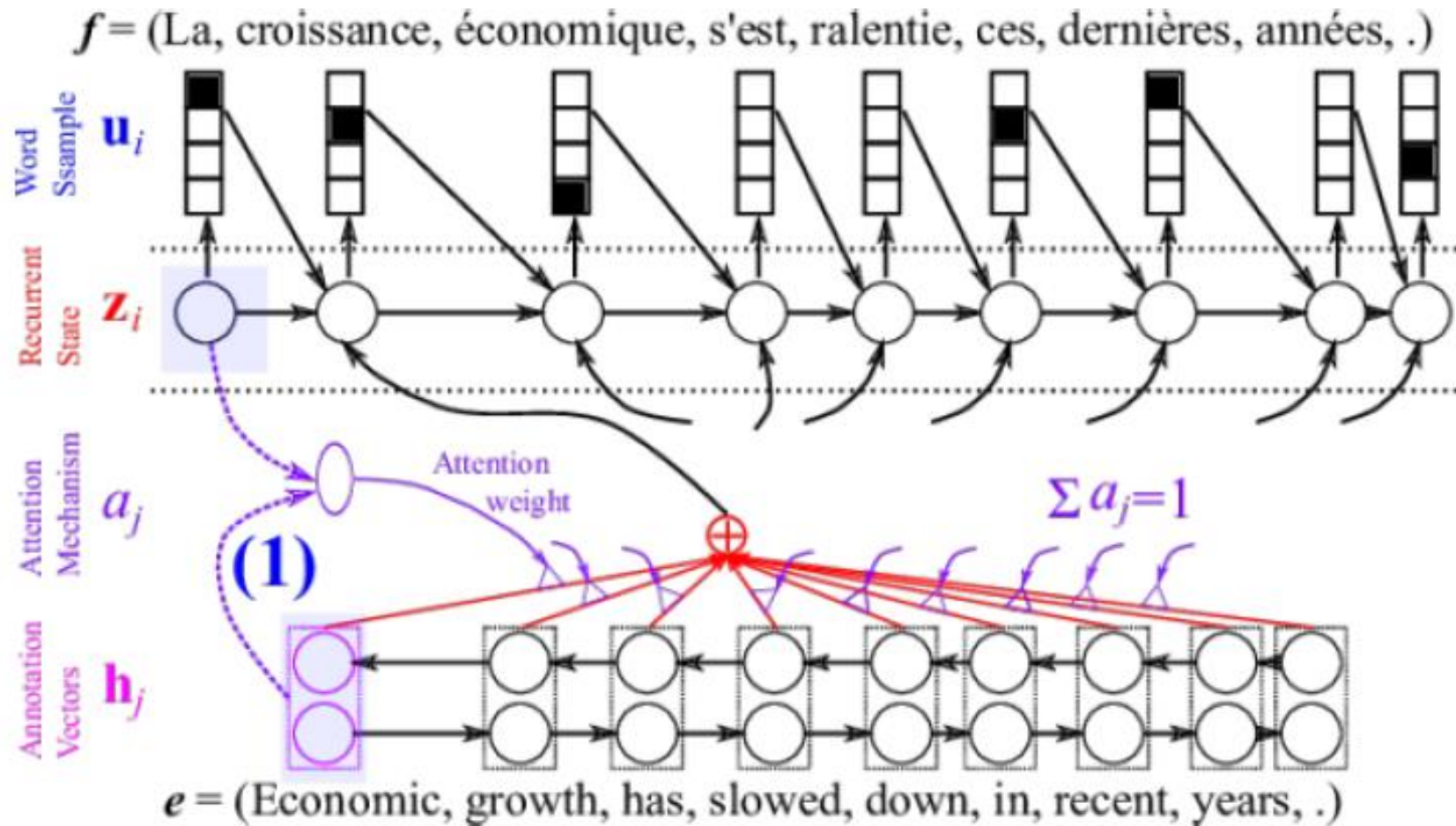
Have such hints computed by the net itself!

Bidirectional encoder in attentional encoder-decoder



- h_i contains x_i together with its context ($\dots, x_{i-1}, x_{i+1}, \dots$).
- (h_1, \dots, h_L) is the new variable-length representation instead of fixed-length c .

Attentional Neural Machine Translation



Attentional Neural Machine Translation

- Brief description:
 - The framework is an end-to-end architecture which takes the source-side string in and encodes it to a continuous vector representation using a bidirectional recurrent neural network h_j ;
 - In the output layer, the decoder is a RNN which takes the previous generated target word, the hidden state Z_i of the decoder and source-side context C_i as conditions to predict current target word.

$$p(u_i | u_{<i}, x) = g(u_{i-1}, z_i, c_i)$$

- The source-side context C_i is a weighted sum of annotation vectors h_j which is a soft alignment between the predicted target word against source words.

What is Attention Mechanism?

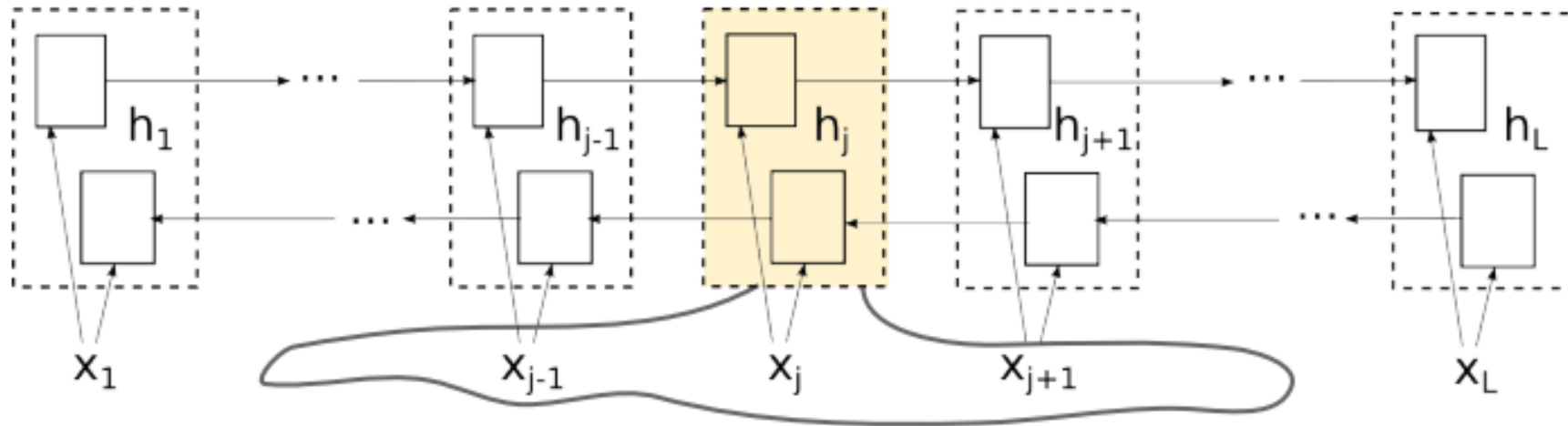
- The attention mechanism indicates
 - an implicit alignment between predicted target words against source-side words.
 - the attention mechanism provides a soft alignment which implies which words in the source side contribute more to the target word just like the word alignment in SMT.
 - the attention mechanism changes the fixed-length context vector to a variable-length context vector in which the weight for each partial annotation vector h_j decides the contribution.
 - the weights are jointly learned when training the NMT system with a feedforward network by using BP algorithm. The sum of all weights is 1.

Step-by-step Attention Mechanism

- Step 1: bidirectional RNN as the new encoder

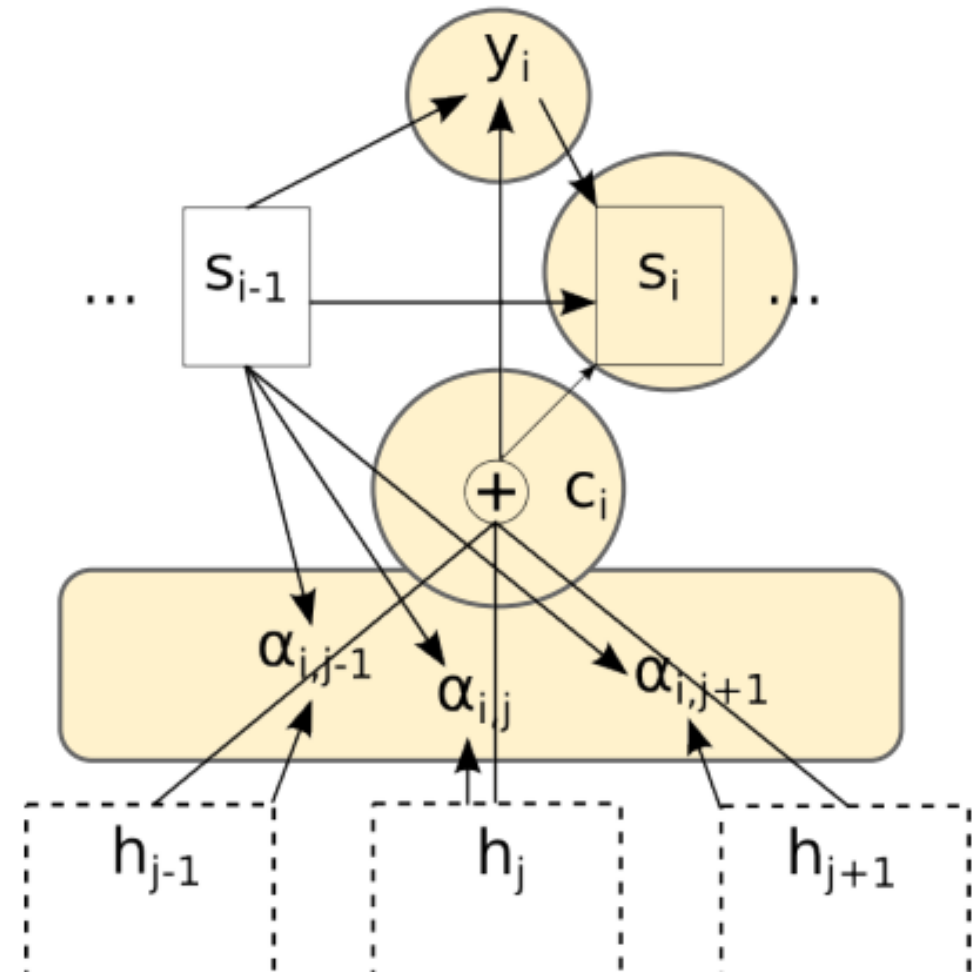
Bidirectional RNN: h_j contains x_j together with its context ($\dots, x_{j-1}, x_{j+1}, \dots$).

(h_1, \dots, h_L) is the new *variable-length* representation instead of *fixed-length* c .



Step-by-step Attention Mechanism

- Step 2: Attentional Decoder
 - compute alignment
 - compute context
 - generate new output
 - compute new decoder state



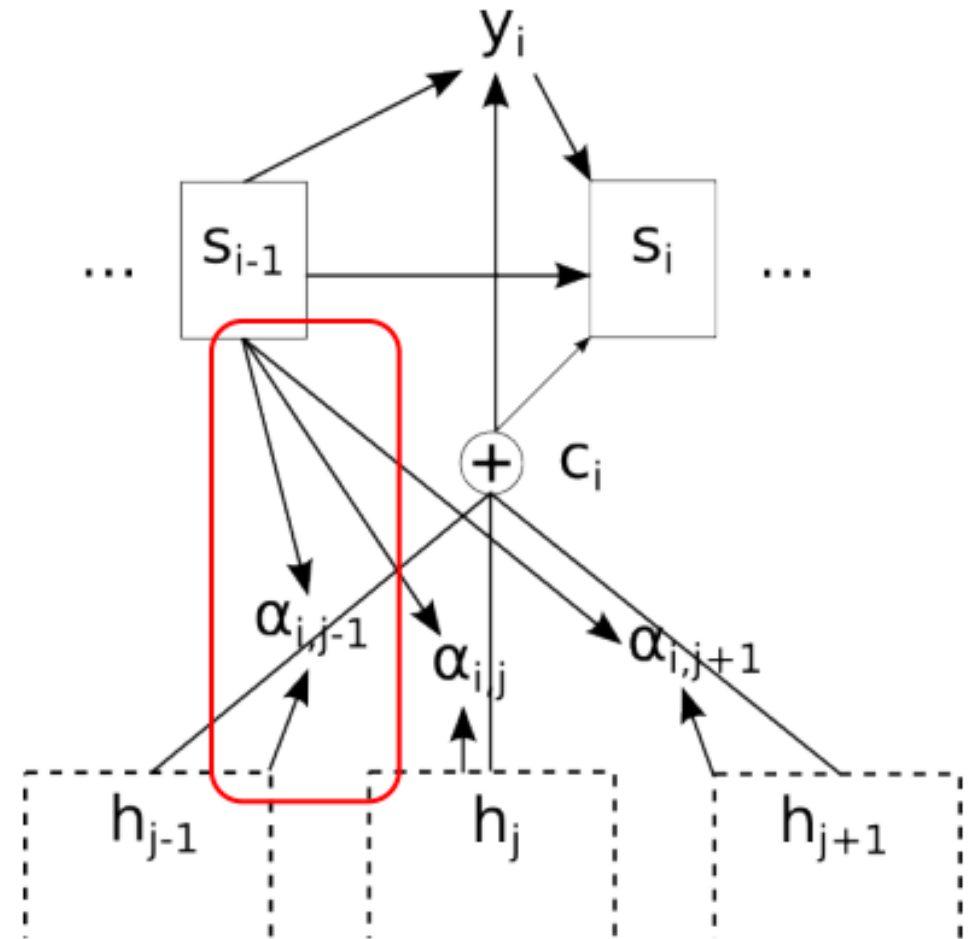
Step-by-step Attention Mechanism

- Step 3: alignment model

$$e_{ij} = v^T \tanh(W s_{i-1} + V h_j) \quad (1)$$

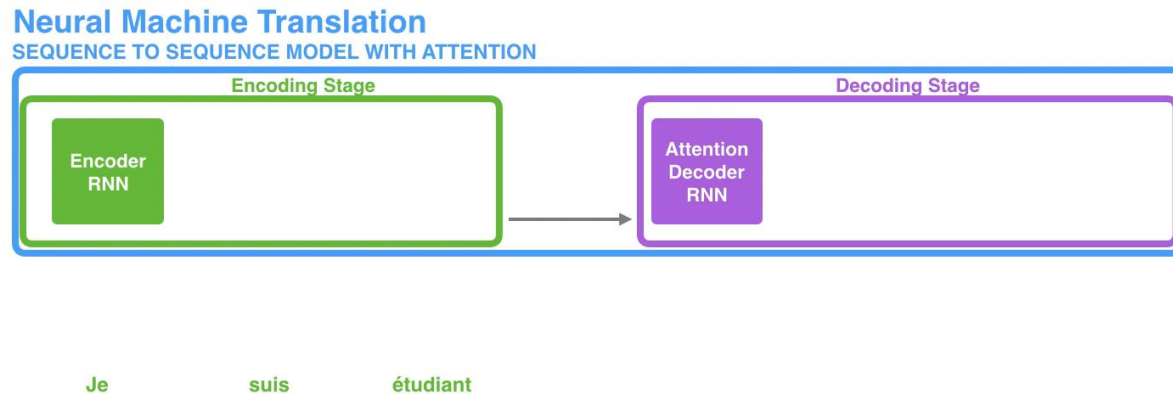
$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^L \exp(e_{ik})} \quad (2)$$

- nonlinearity (tanh) is crucial!
- simplest model possible
- $V h_j$ is precomputed => quadratic complexity with low constant



Visualising the workings of the attention mechanism

- First, the encoder passes a lot more data to the decoder. Instead of passing the last hidden state of the encoding stage, the encoder passes *all* the hidden states to the decoder



Visualising the workings of the attention mechanism

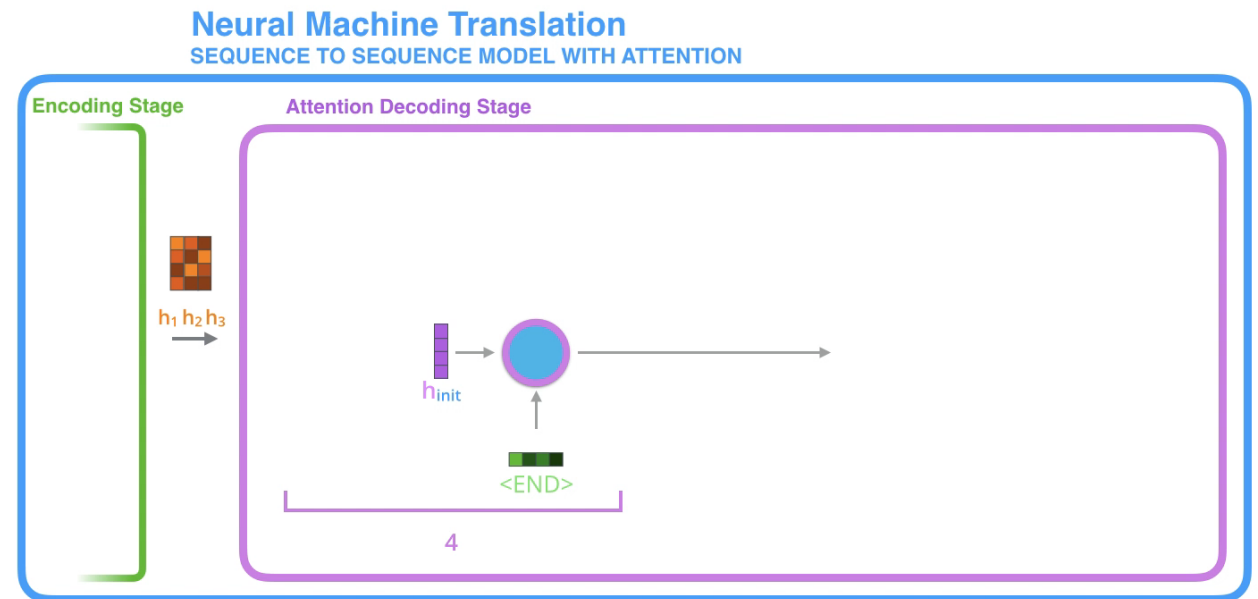
- Second, an attention decoder does an extra step before producing its output. In order to focus on the parts of the input that are relevant to this decoding time step, the decoder does the following:
 - Look at the set of encoder hidden states it received – each encoder hidden states is most associated with a certain word in the input sentence
 - Give each hidden states a score (let's ignore how the scoring is done for now)
 - Multiply each hidden states by its softmaxed score, thus amplifying hidden states with high scores, and drowning out hidden states with low scores

Attention at time step 4



Visualising the workings of the attention mechanism

- This scoring exercise is done at each time step on the decoder side.
- Let us now bring the whole thing together in the following visualization and look at how the attention process works:
 - The attention decoder RNN takes in the embedding of the $\langle \text{END} \rangle$ token, and an initial decoder hidden state.
 - The RNN processes its inputs, producing an output and a new hidden state vector (h_4). The output is discarded.
 - Attention Step: We use the encoder hidden states and the h_4 vector to calculate a context vector (C_4) for this time step.
 - We concatenate h_4 and C_4 into one vector.
 - We pass this vector through a feedforward neural network (one trained jointly with the model).
 - The output of the feedforward neural networks indicates the output word of this time step.
 - Repeat for the next time steps
 - Your browser does not support the video tag.



Attentional encoder-decoder: math

notation

- W, U, E, C, V are weight matrices (of different dimensionality)
 - E one-hot to embedding (e.g. $50000 \cdot 512$)
 - W embedding to hidden (e.g. $512 \cdot 1024$)
 - U hidden to hidden (e.g. $1024 \cdot 1024$)
 - C context (2x hidden) to hidden (e.g. $2048 \cdot 1024$)
 - V_o hidden to one-hot (e.g. $1024 \cdot 50000$)
- separate weight matrices for encoder and decoder (e.g. E_x and E_y)
- input X of length T_x ; output Y of length T_y

Attentional encoder-decoder: math

encoder

$$\begin{aligned}\vec{h}_j &= \begin{cases} 0, & \text{if } j = 0 \\ \tanh(\vec{W}_x E_x x_j + \vec{U}_x h_{j-1}) & \text{if } j > 0 \end{cases} \\ \overleftarrow{h}_j &= \begin{cases} 0, & \text{if } j = T_x + 1 \\ \tanh(\overleftarrow{W}_x E_x x_j + \overleftarrow{U}_x h_{j+1}) & \text{if } j \leq T_x \end{cases} \\ h_j &= (\vec{h}_j, \overleftarrow{h}_j)\end{aligned}$$

Attentional encoder-decoder: math

decoder

$$s_i = \begin{cases} \tanh(W_s \overleftarrow{h_i}), & , if \ i = 0 \\ \tanh(W_y E_y y_{i-1} + U_y s_{i-1} + C c_i), & , if \ i > 0 \end{cases}$$

$$t_i = \tanh(U_o s_{i-1} + W_o W E_y y_{i-1} + C_o c_i)$$

$$y_i = softmax(V_o t_i)$$

Attentional encoder-decoder: math

attention model

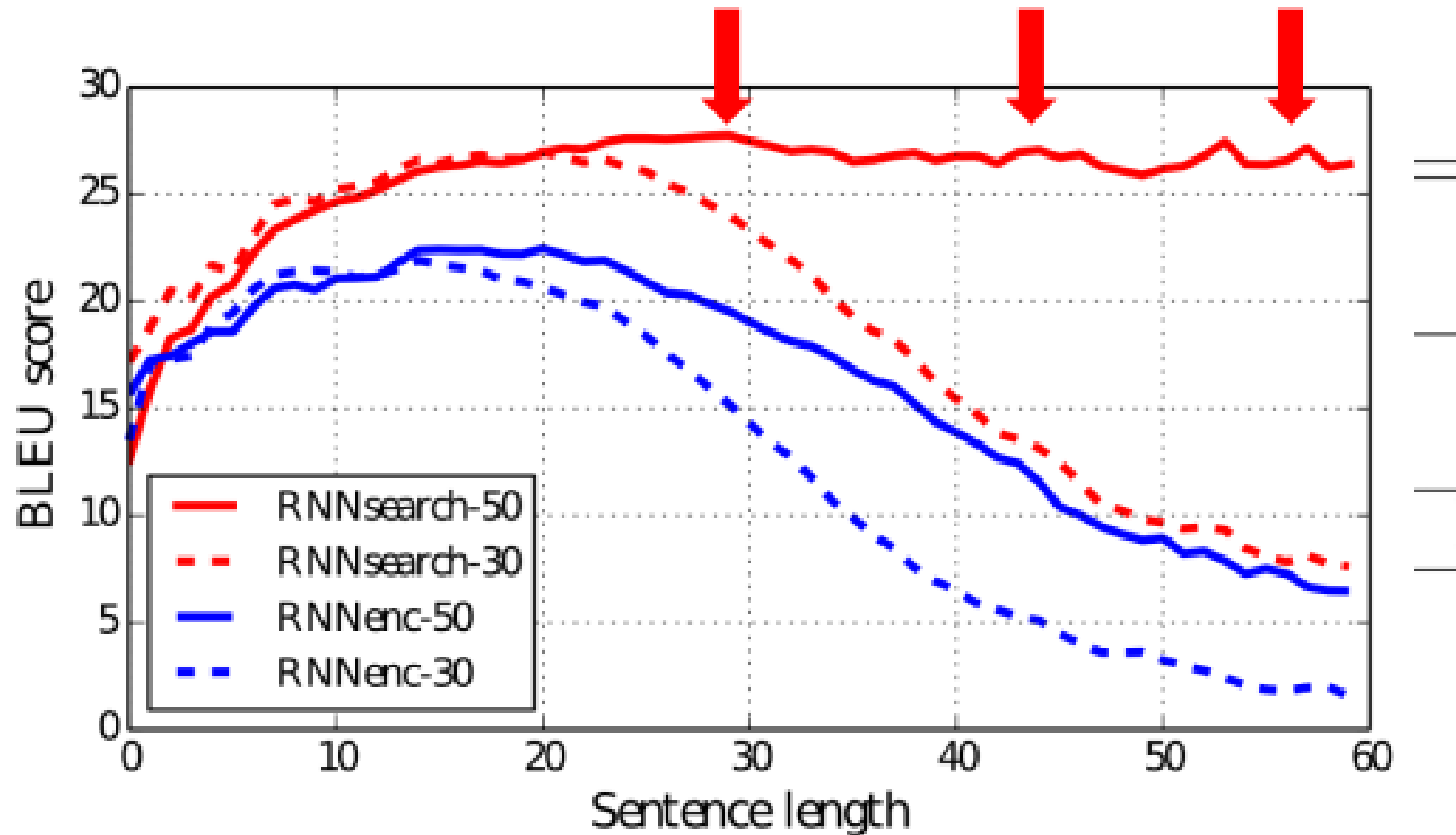
$$e_{ij} = v_a^\top \tanh(W_a s_{i-1} + U_a h_j)$$

$$\alpha_{ij} = \text{softmax}(e_{ij})$$

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$$

Comparison with Vanilla Encoder-Decoder

no performance drop on long sentence:




Attention Model

- side effect: we obtain alignment between source and target sentence
- information can also flow along recurrent connections, so there is no guarantee that attention corresponds to alignment
- applications:
 - visualisation
 - replace unknown words with back-off dictionary [Jean et al., 2015]
 - ...

Alignment from the attention model

The agreement on the European Economic Area was signed in August 1992 .



L' accord sur l' Espace économique européen a été signé en août 1992 .

This diagram illustrates the attention mechanism for the first sentence pair. It shows vertical lines for each word in the English sentence (top) and the French sentence (bottom). Curved lines represent the attention weights between the words. The English words 'agreement', 'European', 'Economic', and 'Area' are aligned with the French words 'accord', 'Espace', 'économique', and 'européen' respectively. The English words 'signed' and 'in' are aligned with the French words 'a été' and 'signé' respectively. The English words 'August' and '1992' are aligned with the French words 'en août' and '1992' respectively. The English word 'The' is aligned with the French word 'L\'', and the English word 'was' is aligned with the French word 'a'.

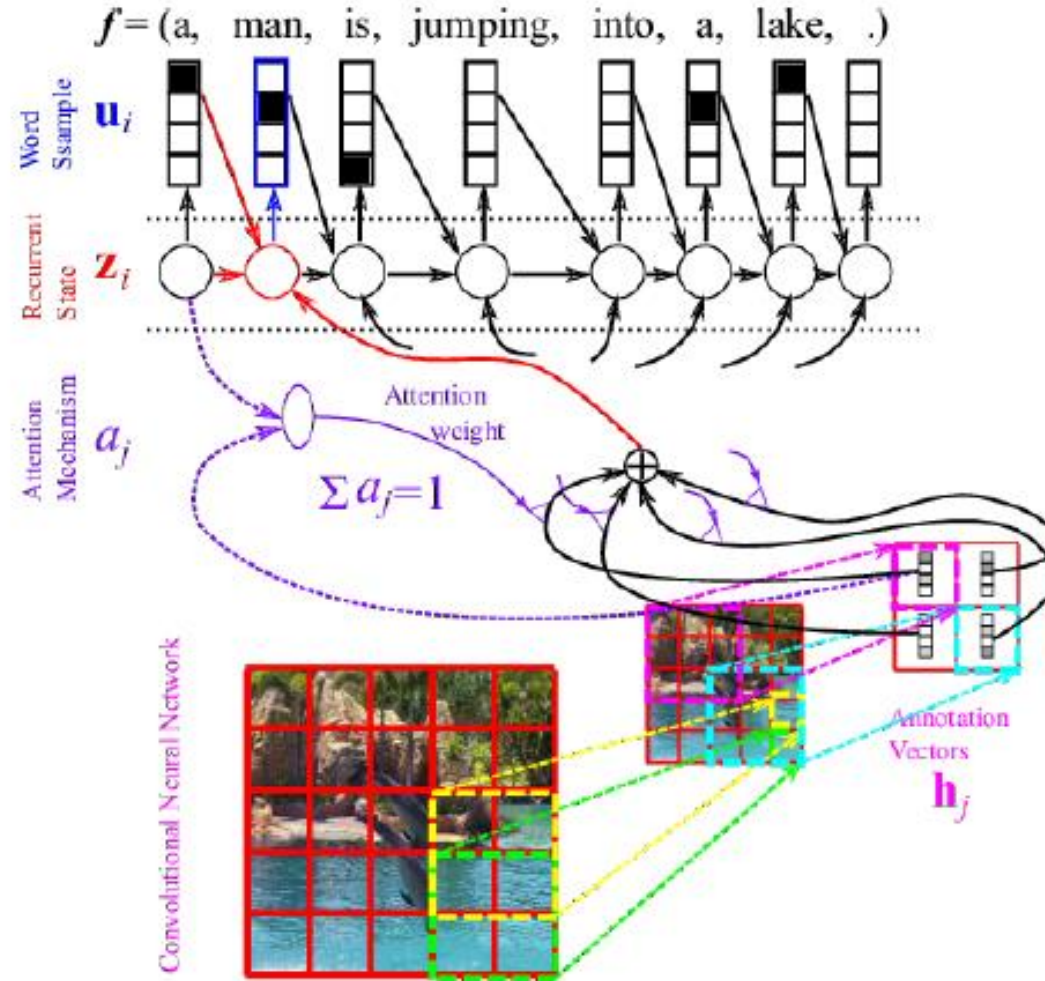
It is known , that the verb often occupies the last position in German sentences



Es ist bekannt , dass das Verb oft die letzte Position in deutschen Strafen einnimmt

This diagram illustrates the attention mechanism for the second sentence pair. It shows vertical lines for each word in the English sentence (top) and the German sentence (bottom). Curved lines represent the attention weights between the words. The English words 'It is known', 'that', 'the', 'verb', 'often', 'occupies', 'the', 'last', 'position', 'in', 'German', and 'sentences' are aligned with the German words 'Es ist bekannt', 'dass', 'das', 'Verb', 'oft', 'die', 'letzte', 'Position', 'in', 'deutschen', 'Strafen', and 'einnimmt' respectively. The English word 'It' is aligned with the German word 'Es', and the English word 'is' is aligned with the German word 'ist'.

Applications with Attention Model



Applications with Attention Model

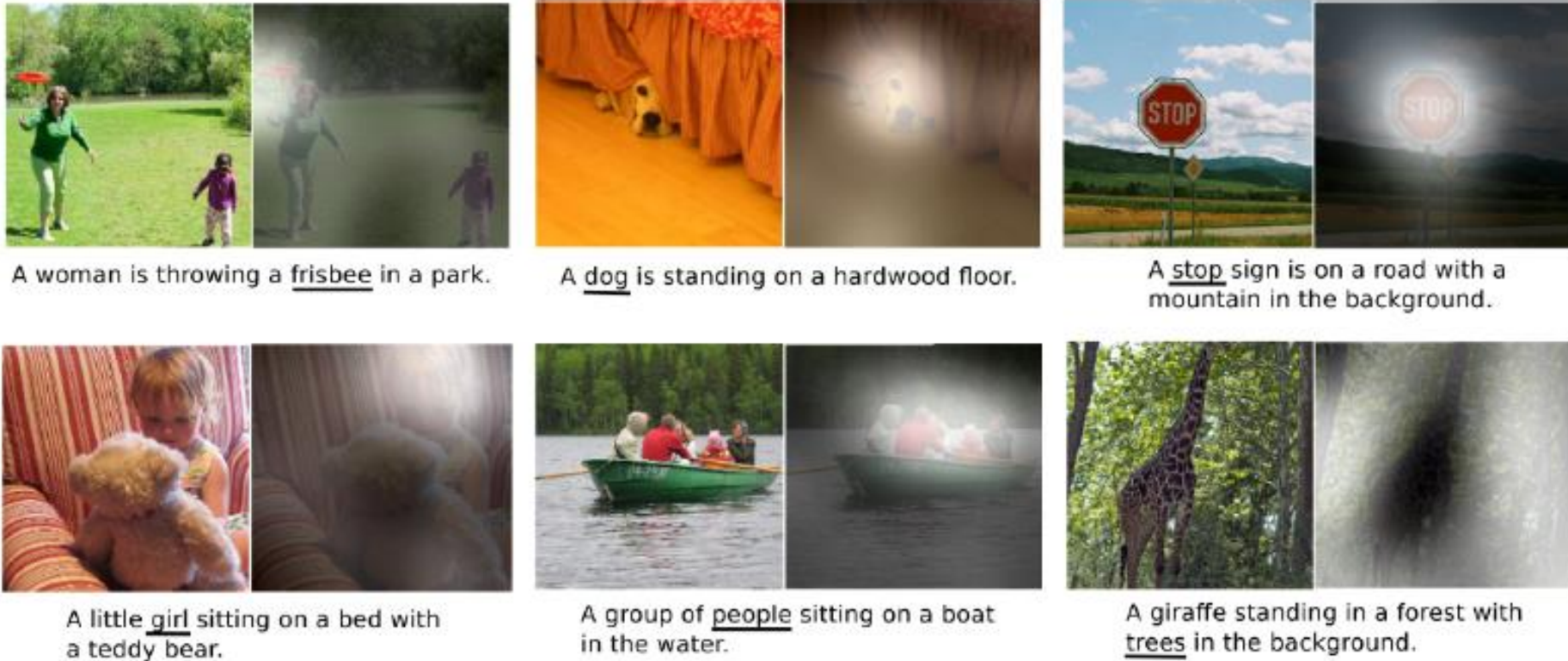


Fig. 5. Examples of the attention-based model attending to the correct object (*white* indicates the attended regions, *underlines* indicated the corresponding word) 22

Decoding for NMT

exact search

- generate every possible sentence T in target language
- compute score $p(T|S)$ for each
- pick best one

- intractable: $|\text{vocab}|^N$ translations for output length N
→ we need approximative search strategy

Decoding for NMT

approximative search/1

- at each time step, compute probability distribution $P(y_i|X, y_{<i})$
 - select y_i according to some heuristic:
 - sampling: sample from $P(y_i|X, y_{<i})$
 - greedy search: pick $\operatorname{argmax}_y p(y_i|X, y_{<i})$
 - continue until we generate $\langle eos \rangle$
-
- efficient, but suboptimal

Decoding for NMT

approximative search/2: **beam search**

- maintain list of K hypotheses (beam)
- at each time step, expand each hypothesis k : $p(y_i^k | X, y_{<i}^k)$
- select K hypotheses with highest total probability:

$$\prod_i p(y_i^k | X, y_{<i}^k)$$

- relatively efficient
- currently default search strategy in neural machine translation
- small beam ($K \approx 10$) offers good speed-quality trade-off

Ensemble NMT

- at each timestep, combine the probability distribution of M different ensemble components.
- combine operator: typically average (log-)probability

$$\log P(y_i|X, y_{<i}) = \frac{\sum_{m=1}^M \log P_m(y_i|X, y_{<i})}{M}$$

- requirements:
 - same output vocabulary
 - same factorization of Y
- internal network architecture may be different
- source representations may be different
(extreme example: ensemble-like model with different source languages [Junczys-Dowmunt and Grundkiewicz, 2016])

Advancing NMT

- Advancing NMT
- Future of NMT

Advancing NMT

- The vocabulary aspect
 - Goal: extend the vocabulary coverage.
- The memory aspect
 - Goal: translate long sentences better.
- The language complexity aspect
 - Goal: handle more language variations.
- The data aspect
 - Goal: utilize more data sources.

Future of NMT

- Multi-task learning
- Larger context
- Mobile devices
- Beyond Maximum Likelihood Estimation

Open Source NMT

- Nematus:
 - Developed by University of Edinburgh
 - <https://github.com/rsennrich/nematus>
 - Used to be based on Theano platform, now is tensorflow
- OpenNMT
 - Developed by Harvard University and Systran
 - <http://opennmt.net/>
 - Initially based on Lua (Torch), now PyTorch and TensorFlow implementations
- MarianNMT
 - Fast, low-level implementation
 - <https://marian-nmt.github.io/>

References Mathematics

- The main content is from:
 - <http://neuralnetworksanddeeplearning.com/chap1.html>
- Other references:
 - <http://sebastianruder.com/optimizing-gradient-descent/>
 - [CS231n: Convolutional Neural Networks for Visual Recognition](#)
 - www.cse.scu.edu/~tschwarz/coen266_09/PPT/Artificial%20Neural%20Networks.ppt
 - https://en.wikipedia.org/wiki/Deep_learning
 - https://en.wikipedia.org/wiki/Artificial_neural_network
 - https://en.wikipedia.org/wiki/Activation_function

References NMT

- The main content is from:

- <http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/>
- <http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-2-implementing-a-language-model-rnn-with-python-numpy-and-theano/>
- <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>
- <https://devblogs.nvidia.com/parallelforall/introduction-neural-machine-translation-gpus-part-3/>
- Rico Sennrich et al. Advances in Neural Machine Translation
- Thang Luong et al. Neural Machine Translation. Tutorial at ACL2016.
- Rico Sennrich. Neural Machine Translation
- <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>
- <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>
- Dzmitry Bahdanau et al. Neural Machine Translation by Jointly Learning to Align and Translate
- Rico Sennrich. Neural Machine Translation: Breaking the Performance Plateau
- <http://jalammar.github.io/visualizing-neural-machine-translation-mechanics-of-seq2seq-models-with-attention/>