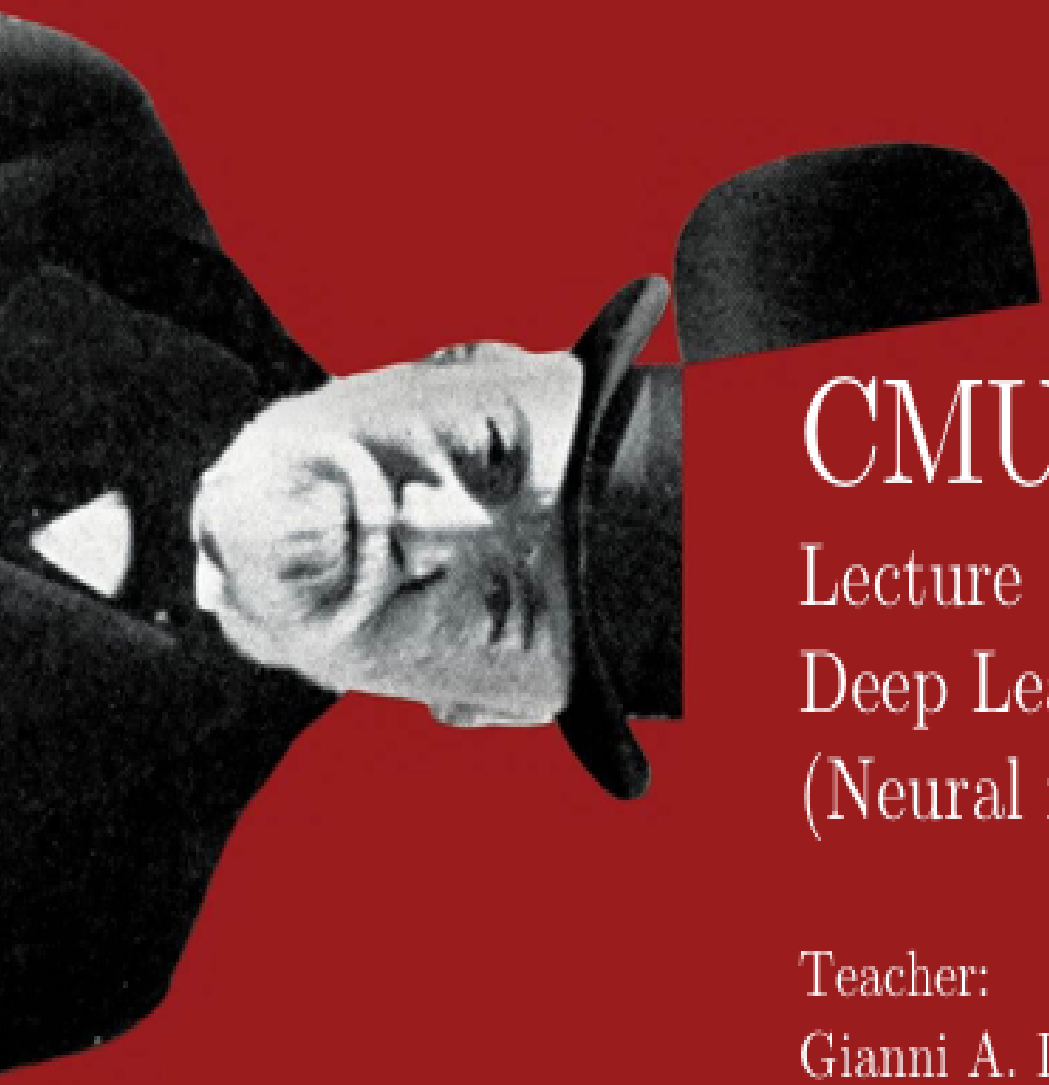


CMU 15-781

Lecture 17:
Deep Learning
(Neural networks)

Teacher:
Gianni A. Di Caro

OUTLINE

OUTLINE

SUPERVISED MACHINE LEARNING CLASSIFICATION SETUP

- Input features $x^{(i)} \in \mathbb{R}^n$
- Outputs $y^{(i)} \in \mathcal{Y}$ (e.g. $\mathbb{R}$, $\{-1, +1\}$, $\{1, \ldots, p\}$)
- Model parameters $\theta \in \mathbb{R}^k$
- Hypothesis function $h_\theta : \mathbb{R}^n \to \mathbb{R}$
- Loss function $\ell : \mathbb{R} \times \mathcal{Y} \to \mathbb{R}_+$
- Machine learning optimization problem

$$\underset{\theta}{\text{minimize}} \quad \sum_{i=1}^{m} \ell(h_\theta(x^{(i)}), y^{(i)})$$

LINEAR CLASSIFIERS

- Linear hypothesis class:

$$h_\theta(x^{(i)}) = \boldsymbol{\theta}^T \boldsymbol{\phi}(\boldsymbol{x}^{(i)})$$

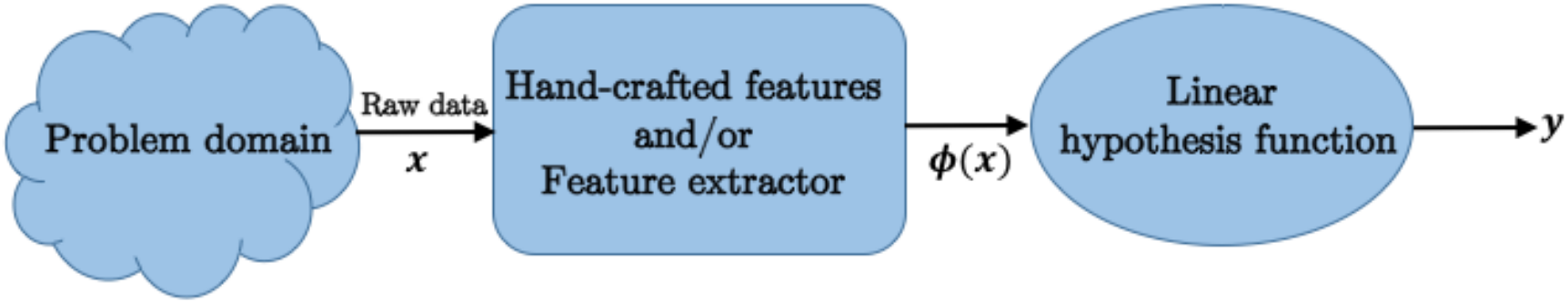
  where the input can be any set of **non-linear features** $\phi : \mathbb{R}^n \to \mathbb{R}^k$

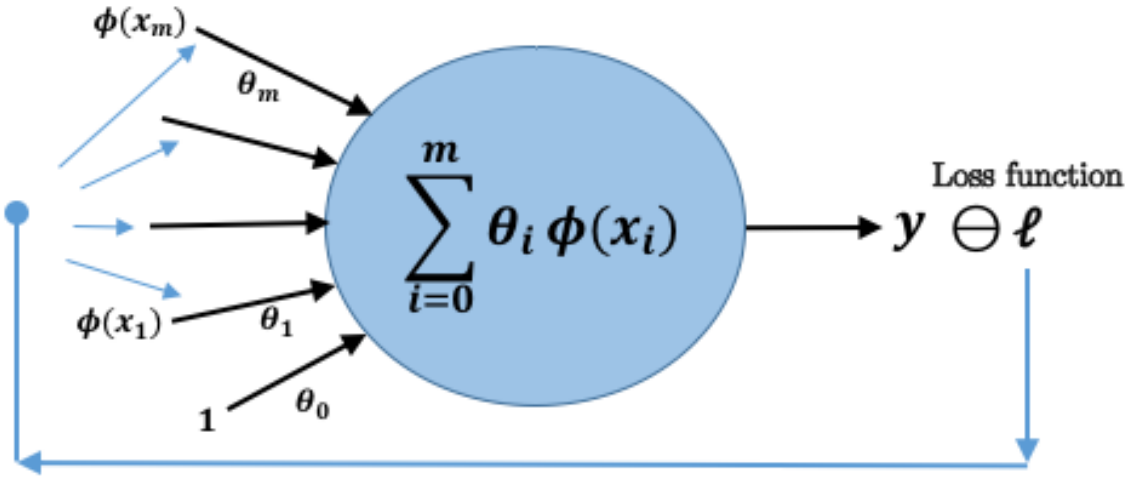
- The generic function ϕ represents some (possibly) selected way to generate non-linear features out of the available ones, for instance:

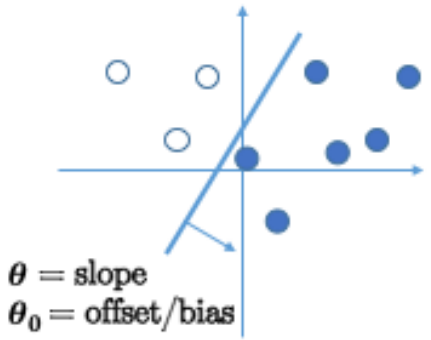
$$x^{(i)} = [\text{temperature for day } i]$$

$$\phi(x^{(i)}) = \begin{bmatrix} 1 \\ x^{(i)} \\ {x^{(i)}}^2 \\ \\ \vdots \end{bmatrix}$$

GENERAL REPRESENTATION OF LINEAR CLASSIFICATION







CHALLENGES WITH LINEAR MODELS

- Linear models crucially depend on choosing "good" features
- Some "standard" choices: polynomial features, radial basis functions, random features (surprisingly effective)
- But, many specialized domains required highly engineered special features
  - E.g., computer vision tasks used Haar features, SIFT features, every 10 years or so someone would engineer a new set of features
- **Key question 1:** Should we stick with linear hypothesis functions? What about using non-linear combinations of the inputs? $\rightarrow$ *Feed-forward neural networks (Perceptrons)*
- **Key question 2:** can we come up with algorithms that will automatically *learn* the features themselves? $\rightarrow$ *Feed-forward neural networks with multiple ($>$ 2!) hidden layers (Deep Networks)*

FEATURE LEARNING: USE TWO CLASSIFIERS IN CASCADE

- Instead of a simple linear classifier, let's consider a **two-stage** hypothesis class where one linear function creates the features, another models the classifier and takes as input the features created by the first one:

$$h_{\boldsymbol{w}}(\boldsymbol{x}) = W_2\phi(\boldsymbol{x}) + b_2 = W_2(W_1\boldsymbol{x} + b_1) + b_2$$

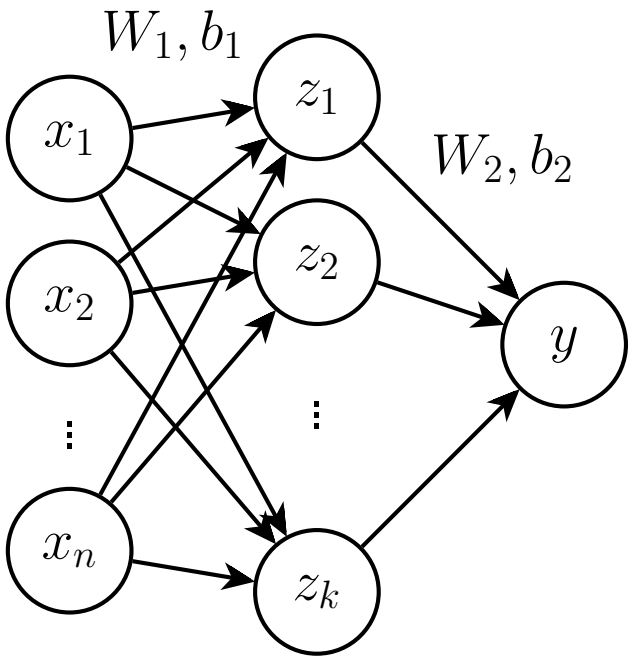
  where

$$\boldsymbol{w} = \{W_1 \in \mathbb{R}^{n\times k}, b_1 \in \mathbb{R}^k, W_2 \in \mathbb{R}^{1\times k}, b_2 \in \mathbb{R}\}$$

- Note that in this notation, we're explicitly separating the parameters on the "constant feature" into the b terms

FEATURE LEARNING: USE TWO CLASSIFIERS IN CASCADE

- Graphical depiction of the obtained function



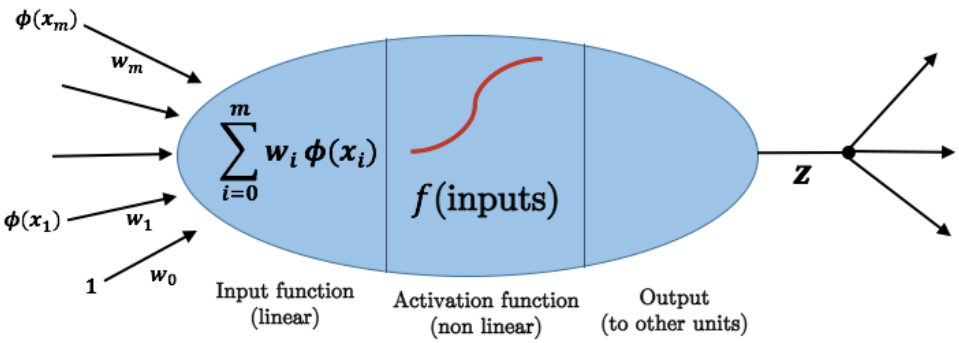
- But there is a problem:

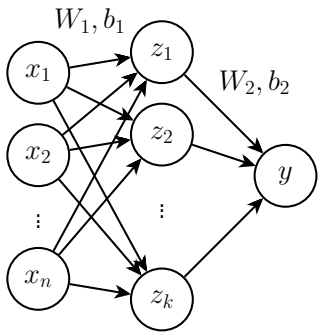
$$h_{\boldsymbol{w}}(\boldsymbol{x}) = W_2(W_1\boldsymbol{x} + b_1) + b_2 = \tilde{\boldsymbol{W}}\boldsymbol{x} + \tilde{b} \tag{1}$$

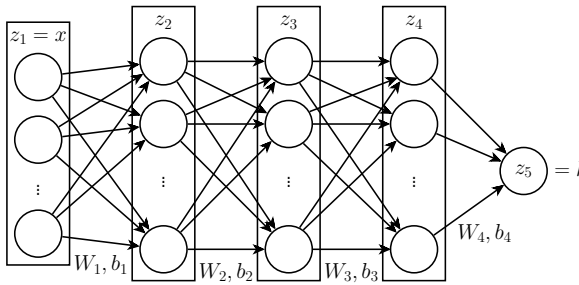
in other words, we are still just using a normal linear classifier: the apparent added complexity by concatenating multiple is not giving us any additional representational power, we can only discriminate linearly separable classes

ARTIFICIAL NEURAL NETWORKS (ANN)

- *Neural networks* provide a way to obtain complexity by:
  1. Using non linear transformations of the inputs
  2. Propagating the information among layers of processing units to realize multi-staged computation
  3. In *deep networks*, the number of stages is relatively large, allowing to automatically learn hierarchical representations of the data features







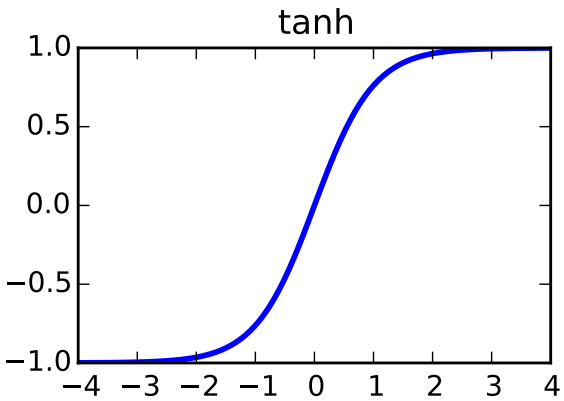
TYPICAL NON-LINEAR ACTIVATION FUNCTIONS

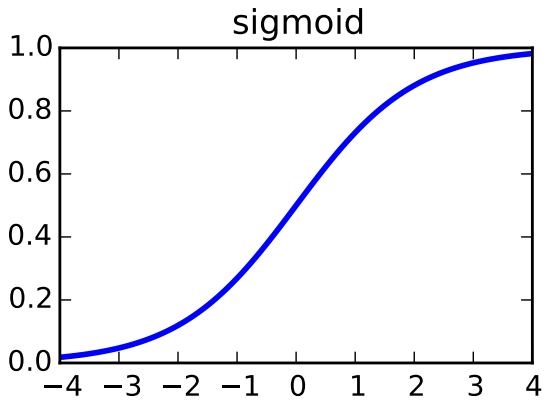
- Using non-linear activation functions at each node, the two-layer network of the previous example, become equivalent to have the following hypothesis function:

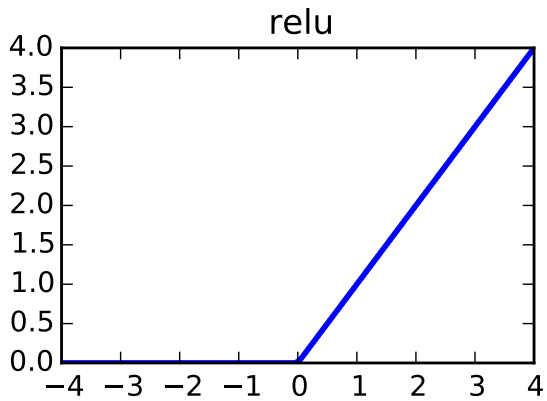
$$h_\theta(x) = f_2(W_2 f_1(W_1 x + b_1) + b_2)$$

where $f_1, f_2 : \mathbb{R} \to \mathbb{R}$ are some non-linear functions (applied elementwise to vectors)

- Common choices for f_i are **hyperbolic tangent** $\tanh(x) = (e^{2x} - 1)/(e^{2x} + 1)$, **sigmoid/logistic** $\sigma(x) = 1/(1 + e^{-x})$, or **rectified linear unit** $f(x) = \max\{0, x\}$

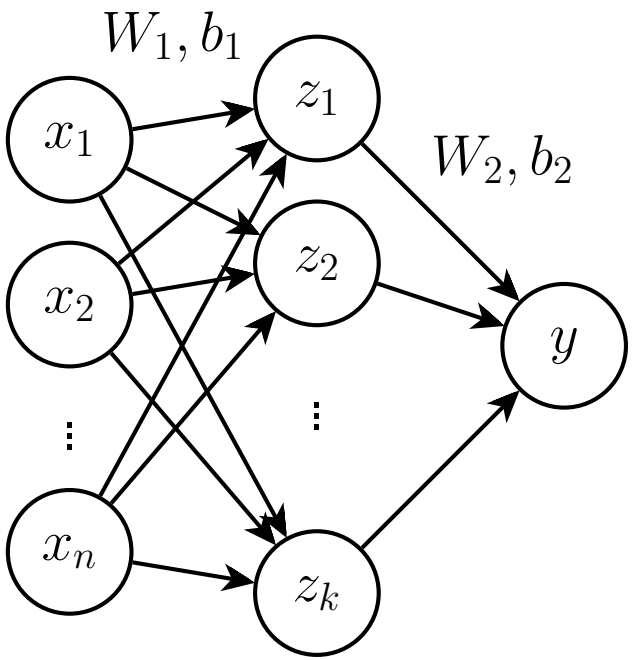






HIDDEN LAYERS AND LEARNED FEATURES

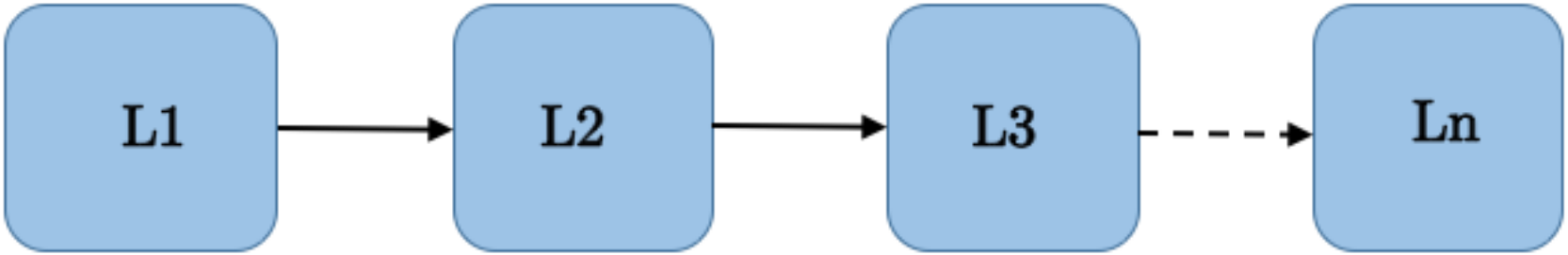
- We draw these the same as before (non-linear functions are virtually always implied in the neural network setting)

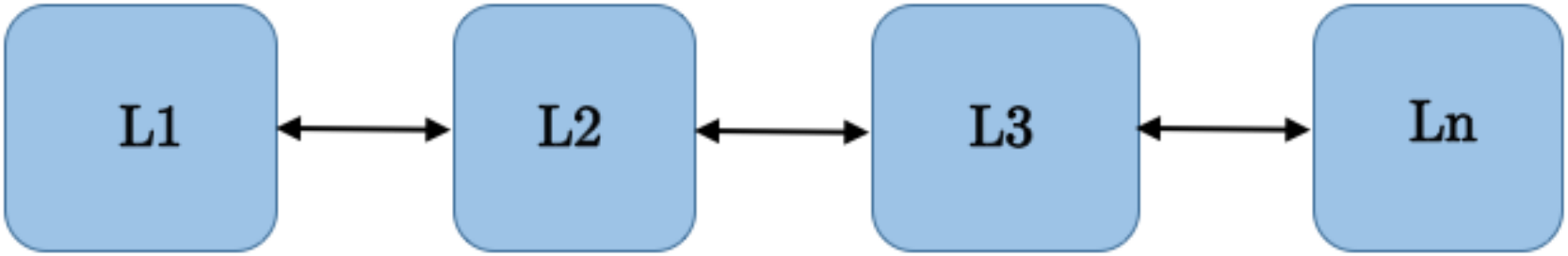


- Middle layer z is referred to as the **hidden layer** or **activations**
- These are the learned features, nothing in the data that prescribes what values these should take, left up to the algorithm to decide
- To have a meaningful feature learning we need multiple hidden layers in cascade
- Networks

TYPES OF NETWORKS

Feed-forward networks (multilayer perceptrons)





Bi-directional networks (recurrent networks, Hopfield networks)

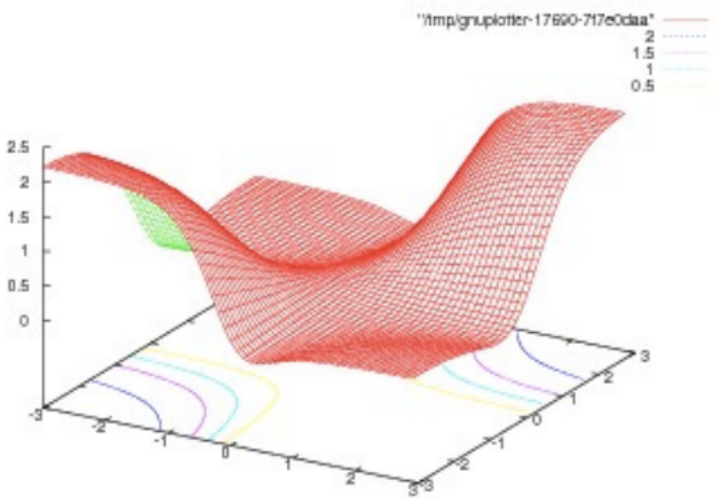
PROPERTIES OF NEURAL NETWORKS

- It turns out that a neural network with a single hidden layer (and a suitably large number of hidden units) is a **universal function approximator**, can approximate *any* function over the input arguments (but this is actually not very useful in practice, c.f. polynomials fitting any sets of points for high enough degree)

- The hypothesis class h_θ is **not a convex function of the parameters** $\theta = \{W_i, b_i\}$

  - **Example: what is the loss function for the simplest 2-layer neural net ever**
    - Function: 1-1-1 neural net. Map 0.5 to 0.5 and -0.5 to -0.5 (identity function) with quadratic cost:

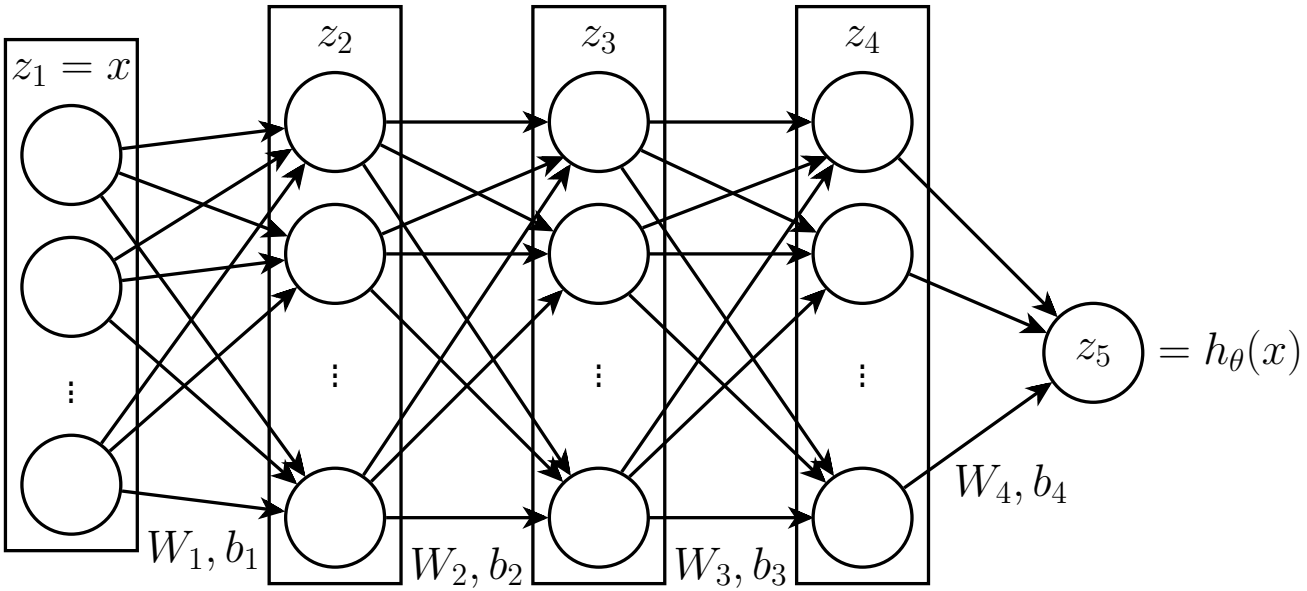
$$y = \tanh(W_1 \tanh(W_0.x)) \quad L = (0.5 - \tanh(W_1 \tanh(W_0 0.5)^2$$



- The number of parameters (weights and biases), layers (depth), topology (connectivity), activation functions, all affect the performance and capacity of the network

DEEP LEARNING

- "Deep" neural networks refer to networks with multiple hidden layers



- Mathematically, a k-layer network has the hypothesis function

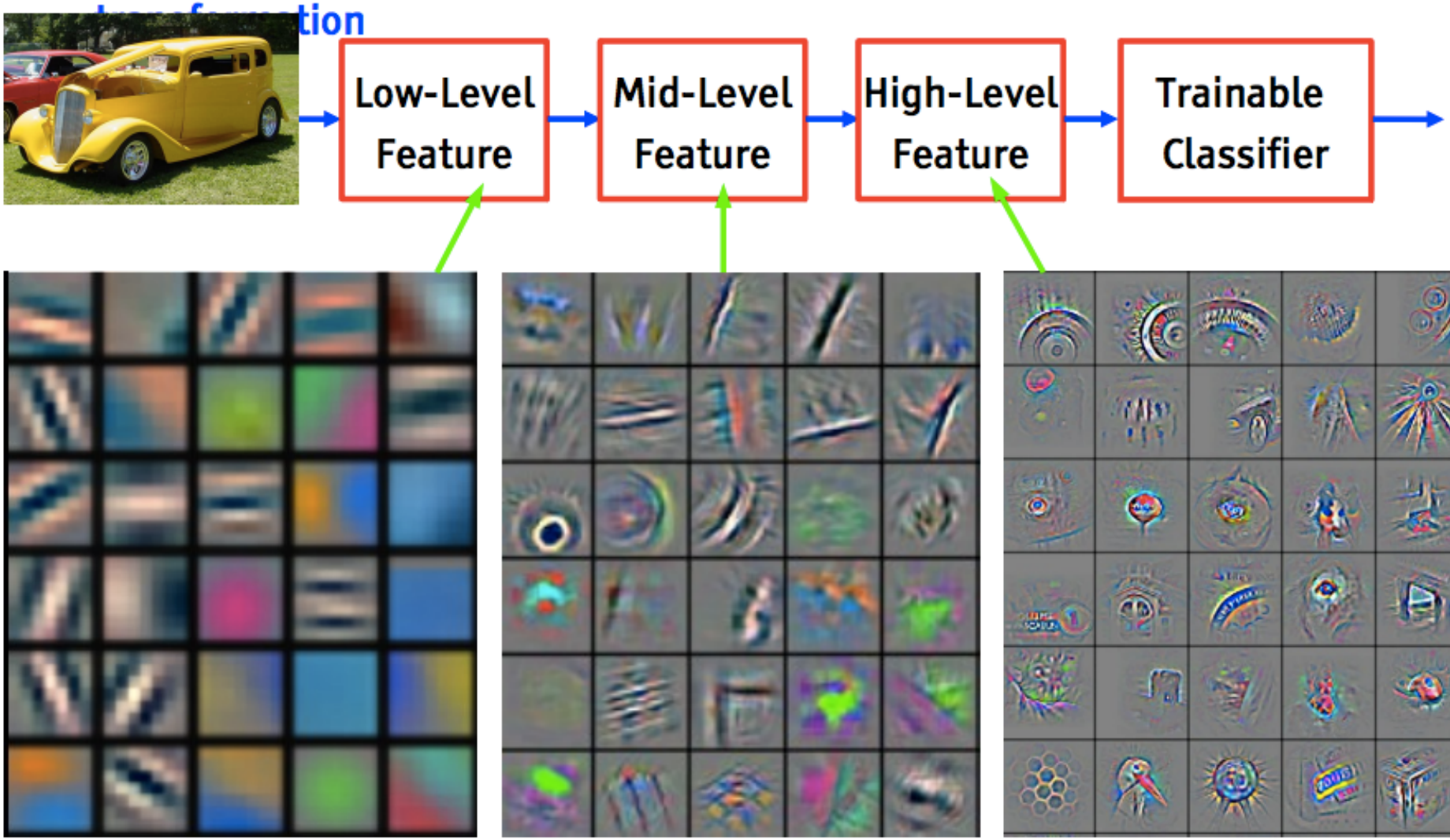
$$\boldsymbol{z}_{i+1} = f_i(\boldsymbol{W}_i \boldsymbol{z}_i + b_i), \quad i = 1, \ldots, k-1, \ \boldsymbol{z}_1 = \boldsymbol{x}$$
$$h_{\boldsymbol{w}}(\boldsymbol{x}) = \boldsymbol{z}_k$$

where $\boldsymbol{z}_i$ terms now indicate *vectors* of output features

WHY USE DEEP NETWORKS?

- A deep architecture trades space for time (or breadth for depth): more layers (more sequential computation), but less hardware (less parallel computation).
- Many functions can be represented more compactly using deep networks than one-hidden layer networks (e.g. parity function would require (2^n) hidden units in 3-layer network, $O(n)$ units in $O(\log n)$-layer network)
- Motivation from neurobiology: brain appears to use multiple levels of interconnected neurons to process information (but careful, neurons in brain are not just non-linear functions)
- In practice: works better for many domains
- Allow for automatic hierarchical feature extraction from the data

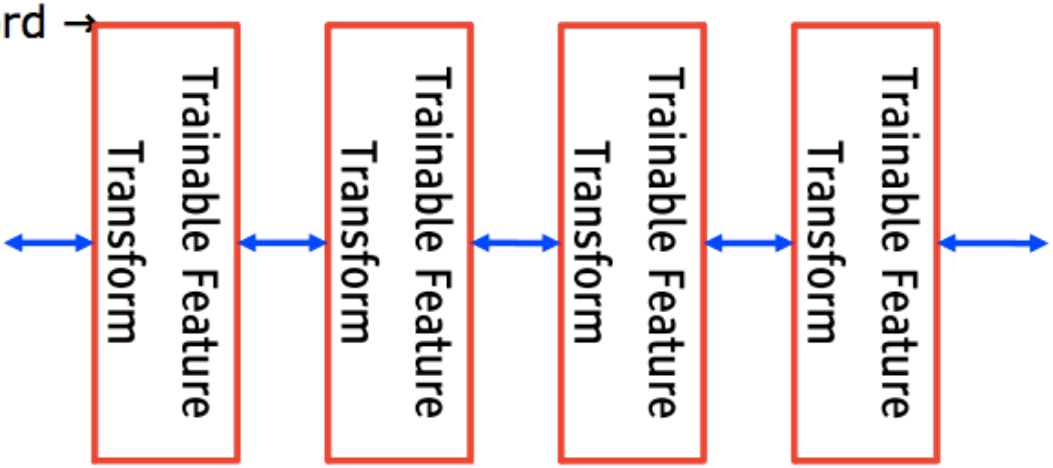
HIERARCHICAL FEATURE REPRESENTATION



Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

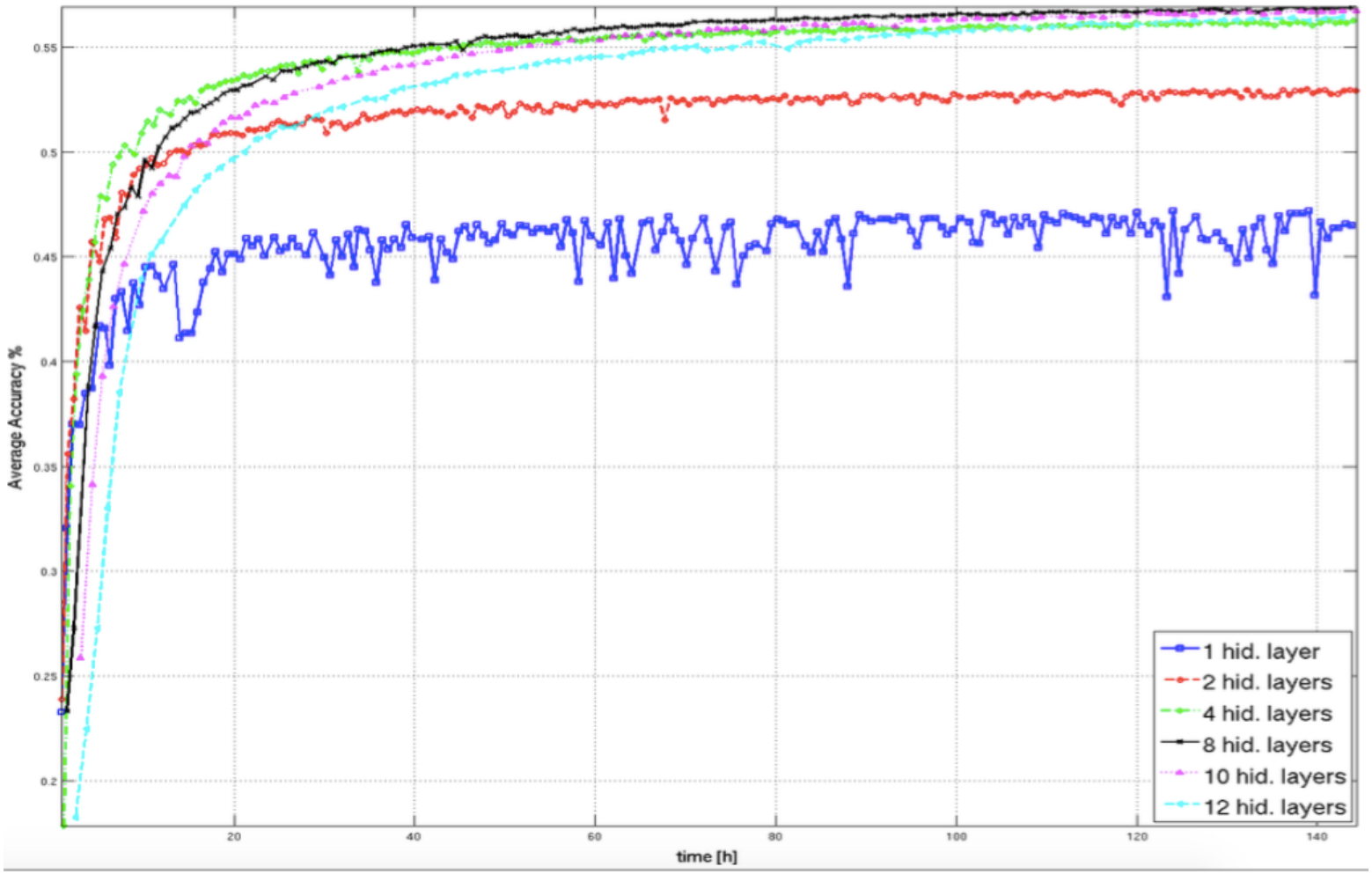
EXAMPLES OF HIERARCHICAL FEATURE REPRESENTATION

- Hierarchy of representations with increasing level of abstraction
- Each stage is a kind of trainable feature transform
- Image recognition
  - Pixel → edge → texton → motif → part → object
- Text
  - Character → word → word group → clause → sentence → story
- Speech
  - Sample → spectral band → sound → … → phone → phoneme → word →



EFFECT OF INCREASING NUMBER OF HIDDEN LAYERS

- Speech recognition task



OUTLINE

OPTIMIZING NEURAL NETWORK PARAMETERS

- How do we optimize the parameters for the machine learning loss minimization problem with a neural network

$$\underset{\theta}{\text{minimize}} \quad \sum_{i=1}^{m} \ell(h_\theta(x^{(i)}), y^{(i)})$$

  now that this problem is non-convex?

- Just do exactly what we did before: initialize with random weights and run stochastic gradient descent

- Now have the possibility of local optima, and function can be harder to optimize, but we won't worry about all that because the resulting models still often perform better than linear models

STOCHASTIC GRADIENT DESCENT FOR NEURAL NETWORKS

- Recall that stochastic gradient descent computes gradients with respect to loss on each example, updating parameters as it goes

function SGD($\{(x^{(i)}, y^{(i)})\}, h_\theta, \ell, \alpha$)
  Initialize: $W_j, b_j \leftarrow$ Random, $j = 1, \ldots, k$
  Repeat until convergence:
    For $i = 1, \ldots, m$:
      Compute $\nabla_{W_j, b_j} \ell(h_\theta(x^{(i)}), y^{(i)}),\ j = 1, \ldots, k-1$
      Take gradient steps in all directions:
        $W_j \leftarrow W_j - \alpha \nabla_{W_j} \ell(h_\theta(x^{(i)}), y^{(i)}),\quad j = 1, \ldots, k$
        $b_j \leftarrow b_j - \alpha \nabla_{b_j} \ell(h_\theta(x^{(i)}), y^{(i)}),\quad j = 1, \ldots, k$
  return $\{W_j, b_j\}$

- How do we compute the gradients $\nabla_{W_j, b_j} \ell(h_\theta(x^{(i)}), y^{(i)})$?

BACK-PROPAGATION

- Back-propagation is a method for computing all the necessary gradients using one **forward pass** (just computing all the activation values at layers), and one **backward pass** (computing gradients backwards in the network)

- The equations sometimes look complex, but it's just an application of the **chain rule** of calculus and the use of **Jacobians**

JACOBIANS AND CHAIN RULE

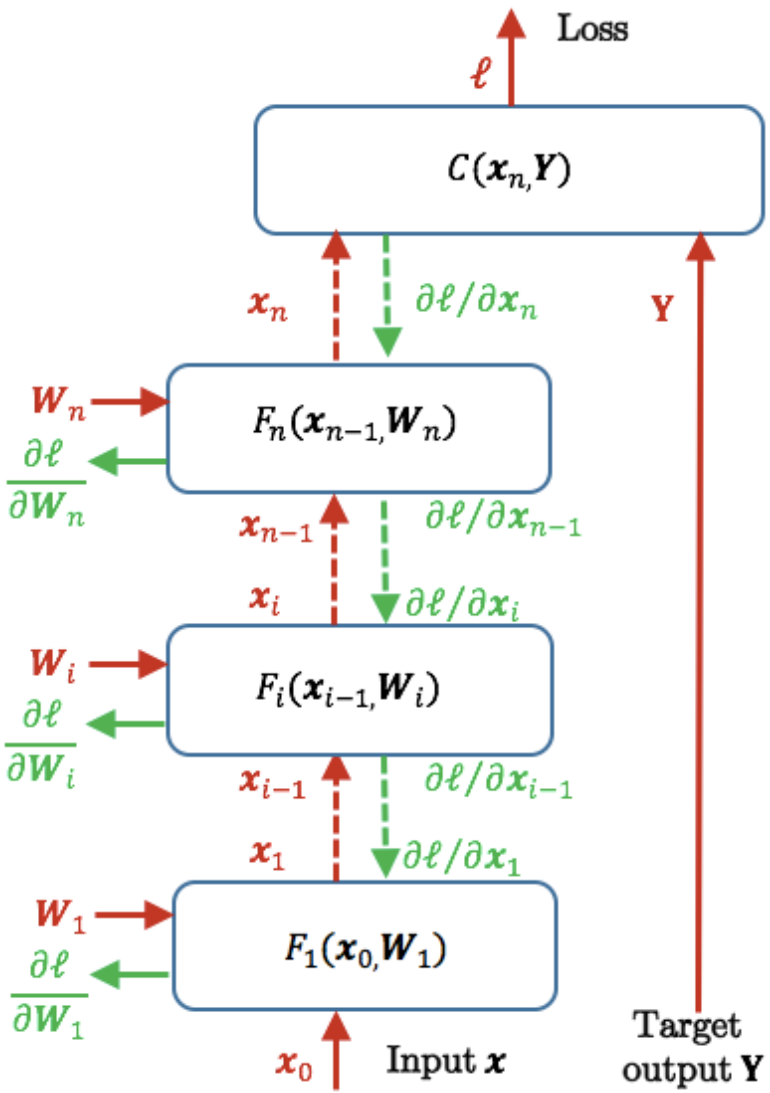
- For a multivariate, vector-valued function $f : \mathbb{R}^n \to \mathbb{R}^m$, the **Jacobian** is a $m \times n$ matrix

$$\left(\frac{\partial f(x)}{\partial x}\right) \in \mathbb{R}^{m \times n} = \begin{bmatrix} \frac{\partial f_1(x)}{\partial x_1} & \frac{\partial f_1(x)}{\partial x_2} & \dots & \frac{\partial f_1(x)}{\partial x_n} \\ \frac{\partial f_2(x)}{\partial x_1} & \frac{\partial f_2(x)}{\partial x_2} & \dots & \frac{\partial f_2(x)}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m(x)}{\partial x_1} & \frac{\partial f_m(x)}{\partial x_2} & \dots & \frac{\partial f_m(x)}{\partial x_n} \end{bmatrix}$$

- For a scalar-valued function $f : \mathbb{R}^n \to \mathbb{R}$, the Jacobian is the transpose of the gradient $\frac{\partial f(x)}{\partial x}^T = \nabla_x f(x)$
- For a vector-valued function, row i of the Jacobian corresponds to the gradient of the component f_i of the output vector, $i = 1, \dots, m$. It tells how the variation of each input variable affects the variation of the output component
- Column j of the Jacobian is the impact of the variation of the j-th input variable, $j = 1, \dots, n$, on each one of the m components of the output
- **Chain rule** for the derivation of a composite function:

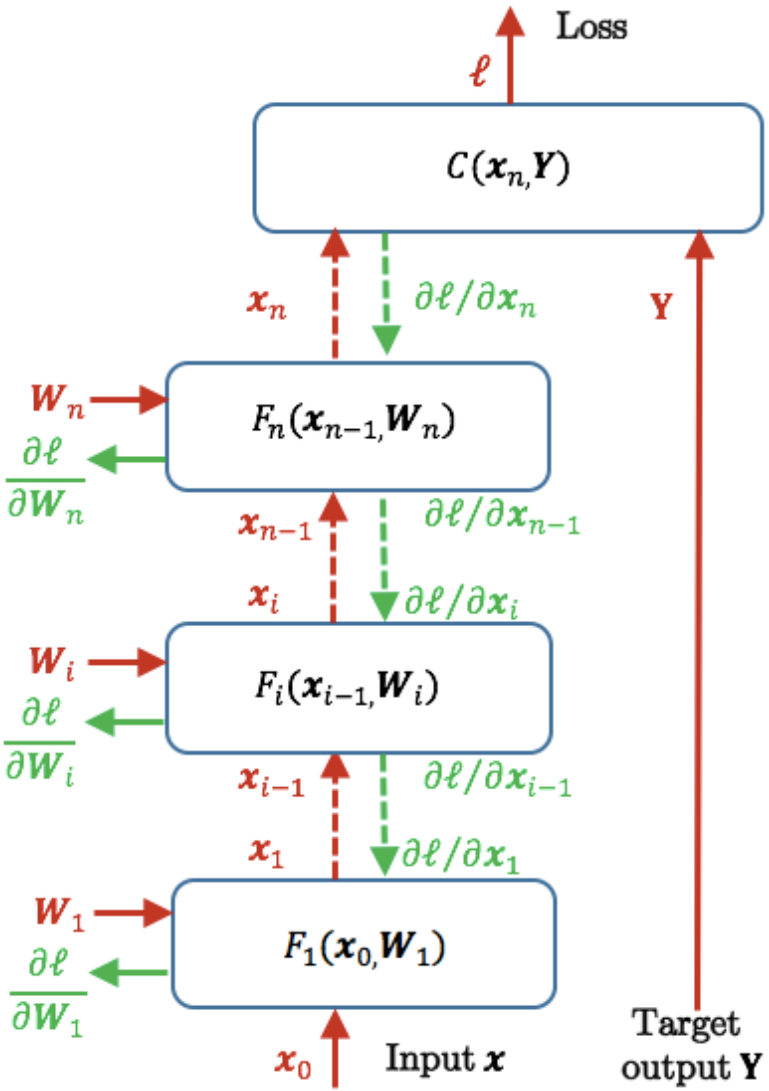
$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial f(g(x))}{\partial g(x)} \frac{\partial g(x)}{\partial x}$$

MULTI-LAYER / MULTI-MODULE FF



- **Multi-layered feed-forward architecture** (cascade): module/layer i gets as input the feature vector $\boldsymbol{x}_{i-1}$ output from module $i-1$, applies the transformation function $F_i(\boldsymbol{x}_{i-1}, \boldsymbol{W}_i)$, and produces the output $\boldsymbol{x}_i$
- Each layer i contains $(N_{\boldsymbol{x}})_i$ parallel nodes $(\boldsymbol{x}_i)_j, \;\; j = 1, \ldots, (N_{\boldsymbol{x}})_i$. Each node gets the input from previous layer's nodes through $(N_{\boldsymbol{x}})_{i-1} \equiv (N_{\boldsymbol{W}})_i$ connections with weights $(\boldsymbol{W}_i)_j, \;\; j = 1, \ldots, (N_{\boldsymbol{W}})_i$
- Each layer *learns* its own weight vector $\boldsymbol{W}_i$.
- F_i is a vector function. At each node j of layer i, the transformation function is $(F_i)_j = f((\boldsymbol{W}_i)_j^T \cdot (\boldsymbol{x}_{i-1})_j)$ and f is the *activation function* (e.g., a sigmoid), that can be safely considered being the same for all nodes.
- In the following the notation is made simpler by dropping the second indices and reasoning at the aggregate vector level of each layer.

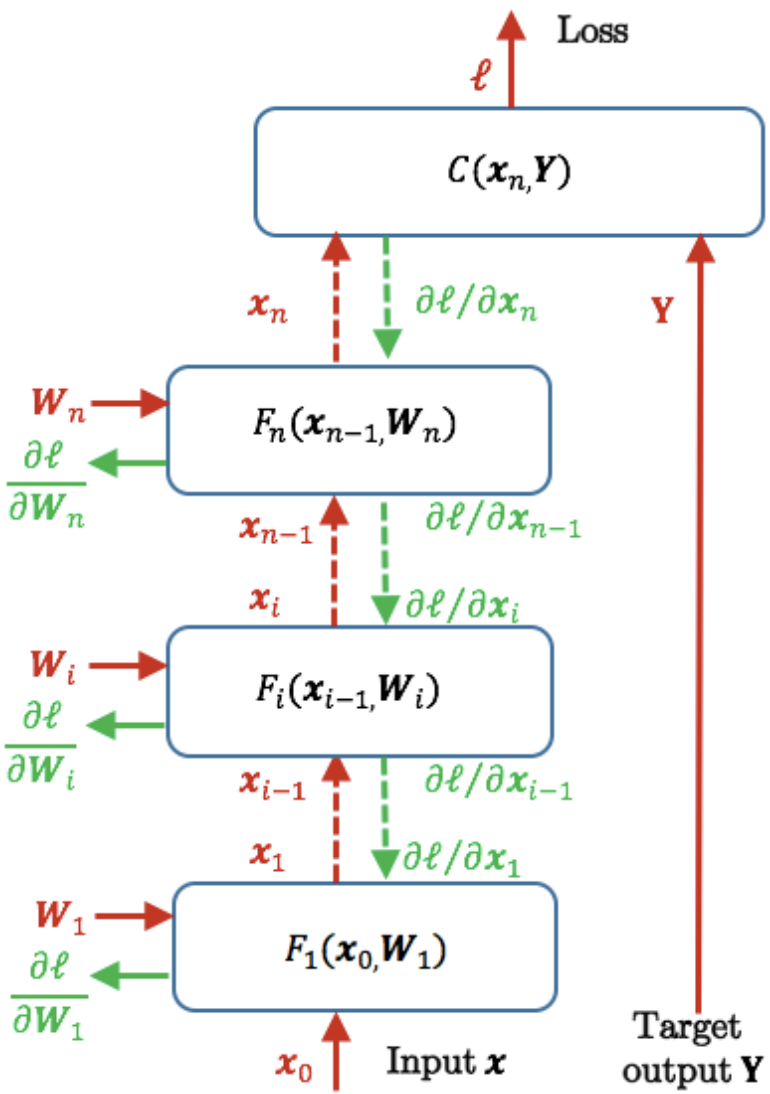
FORWARD PROPAGATION



- **Forward Propagation:**
  - Following the presentation of the training input $\boldsymbol{x}_0$, the output vectors $\boldsymbol{x}_i$ resulting from the activation function F_i at all layers $i = 1, \ldots, n$, are computed in *sequence*, starting from $\boldsymbol{x}_1$, and are stored
  - The output of the network, the *loss* ℓ (to be minimized), results from the forward propagation at output layer and computing the deviation with respect to the target $\boldsymbol{Y}$:

$$\ell(\boldsymbol{Y}, \boldsymbol{x}, \boldsymbol{W}) = C(\boldsymbol{x}_n, \boldsymbol{Y})$$

COMPUTING GRADIENTS

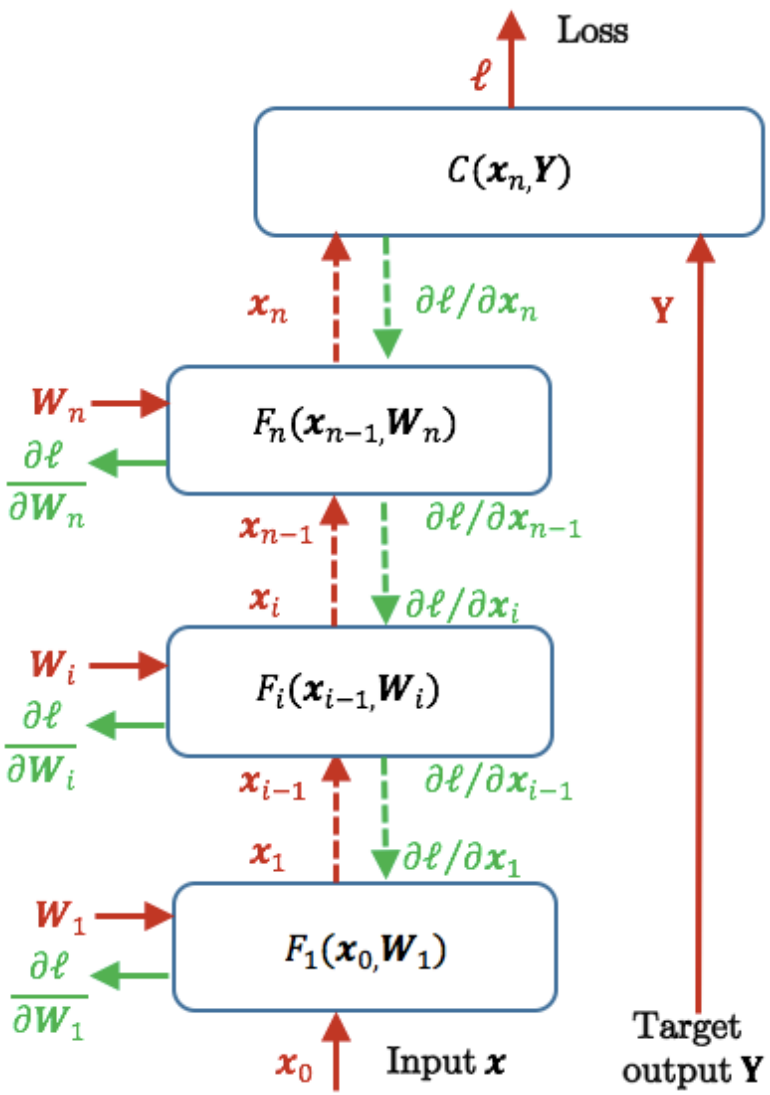


- At each iteration of SGD, the gradients with respect to all the parameters of the system need to be computed (i.e., the weights $\boldsymbol{W}_i$, that could be split in weights for input and weights for bias, but hereafter we just use the general form $\boldsymbol{W}_i$)
- After the *Forward pass*, let's start setting up the relations for the *Backward pass*
- Let's consider the generic layer i: from the Forward propagation, its output value is available and is $\boldsymbol{x}_i = F_i(\boldsymbol{x}_{i-1}, \boldsymbol{W}_i)$
- In addition, let's assume that we already know

$$\frac{\partial \ell}{\partial \boldsymbol{x}_i},$$

that is, we know for each component of the vector $\boldsymbol{x}_i$ the variation of ℓ in relation to a variation of $\boldsymbol{x}_i$. We can assume that we know $\frac{\partial \ell}{\partial \boldsymbol{x}_i}$ since we will proceed backward

COMPUTING GRADIENTS



- Since we assume as known $\frac{\partial \ell}{\partial \boldsymbol{x}_i}$ and we have computed $\boldsymbol{x}_i = F_i(\boldsymbol{x}_{i-1}, \boldsymbol{W}_i)$, we can use the *chain rule* to compute $\frac{\partial \ell}{\partial \boldsymbol{W}_i}$, which is the quantity of interest, and which tells us the variation in ℓ as a response to a variation in the weights of $\boldsymbol{W}_i$:

$$\frac{\partial \ell}{\partial \boldsymbol{W}_i} = \frac{\partial \ell}{\partial \boldsymbol{x}_i} \frac{\partial F_i(\boldsymbol{x}_{i-1}, \boldsymbol{W}_i)}{\partial \boldsymbol{W}_i}$$

  where $\boldsymbol{x}_i$ is a substitute for $F_i(\boldsymbol{x}_{i-1}, \boldsymbol{W}_i)$

- Dimensionally, the previous equation is as follows:

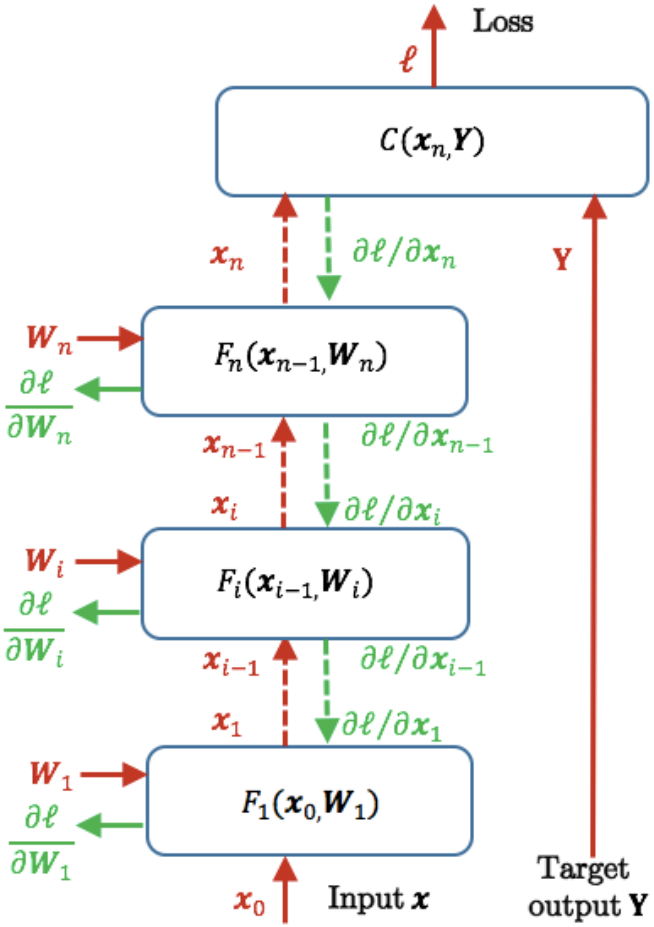
$$[1 \times N_{\boldsymbol{W}}] = [1 \times N_{\boldsymbol{x}}] \cdot [N_{\boldsymbol{x}} \times N_{\boldsymbol{W}}]$$

- $\frac{\partial F_i(\boldsymbol{x}_{i-1}, \boldsymbol{W}_i)}{\partial \boldsymbol{W}_i}$ is the *Jacobian matrix* of F_i with respect to $\boldsymbol{W}_i$

- The element (k, l) of the Jacobian quantifies the variation in the k-th output when a variation is exerted on the l-th weight

$$\left[\frac{\partial F_i(\boldsymbol{x}_{i-1}, \boldsymbol{W}_i)}{\partial \boldsymbol{W}_i}\right]_{kl} = \frac{\partial \left[F_i(\boldsymbol{x}_{i-1}, \boldsymbol{W}_i)\right]_k}{\partial \left[\boldsymbol{W}_i\right]_l}$$

COMPUTING GRADIENTS



- Let's keep assuming that we known $\frac{\partial \ell}{\partial \boldsymbol{x}_i}$ and let's use it this time to compute $\frac{\partial \ell}{\partial \boldsymbol{x}_{i-1}}$
- Applying the *chain rule*:

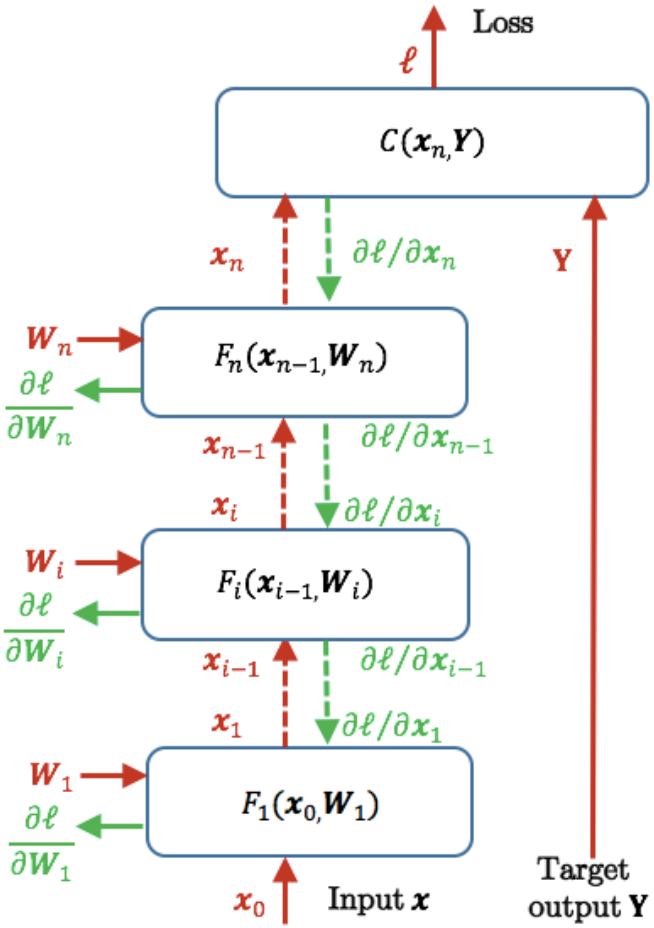
$$\frac{\partial \ell}{\partial \boldsymbol{x}_{i-1}} = \frac{\partial \ell}{\partial \boldsymbol{x}_i} \frac{\partial F_i(\boldsymbol{x}_{i-1}, \boldsymbol{W}_i)}{\partial \boldsymbol{x}_{i-1}}$$

- Dimensionally, the previous equation is as follows:

$$[1 \times N_{\boldsymbol{x}}] = [1 \times N_{\boldsymbol{x}}] \cdot [N_{\boldsymbol{x}} \times N_{\boldsymbol{x}}]$$

- $\frac{\partial F_i(\boldsymbol{x}_{i-1}, \boldsymbol{W}_i)}{\partial \boldsymbol{x}_{i-1}}$ is the *Jacobian matrix* of F_i with respect to $\boldsymbol{x}_{i-1}$
- The element (k, l) of the Jacobian quantifies the variation in the k-th output when a variation is exerted on the l-th input
- **The equation above is a recurrence equation!**

BACK-PROPAGATION (BP)



- To sequentially compute all the gradients needed by SGD, a backward sweep is applied, which is called the **back-propagation algorithm**, that precisely makes use of the recurrence equation for $\frac{\partial \ell}{\partial \boldsymbol{x}_i}$

1. $\dfrac{\partial \ell}{\partial \boldsymbol{x}_n} = \dfrac{\partial C(\boldsymbol{x}_n, \boldsymbol{Y})}{\partial \boldsymbol{x}_n}$
2. $\dfrac{\partial \ell}{\partial \boldsymbol{x}_{n-1}} = \dfrac{\partial \ell}{\partial \boldsymbol{x}_n} \dfrac{\partial F_n(\boldsymbol{x}_{n-1}, \boldsymbol{W}_n)}{\partial \boldsymbol{x}_{n-1}}$
3. $\dfrac{\partial \ell}{\partial \boldsymbol{W}_n} = \dfrac{\partial \ell}{\partial \boldsymbol{x}_n} \dfrac{\partial F_n(\boldsymbol{x}_{n-1}, \boldsymbol{W}_n)}{\partial \boldsymbol{W}_n}$
4. $\dfrac{\partial \ell}{\partial \boldsymbol{x}_{n-2}} = \dfrac{\partial \ell}{\partial \boldsymbol{x}_{n-1}} \dfrac{\partial F_{n-1}(\boldsymbol{x}_{n-2}, \boldsymbol{W}_{n-1})}{\partial \boldsymbol{x}_{n-2}}$
5. $\dfrac{\partial \ell}{\partial \boldsymbol{W}_{n-1}} = \dfrac{\partial \ell}{\partial \boldsymbol{x}_{n-1}} \dfrac{\partial F_{n-1}(\boldsymbol{x}_{n-2}, \boldsymbol{W}_{n-1})}{\partial \boldsymbol{W}_{n-1}}$
6. . . . until we reach the first, input layer
7. $\rightarrow$ all the gradients $\dfrac{\partial \ell}{\partial \boldsymbol{W}_i}$, $\forall i = 1, \ldots, n$ have been computed!

ACTIVATION FUNCTIONS EXAMPLES

- Remember that the transfer function at layer i is $F_i(\boldsymbol{x}_{i-1}, \boldsymbol{W}_i) = f\left(\boldsymbol{W}_i^T \cdot \boldsymbol{x}_{i-1}\right)$, and for the j-th neuron in layer i,

$$(F_i)_j = f\left((\boldsymbol{W}_i)_j^T \cdot (\boldsymbol{x}_{i-1})_j\right) = \sum_{j=1}^{(N_{\boldsymbol{x}})_{i-1}} w_{ij} \cdot x_{i-1,j}$$

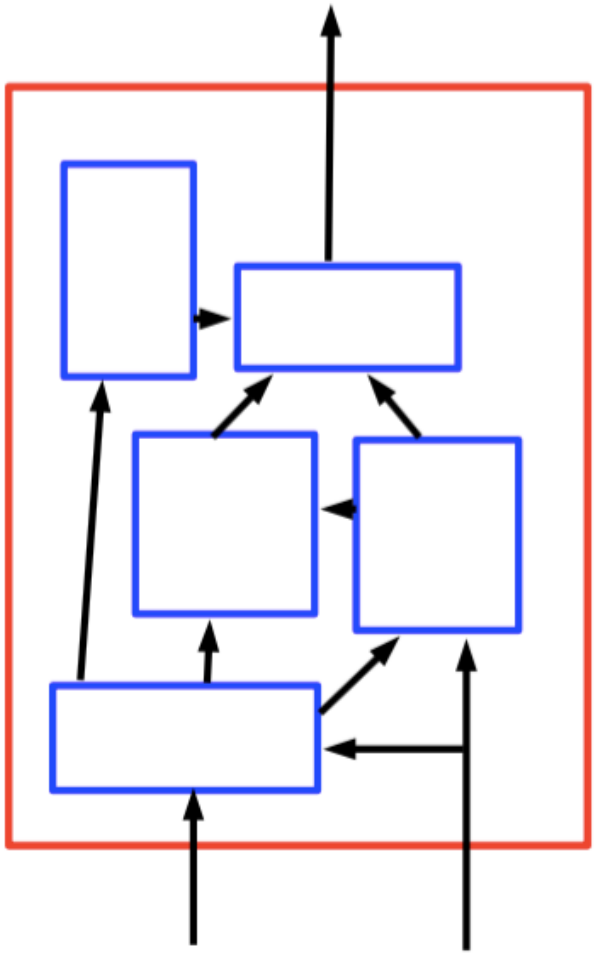
  where $f(\cdot)$ is the activation function

- **Linear activation function:** $f(z) = Az + B$
  - $(F_i)_j = A \cdot ((\boldsymbol{W}_i)_j^T \cdot (\boldsymbol{x}_{i-1})_j) + B$, used in the Forward pass
  - $\dfrac{\partial F_i(\boldsymbol{x}_{i-1}, \boldsymbol{W}_i)}{\partial \boldsymbol{x}_{i-1}} = A\boldsymbol{W}_i$, used in the Backward pass

- **Hyperbolic tangent activation function:** $f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
  - $(F_i)_j = \tanh\left((\boldsymbol{W}_i)_j^T \cdot (\boldsymbol{x}_{i-1})_j\right)$, used in Fw pass
  - $f'(z) = 1 - \tanh^2(z) \rightarrow \frac{\partial F_i(\boldsymbol{x}_{i-1}, \boldsymbol{W}_i)}{\partial \boldsymbol{x}_{i-1}} = 1 - \tanh^2(\boldsymbol{W}_i^T \cdot \boldsymbol{x}_{i-1})$ used in Bw pass

- **Logistic / sigmoid activation function:** $f(z) = \dfrac{1}{1 + e^{-x}}$
  - $(F_i)_j = \frac{1}{1+e^{-(\boldsymbol{W}_i)_j^T \cdot (\boldsymbol{x}_{i-1})_j}}$, used in Fw pass
  - $f'(z) = f(z)(1 - f(z)) \rightarrow \frac{\partial F_i(\boldsymbol{x}_{i-1}, \boldsymbol{W}_i)}{\partial \boldsymbol{x}_{i-1}} = f\left(\boldsymbol{W}_i^T \cdot \boldsymbol{x}_{i-1}\right) \cdot \left(1 - f\left(\boldsymbol{W}_i^T \cdot \boldsymbol{x}_{i-1}\right)\right)$ used in Bw pass

NO ARCHITECTURAL CONSTRAINTS



- **Any connection is permissible**
  - Networks with loops must be "unfolded in time".
- **Any module is permissible**
  - As long as it is continuous and differentiable almost everywhere with respect to the parameters, and with respect to non-terminal inputs.

AVAILABLE TOOLS

- Gradients can still get somewhat tedious to derive by hand, especially for the more complex models that follow
- Fortunately, a lot of this work has already been done for you
- Tools like Theano (http://deeplearning.net/software/theano/), Torch (http://torch.ch/), TensorFlow (http://www.tensorflow.org/) all let you specify the network structure and then automatically compute all gradients (and use GPUs to do so)
- Autograd package for Python (https://github.com/HIPS/autograd) lets you compute the derivative of (almost) any arbitrary function using numpy operations using automatic back-propagation

WHAT'S CHANGED SINCE THE 80S?

- Most of these algorithms were developed in the 80s or 90s
- So why are these just becoming more popular in the last few years?
  - More data
  - Faster computers (GPUs)
  - (Some) better optimization techniques

ISSUES

- **Vanishing gradients:** as we add more and more hidden layers, back-propagation becomes less and less useful in passing information to the lower layers. In effect, as information is passed back, the gradients begin to vanish and become small relative to the weights of the networks.
  - Each gradient assigns "credit" to each neuron i for the (mis)classification of the input sample, however, credit depends (backward) on the average error associated to the neurons that take the output of i, such that going backward the credit has the tendency to vanish
  - If the activation function has a gradient "mostly" null, as in the case of sigmoids, then, again, gradient corrections become very small
- **Overfitting!**
- How many layers/nodes? (which is related to overfitting . . .)
- Non-convexity (only local minimima can be reached), computation time, . . .
- Time complexity of one BP iteration is $O(|\boldsymbol{W}|^3)$, which is he input for one iteration of SGD. No general guarantees on convergence time

IDEAS FOR OVERCOMING ISSUES

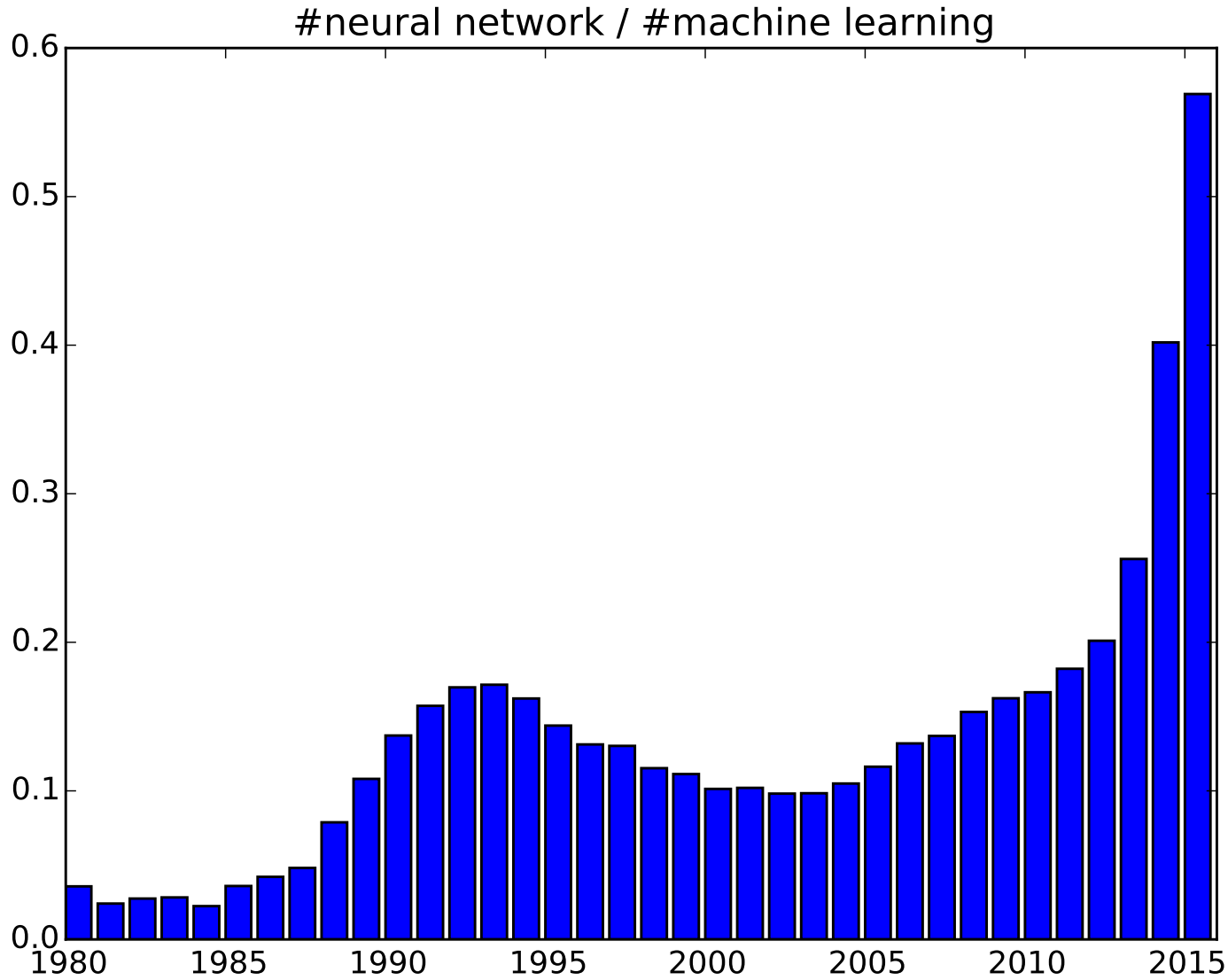
- Hidden layers of autoencoders and RBMs act as effective feature detectors; these structures can be stacked to form deep networks. These networks can be **trained greedily, one layer at a time**, to help to overcome the vanishing gradient and overfitting problems.

- **Unsupervised pre-training (Hinton et al., 2006)**: "Pre-train" the network have the hidden layers recreate their input, one layer at a time, in an unsupervised fashion
  - This paper was partly responsible for re-igniting the interest in deep neural networks, but the general feeling now is that it doesn't help much

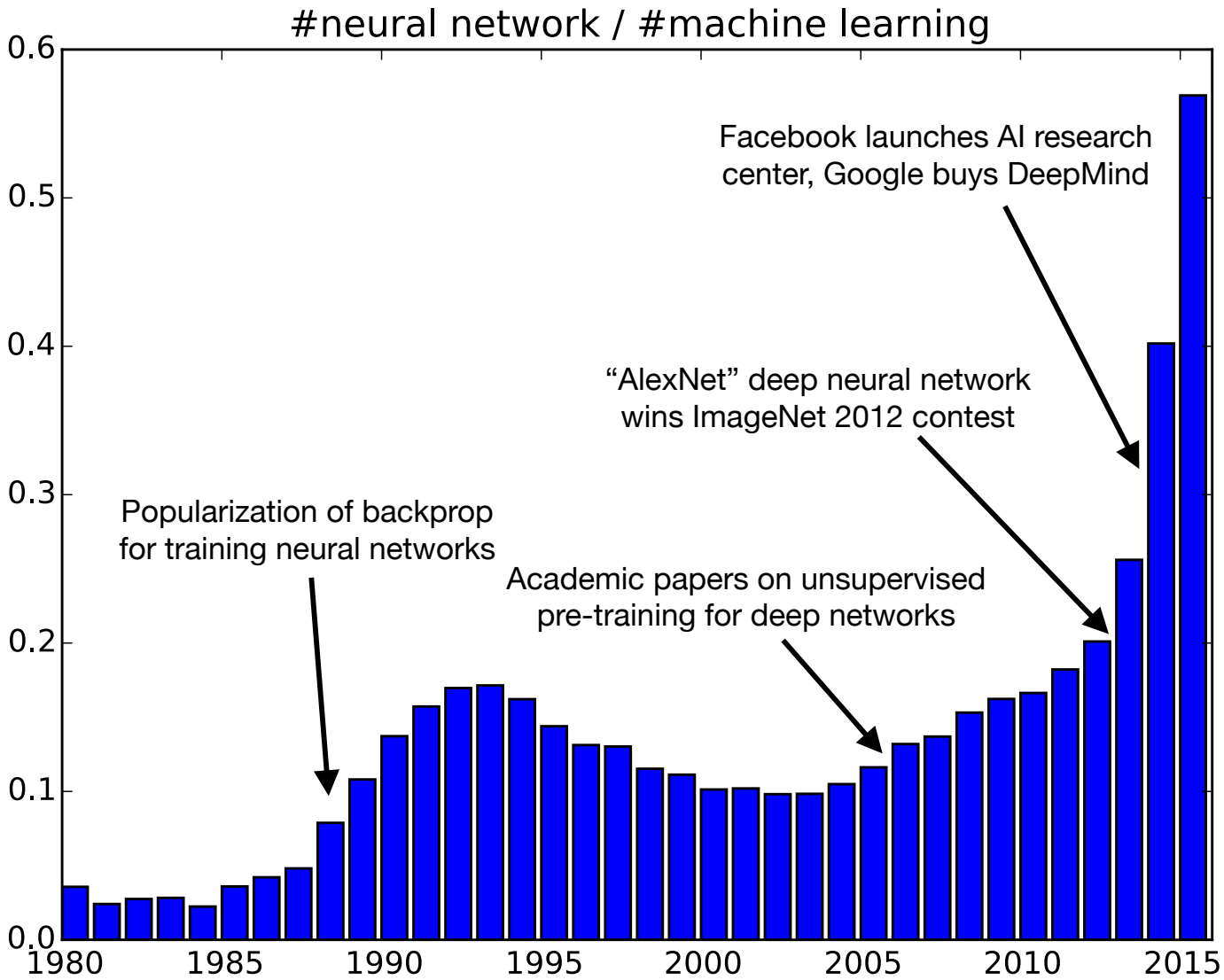
- **Dropout (Hinton et al., 2012)**: During training and computation of gradients, randomly set about half the hidden units to zero (a different randomly selected set for each stochastic gradient step)
  - Acts like regularization, prevents the parameters for overfitting to particular examples

- **Different non-linear functions (Nair and Hinton, 2010)**: Use non-linearity $f(x) = \max\{0, x\}$ instead of $f(x) = \tanh(x)$

OUTLINE

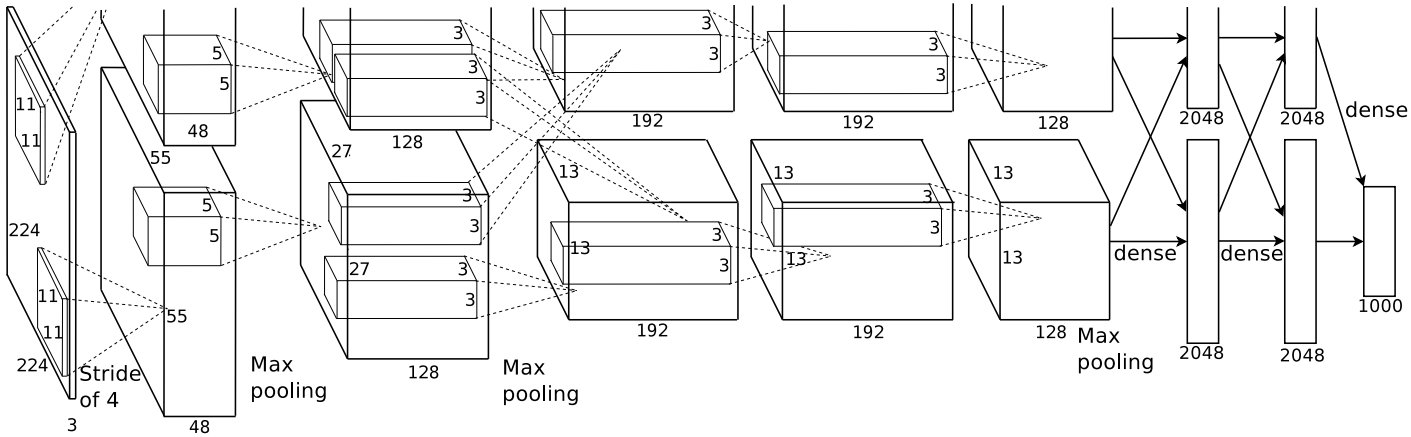


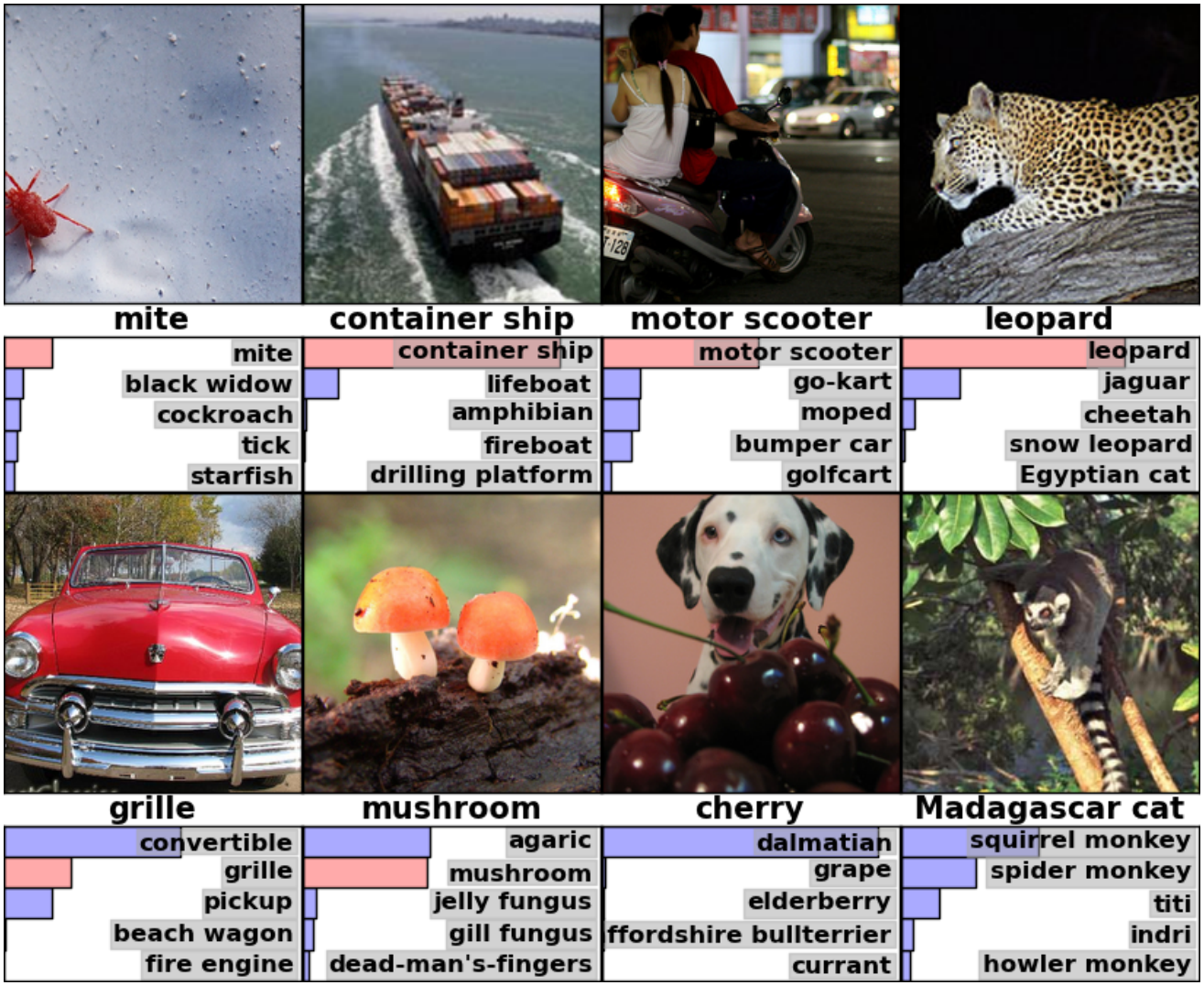
Google scholar counts of papers containing "neural network" divided by count of papers containing "machine learning"



A non-exhaustive list of some of the important events that impacted this trend

- "AlexNet" (Krizhevsky et al., 2012), winning entry of ImageNet 2012 competition with a Top-5 error rate of 15.3% (next best system with highly engineered features based upon SIFT got 26.1% error)

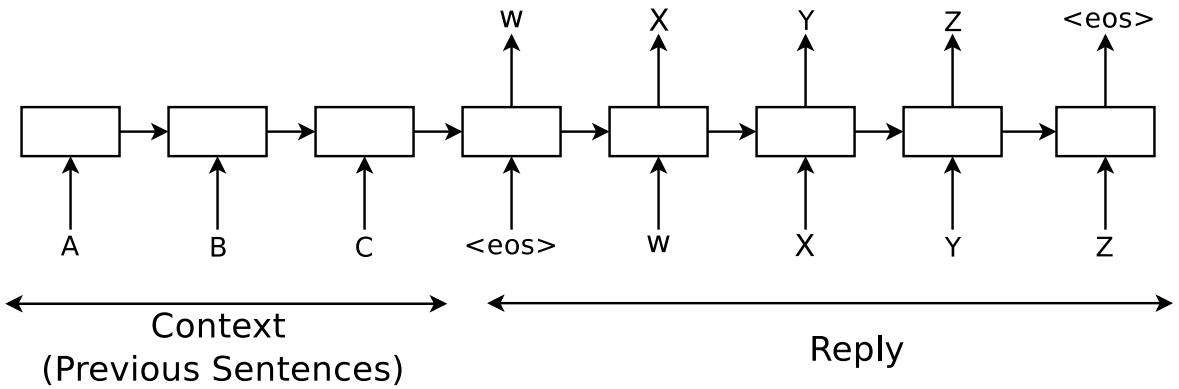




Some classification results from AlexNet

- Google Deep Dream software: adjust input images (by gradient descent) to strengthen the activations that are present in an image

- Question answering network (Vinyals and Le, 2015), using sequence to sequence learning method (Sutskever et al., 2014)

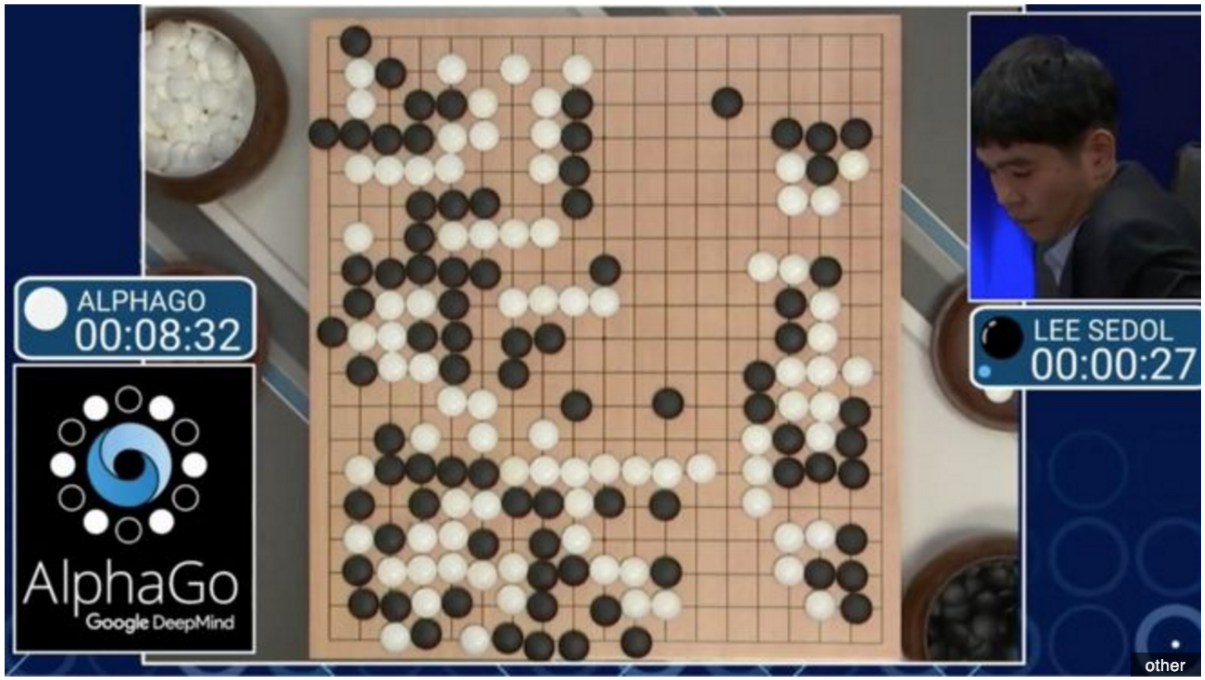


Machine: *hi*
Human: *hello*
Machine: *could you please let me know what are the operating systems you are using ?*
Human: *i am using linux*
Machine: *ok it was in the ssl vpn on which you stated it was broken*
Machine: *please check if you are logged in to corp network*
Machine: *what is the error that you are running please*
Human: *i am seeing an error related to vpn*
Machine: *what is the error message that you are getting when connecting to vpn using network connect ?*
Human: *connection refused or something like that*
Machine: *may i know the version of network connect you connect ?*
Human: *i am not sure i know that*
Machine: *is the network connect prompting for any user certificate*
Machine: *?*
Human: *yes*
Machine: *are you at home or at work at the moment ?*
Machine: *office ?*
Human: *i am at home*
Machine: *try this*
Machine: *goto <URL> page and enter your name and password*
Human: *done*
Machine: *try logging in with <NAME> and check if its working fine now*
Human: *yes , now it works !*
Machine: *great*
Machine: *anything else that i can help*
Human: *no , that was very useful*
Machine: *you are other than <NAME> :)*
Machine: *have a nice night*

- AlphaGo (Silver et al., 2016) beats Lee Sedol in 5 game competition

Artificial intelligence: Google's AlphaGo beats Go master Lee Se-dol

12 March 2016 | Technology



A computer program has beaten a master Go player 3-0 in a best-of-five competition, in what is seen as a landmark moment for artificial intelligence.

Google's AlphaGo program was playing against Lee Se-dol in Seoul, in South Korea.