Machine Learning for Finance (FIN 570)
Neural Network (Deep Learning)

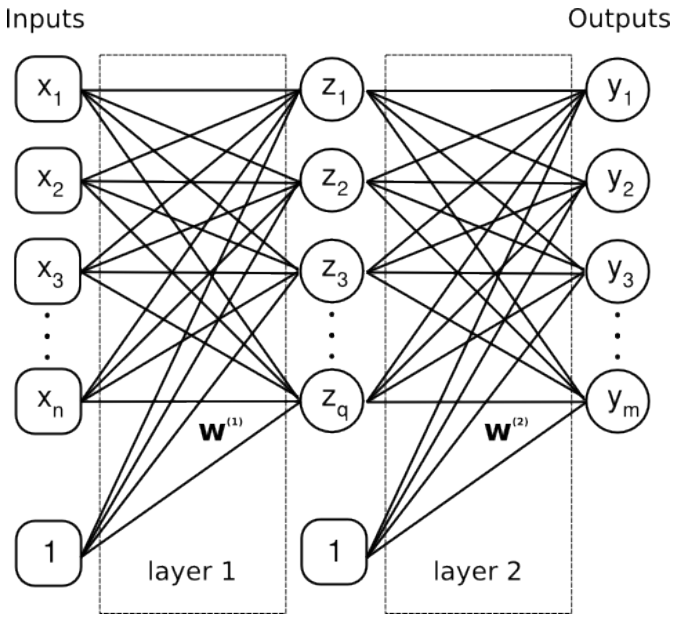
Instructor: Jaehyuk Choi

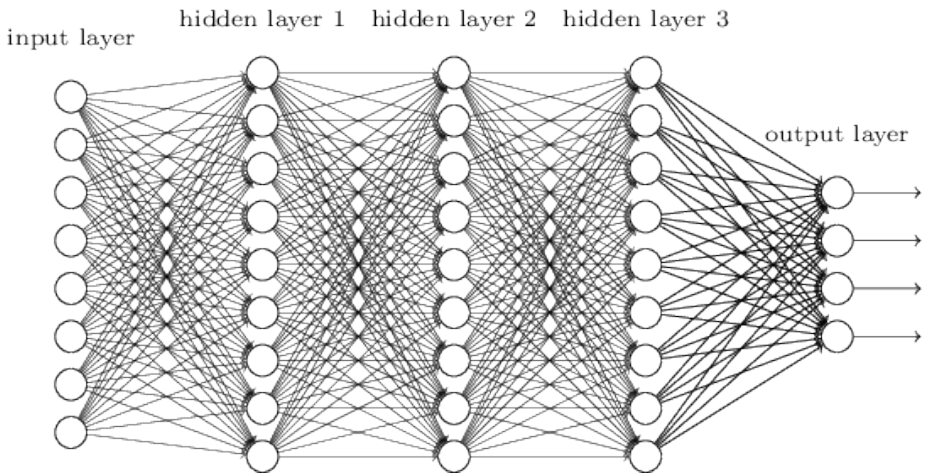
Peking University HSBC Business School, Shenzhen, China

2022-23 Module 3 (Spring 2023)

Neural Network

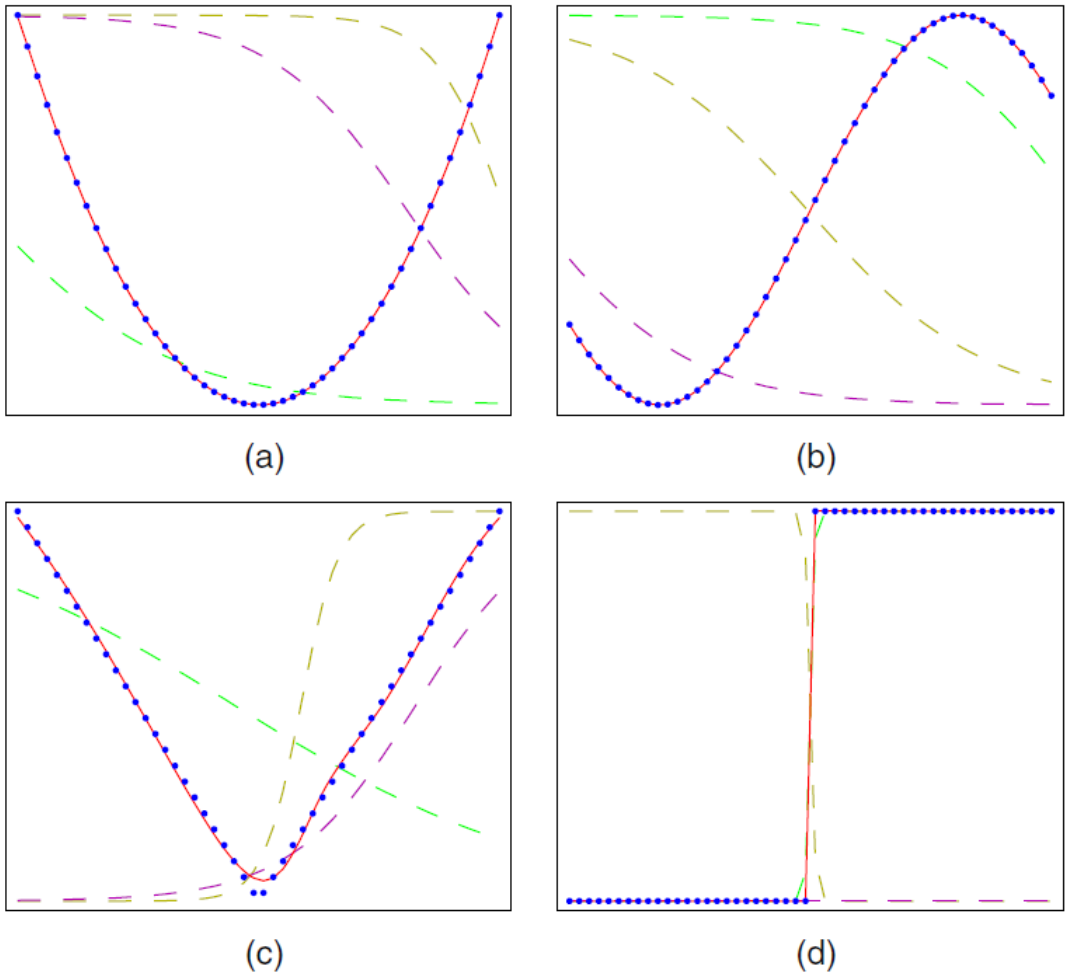
- Multi-layer 'Perceptron' (MLP): logistic regression
- **Deep learning**: a set of methods to efficiently train multi-layer NN
  - backpropagation, activation function, dropout, etc





Neural Network

Figure 5.3 Illustration of the capability of a multilayer perceptron to approximate four different functions comprising (a) $f(x) = x^2$, (b) $f(x) = \sin(x)$, (c), $f(x) = |x|$, and (d) $f(x) = H(x)$ where $H(x)$ is the Heaviside step function. In each case, $N = 50$ data points, shown as blue dots, have been sampled uniformly in x over the interval $(-1, 1)$ and the corresponding values of $f(x)$ evaluated. These data points are then used to train a two-layer network having 3 hidden units with 'tanh' activation functions and linear output units. The resulting network functions are shown by the red curves, and the outputs of the three hidden units are shown by the three dashed curves.



NN Notations

Single layer

$$\boldsymbol{y} = \phi(\boldsymbol{A} = \boldsymbol{x}\boldsymbol{W}) \quad \Rightarrow \quad [\cdots\ y_j\ \cdots] = [\cdots\ x_i\ \cdots] \begin{bmatrix} \cdots \\ \cdots\ W_{ij}\ \cdots \\ \cdots \end{bmatrix} \quad (x_0 = 1)$$

W_{ij} connects i-th input x_i to j-th output y_j. $\phi(x)$ is the sigmoid function. Note that this convention is **different** from **PML**.

Multi-layer: L layers

Input: $\boldsymbol{x} = \boldsymbol{x}^{(0)}$, Output: $\boldsymbol{y} = \boldsymbol{x}^{(L)}$.

$$\boldsymbol{x}^{(1)} = \phi(\boldsymbol{a}^{(1)} = \boldsymbol{x}^{(0)}\boldsymbol{W}^{(1)})$$
$$\boldsymbol{x}^{(2)} = \phi(\boldsymbol{a}^{(2)} = \boldsymbol{x}^{(1)}\boldsymbol{W}^{(2)})$$
$$\cdots$$
$$\boldsymbol{y} = \boldsymbol{x}^{(L)} = \phi(\boldsymbol{a}^{(L)} = \boldsymbol{x}^{(L-1)}\boldsymbol{W}^{(L)})$$

$$\boldsymbol{y} = \phi(\cdots\phi(\phi(\boldsymbol{x}^{(0)}\boldsymbol{W}^{(1)})\boldsymbol{W}^{(2)})\cdots)$$

$$[\cdots\ y_k\ \cdots] = [\cdots\ x_i\ \cdots] \times \begin{bmatrix} \cdots \\ \cdots\ W_{ij}^{(1)}\ \cdots \\ \cdots \end{bmatrix} \cdots \begin{bmatrix} \cdots \\ \cdots \times W_{jk}^{(L)}\ \cdots \\ \cdots \end{bmatrix}$$

Training NN

- Multi-variate response variable (n: samples, j: responses)

$$\boldsymbol{Y} = \begin{bmatrix} \cdots \\ \cdots\ Y_{nj}\ \cdots \\ \cdots \end{bmatrix}, \quad \hat{\boldsymbol{Y}} = \phi(\boldsymbol{A} = \boldsymbol{X}\boldsymbol{W}), \quad \hat{Y}_{nj} = \phi(A_{nj} = \boldsymbol{X}_{n*}\boldsymbol{W}_{*j})$$

- Error (loss) function for the n-th sample:

$$J_n(\boldsymbol{W}) = -\sum_j Y_{nj} \log \phi(A_{nj}) + (1 - Y_{nj}) \log\left(1 - \phi(A_{nj})\right)$$

- Gradient: (i: input, j: output)

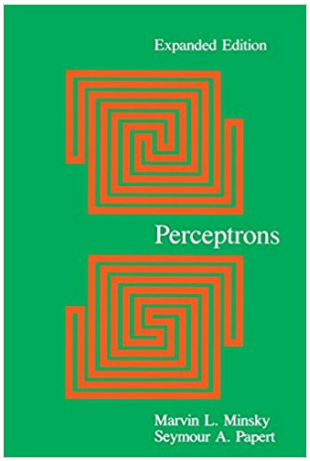
$$\frac{\partial J_n(\boldsymbol{W})}{\partial W_{ij}} = -(Y_{nj} - \phi(A_{nj}))X_{ni} = -(Y_{nj} - \hat{Y}_{nj})X_{ni}$$

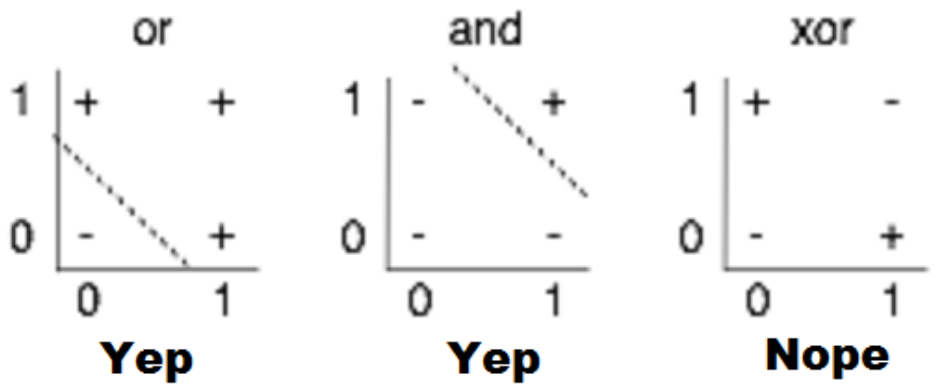
- Total loss function with regularization:

$$J(\boldsymbol{W}) = \sum_n J_n(\boldsymbol{W}) + \lambda_1 \sum_k \sum_{ij} |W_{ij}^{(k)}| + \frac{\lambda_2}{2} \sum_k \sum_{ij} (W_{ij}^{(k)})^2$$

The 1st AI winter (1969∼1986)

- *Perceptrons* by Minsky and Papert (1969) mathematically proved that the single layer can not learn exclusive OR (XOR) gate due to its nonlinear feature. MLP is needed but it's too complicated to train the parameters.
- The criticism from the leading AI figure effectively killed all AI research.
- The difficulty is resolved by backpropagation by Werbos (1974,1982), Hinton(1986).





Backpropagation: chain rule

It's a fancy name of chain rule in calculus. Imagine input x_0 goes through functions f_1, f_2 and f_3 to reach the output $y = f_3(f_2(f_1(x_0)))$:

$$x_0 \to f_1(x_0) = x_1 \to f_2(x_1) = x_2 \to f_3(x_2) = x_3 = y$$

The chain rule is to multiply the derivative of each function:

$$\frac{\partial y}{\partial x_0} = \frac{\partial x_3}{\partial x_2}\frac{\partial x_2}{\partial x_1}\frac{\partial x_1}{\partial x_0} = f_3'(x_2)f_2'(x_1)f_1'(x_0),$$

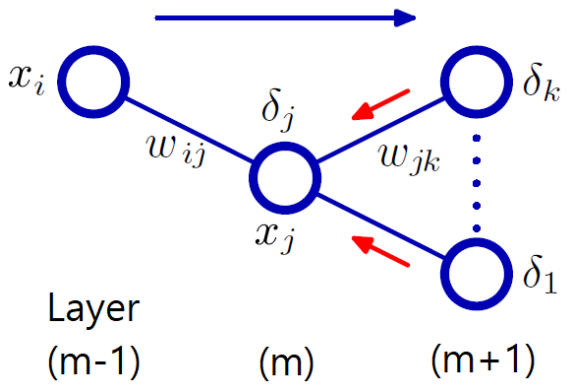
Now imagine the weight w is considered, $y = f_3(f_2(f_1(x_0w_1)w_2)w_3)$

$$x_0 \to f_1(a_1 = x_0w_1) = x_1 \to f_2(a_2 = x_1w_2) = x_2 \to f_3(a_3 = x_2w_3) = x_3 = y$$

$$\frac{\partial y}{\partial w_3} = \frac{\partial}{\partial w_3} f_3(x_2w_3) = f_3'(x_2w_3)x_2 = f_3'(a_3)x_2 = \frac{\partial y}{\partial a_3}x_2$$
$$\frac{\partial y}{\partial w_2} = \frac{\partial}{\partial w_2} f_3(f_2(x_1w_2)w_3) = f_3'(a_3)w_3 \cdot f_2'(a_2)x_1 = \frac{\partial y}{\partial a_2}x_1$$
$$\frac{\partial y}{\partial w_1} = \frac{\partial}{\partial w_1} f_3(f_2(f_1(x_0w_1)w_2)w_3) = f_3'(a_3)w_3 \cdot f_2'(a_2)w_2 \cdot f_1'(a_1)x_0 = \frac{\partial y}{\partial a_1}x_0$$

Backpropagation: error term δ

Imagine W_{ij} connects unit x_i (layer $m-1$) to x_j (layer m) and the unit x_j is connected to the units x_k in the next layer $m+1$. Remind that



$$a_j = \sum_i x_i W_{ij}, \quad x_j = \phi(a_j).$$

$$\frac{\partial J_n}{\partial W_{ij}} = \frac{\partial J_n}{\partial a_j}\frac{\partial a_j}{\partial W_{ij}} = \delta_j x_i, \quad \text{where} \quad \delta_j := \frac{\partial J_n}{\partial a_j}$$

The term δ_j is referred to as error because $\delta_j = \hat{y}_j - y_j$ for the output units.
Now apply chain rule to obtain the backpropagation,

$$\delta_j := \frac{\partial J_n}{\partial a_j} = \sum_k \frac{\partial J_n}{\partial a_k} \cdot \frac{\partial a_k}{\partial x_j} \cdot \frac{\partial x_j}{\partial a_j} = \phi'(a_j) \sum_k W_{jk}\delta_k$$

Back-propagation: $\boldsymbol{\delta}^{(m)} = \phi'(\boldsymbol{a}^{(m)}) * \left(\boldsymbol{\delta}^{(m+1)}(\boldsymbol{W}^{(m+1)})^T\right)$

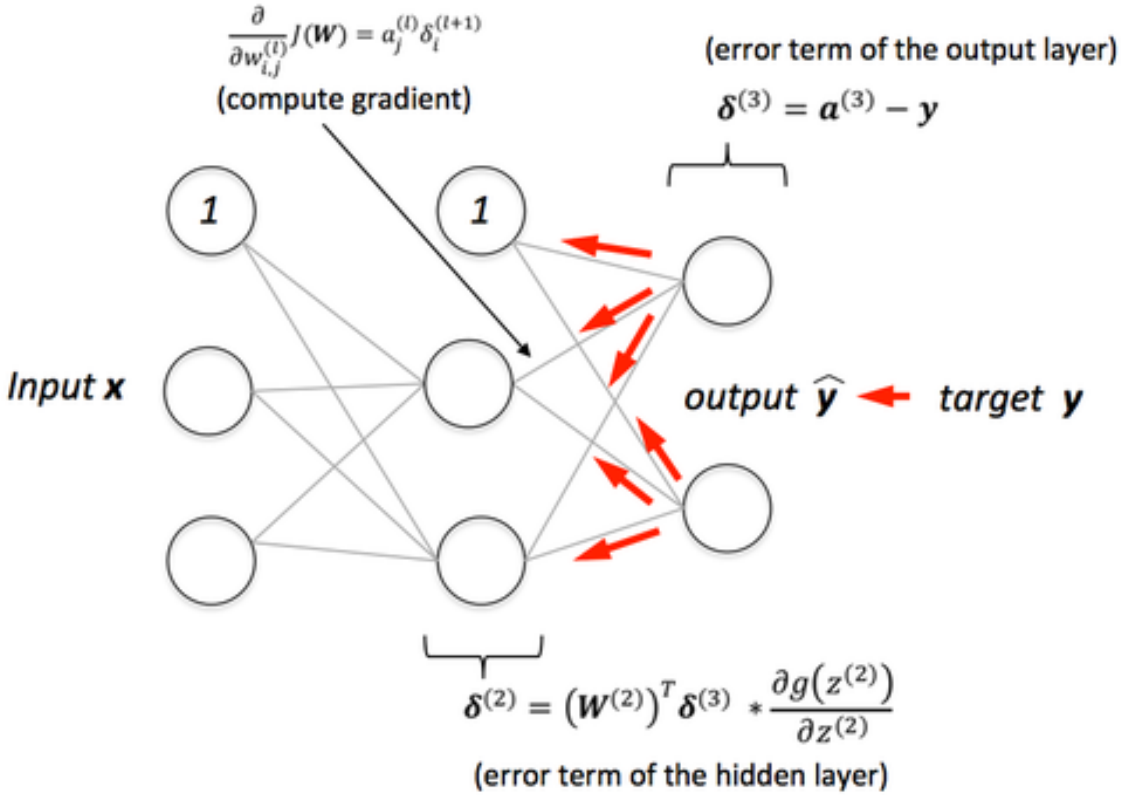
v.s. Forward-propagation: $\boldsymbol{x}^{(m+1)} = \phi(\boldsymbol{a}^{(m)} = \boldsymbol{x}^{(m)}\boldsymbol{W}^{(m+1)})$

where $*$ is the element-wise multiplication.

Backpropagation: $\phi'(a)$

$$x = \phi(a) = \frac{1}{1+e^{-a}} \quad \rightarrow \quad \frac{\partial x}{\partial a} = \phi'(a) = \frac{e^{-a}}{(1+e^{-a})^2} = \phi(a)(1-\phi(a)) = x(1-x)$$

$$x = \phi(a) = \tanh(a) = \frac{e^{a}-e^{-a}}{e^{a}+e^{-a}} \quad \rightarrow \quad \frac{\partial x}{\partial a} = \phi'(a) = (1-\phi^2(a)) = (1-x^2)$$



Backpropagation: procedure and benefit

Procedure

1. Forward-propagate $\boldsymbol{x}$'s and $\boldsymbol{a}$'s: $\boldsymbol{x}^{(m+1)} = \phi(\boldsymbol{a}^{(m+1)} = \boldsymbol{x}^{(m)}\boldsymbol{W}^{(m+1)})$
2. Evaluate $\boldsymbol{\delta}^{(L)} = \hat{\boldsymbol{y}} - \boldsymbol{y}$ at the output units
3. Backpropagate $\boldsymbol{\delta}$'s: $\boldsymbol{\delta}^{(m)} = \phi'(\boldsymbol{a}^{(m)}) * \left(\boldsymbol{\delta}^{(m+1)}(\boldsymbol{W}^{(m+1)})^T\right)$
4. Obtain derivatives: $\dfrac{\partial J_n}{\partial W_{ij}^{(m)}} = \delta_j^{(m)} X_{ni}^{(m-1)}, \quad \dfrac{\partial J}{\partial W_{ij}^{(m)}} = \sum \delta_j^{(m)} X_{ni}^{(m-1)}$

Benefits

Numerical differentiation:

- $\frac{\partial J}{\partial W_{ij}} = \frac{J(W_{ij}+\varepsilon) - J(W_{ij}-\varepsilon)}{2\varepsilon}$
- require $O(N_w^2)$ operations for the total number of weights N_w.
- can be used to validate the backpropagation

Back-propagation:

- require $O(N_w)$ operations.

Activation function: vanishing gradient

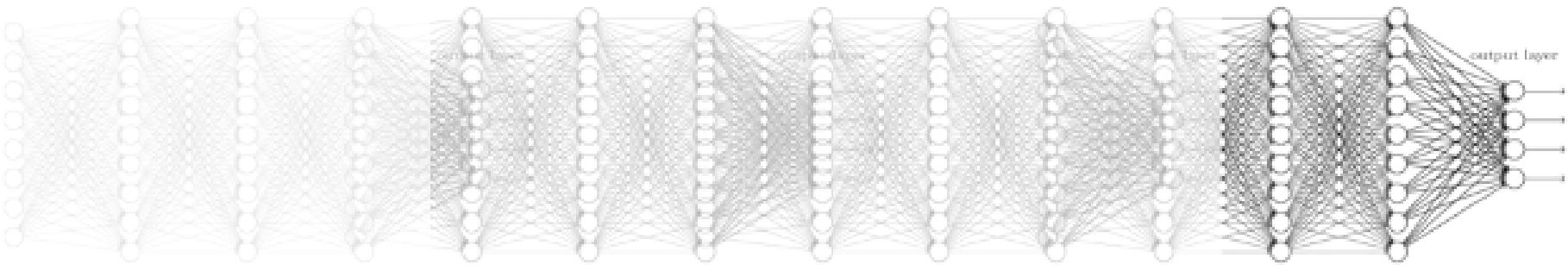
- For sigmoid $\phi(a)$,

$$\phi'(a) = \frac{e^{-a}}{(1+e^{-a})^2} \approx e^{-|a|} \to 0 \quad \text{for large } |a|.$$

- In backpropagation (simple example), the derivative is very small because it is the product of $\phi'(a)$:

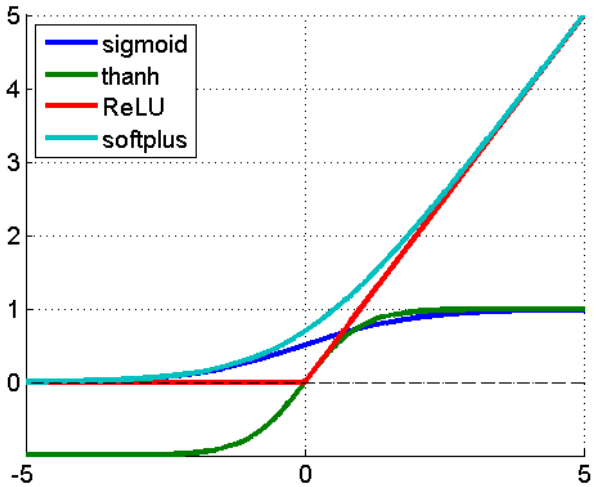
$$\frac{\partial y}{\partial w_1} = \frac{\partial}{\partial w_1}\phi(\phi(\phi(x_0 w_1)w_2)w_3) = \phi'(a_3)w_3 \cdot \phi'(a_2)w_2 \cdot \phi'(a_1)x_0$$

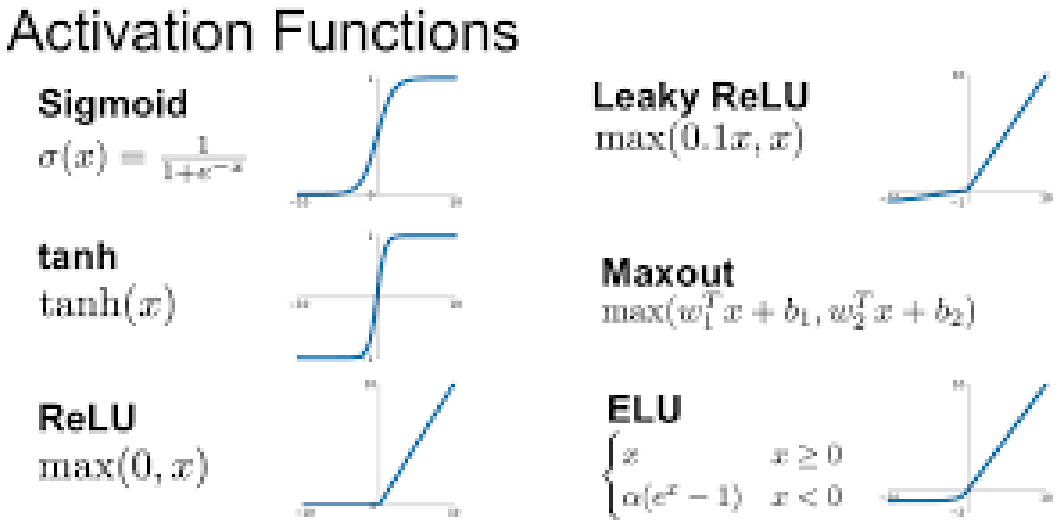
- Even with backpropagation, the training of deep learning models is difficult due to the gradient vanishes. The vanishing gradient caused the 2nd AI winter (1986-2006).



Activation function: ReLU

- Instead of the activation functions with vanishing gradients,
  - **Sigmoid (LR)**: $\frac{1}{1+e^{-x}}$
  - **Tanh**: $\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$,
- Deep learning uses **ReLU** or **Softplus** with non-vanishing gradient:
  - **ReLU (Rectified linear unit)**: $\max(0, x)$
  - **Softplus**: $\log(1 + e^{x})$
- See Wiki for other activation functions.





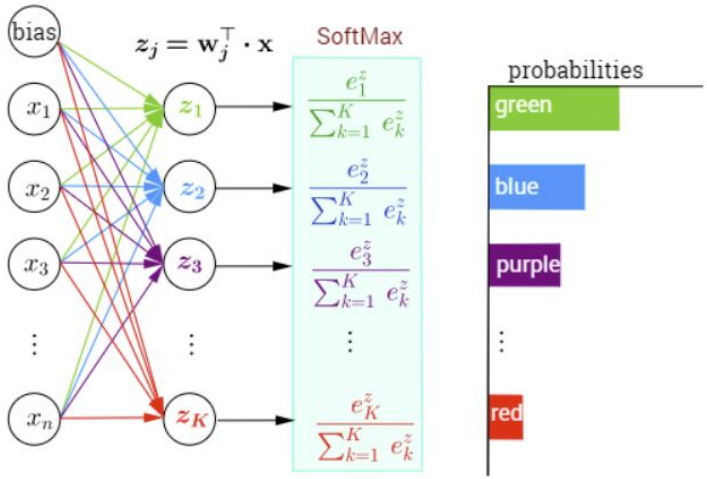
Activation at the output layer: soft-max

- In multi-class classification, we need to *normalize* the output from the last layer to the probability on each class. (We cannot use ReLU in the last layer.)

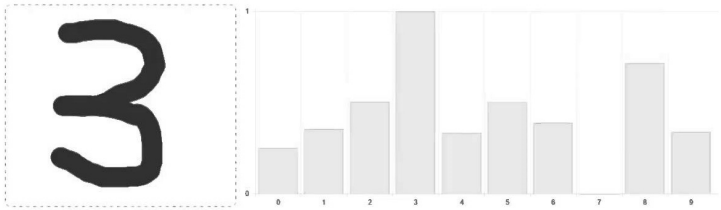
$$0 \leq \boldsymbol{y} = \phi(\boldsymbol{a}^{(L)} = \boldsymbol{x}^{(L-1)}\boldsymbol{W}^{(L)}) \leq 1$$

- Given $\boldsymbol{a}$, we use the softmax function:

$$\phi_{\text{softmax}}(\boldsymbol{a}) = P(y = i|\boldsymbol{a}) = \frac{e^{a_i}}{\sum_k e^{a_k}}$$



- $\sum_i \phi_{\text{softmax}}(a_i) = 1$.
- In prediction, we pick the class i with the largest probability (or largest a_i).
- The logistic function is the special case with two classes and $a_0 = 0$:



$$\phi(a_1) = P(y = 1) = \frac{e^{a_1}}{e^0 + e^{a_1}} = \frac{1}{1 + e^{-a_1}}.$$

Weight initialization: Xavier scheme

- Bad weight initialization choice can lead to a poor convergence of training.
- Big negative/positive wights lead to vanishing/exploding gradient.
- Small weights remove randomness.
- Original Xavier initialization:

$$W_{ij} \sim Z/\sqrt{N_{\text{in}}} \text{ for a standard normal } Z$$

so that

$$\text{Var}(A_j) = \text{Var}\left(\sum_{i=0}^{N_{\text{in}}} W_{ij} X_i\right) \approx \text{Var}(W_{ij}) N_{\text{in}} \approx 1$$

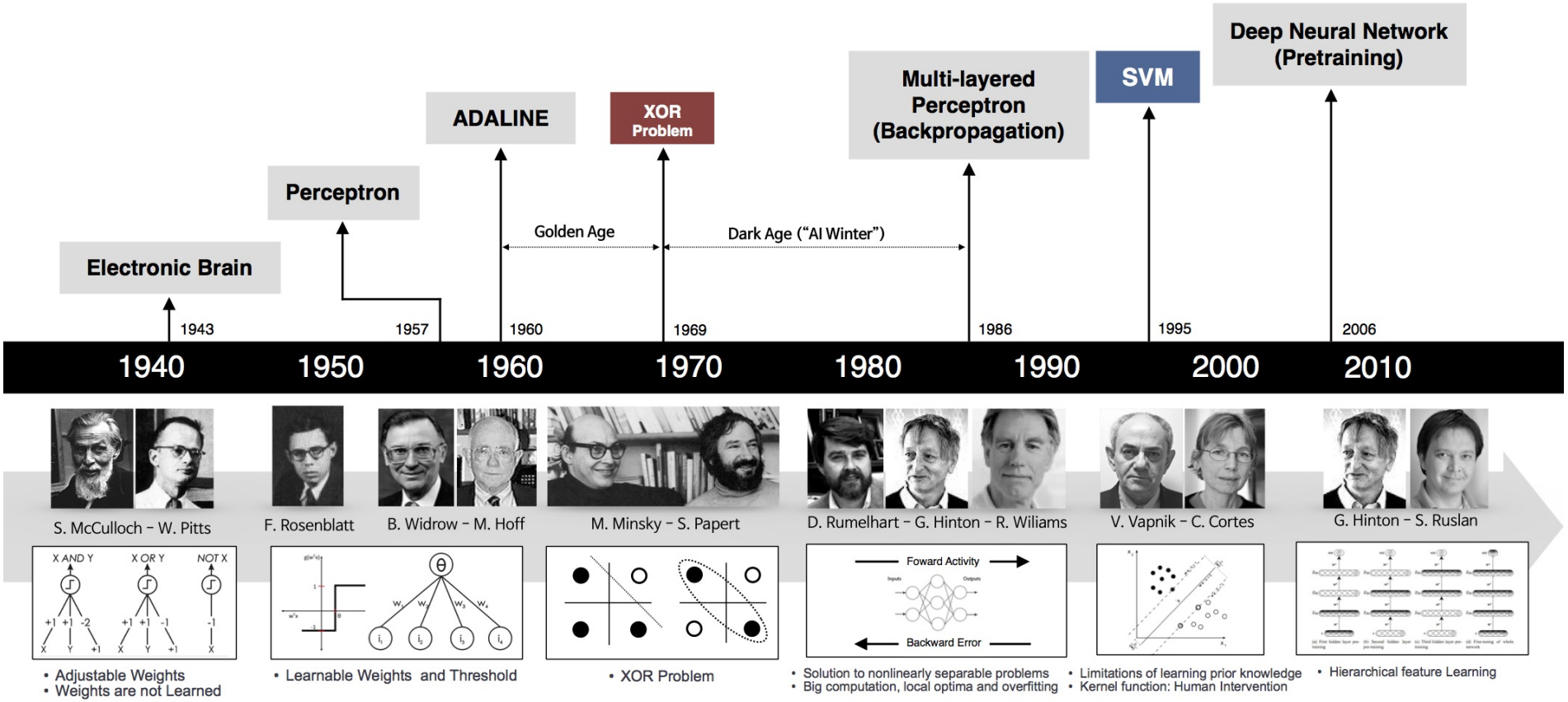
- Glorot and Bengoi (2020) initialization scheme:

$$W_{ij} \sim \sqrt{\frac{6}{N_{\text{in}} + N_{\text{out}}}}\,(2U - 1) \quad \text{or} \quad \sqrt{\frac{2}{N_{\text{in}} + N_{\text{out}}}}\,Z,$$

where U/Z are standard uniform/normal random variable.

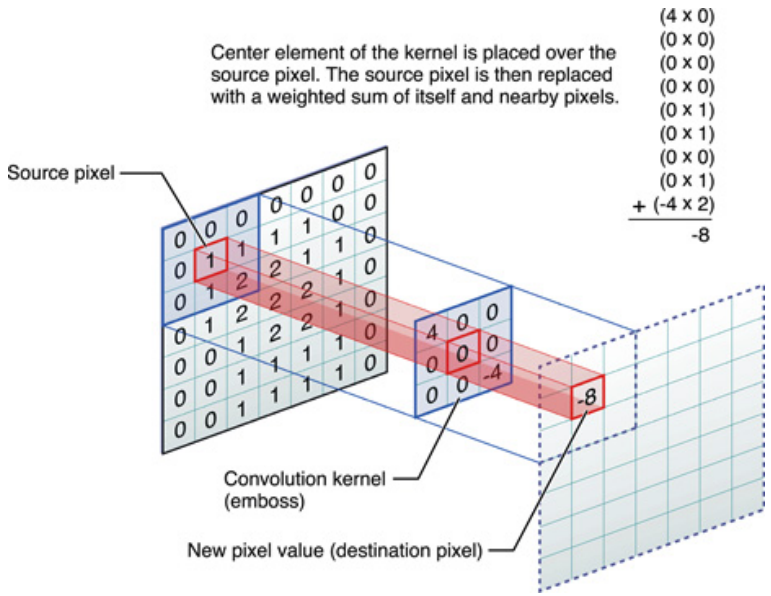
Timeline of NN

From Andrew Beam's Deep Learning 101



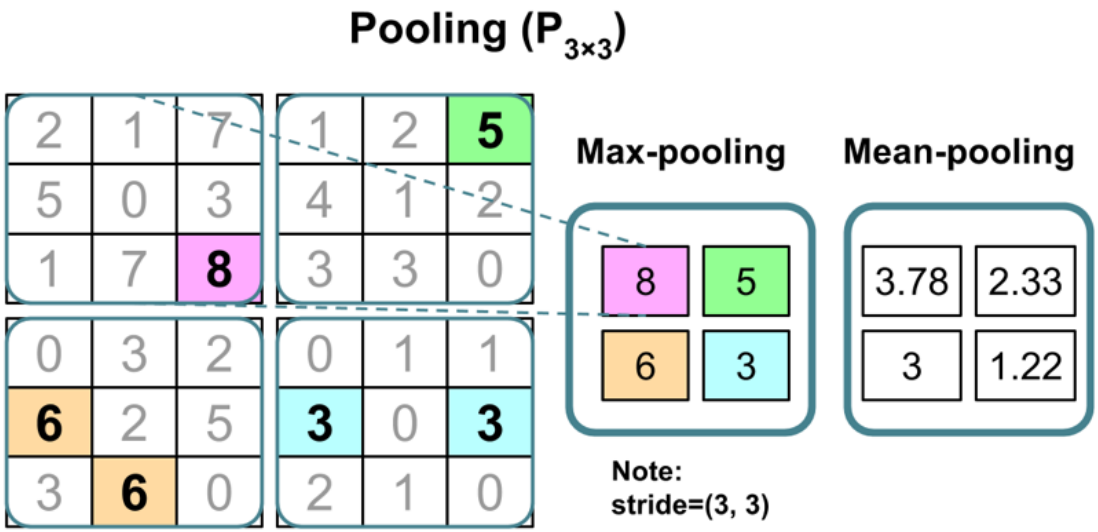
Convolutional Neural Network (CNN): Kernel

- Apply kernel/convolution matrix to image to create feature maps.
- For various image processing kernels, see WikipediA
- The original image is "padded" with zero to avoid boundary loss.
- The layer is locally connected (vs fully connected) with fixed weights.



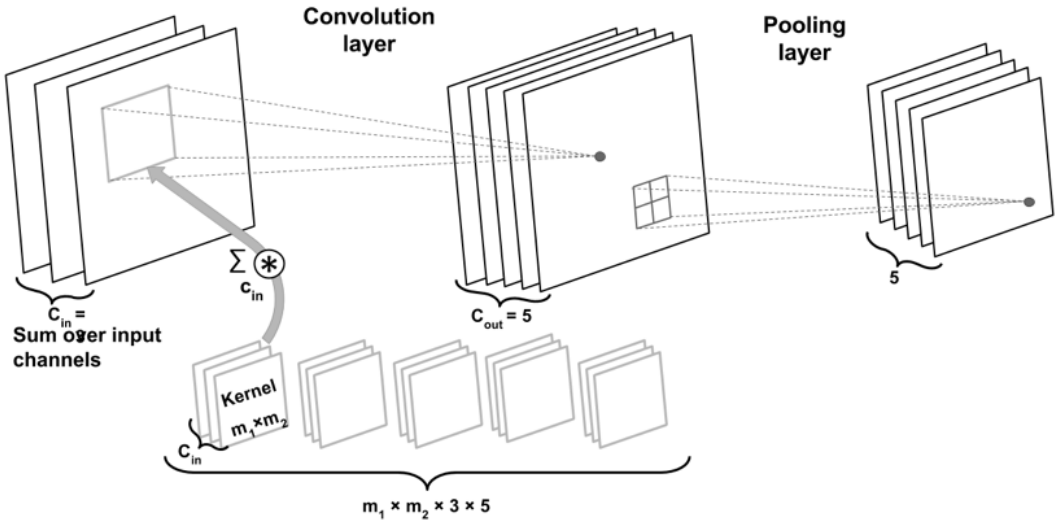
CNN: Pooling/Subsampling

- Either **max** or **mean** applied in non-overlapping way.
- Captures local invariance, robust to noise.
- Reduces feature sizes (e.g. pixel), so avoid overfitting.
- Predetermined operation. No need for learning weights.

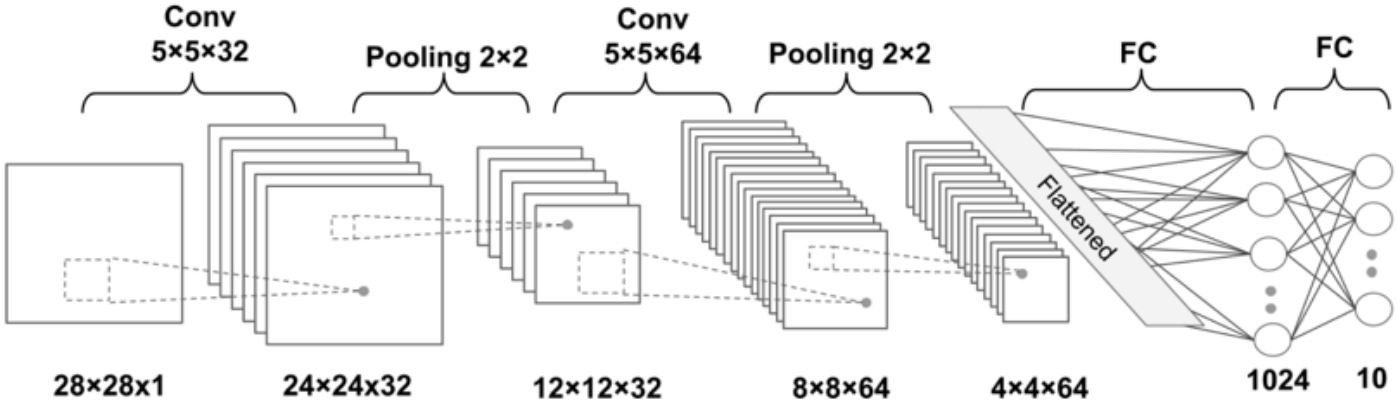


CNN: PML example

- Convolution + pooling unit



- Tensor dimensions



Regularization: Co-adaptation

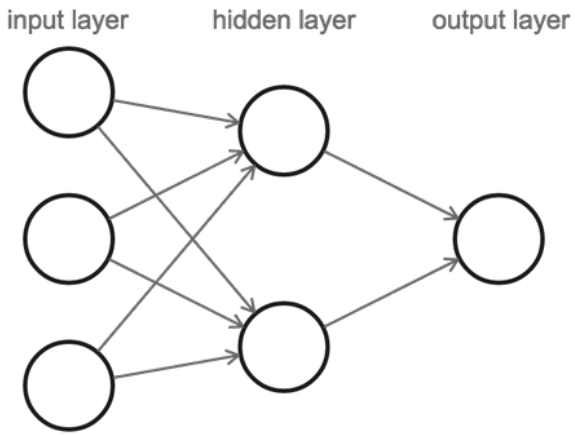
- A typical problem in training deep learning.
- In $3 \times 2 \times 1$ network, suppose

$$W^{(1)} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \\ -2 & 2 \end{bmatrix}, \quad W^{(2)} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

- Assuming linear activation $\phi(x) = x$,

$$\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} W^{(1)} W^{(2)} = 2x_1$$

  x_1 gets too much "attention," and x_2 and x_3 are ignored being canceled each other.)

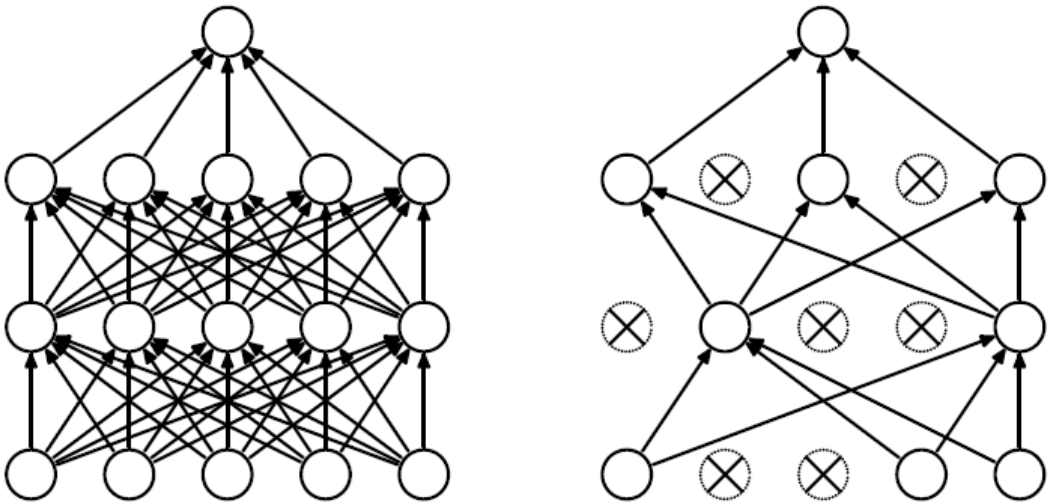


Regularization: Dropout

- Deep neural nets with a large number of parameters are very powerful machine learning systems. However, overfitting is a serious problem in such networks. Large networks are also slow to use, making it difficult to deal with overfitting by combining the predictions of many different large neural nets at test time.
- Dropout is a technique for addressing this problem. The key idea is to randomly drop units (along with their connections) from the neural network during training. This prevents units from co-adapting too much.
- During training, dropout samples from an exponential number of different "thinned" networks. At test time, it is easy to approximate the effect of averaging the predictions of all these thinned networks by simply using a single unthinned network that has smaller weights. This significantly reduces overfitting and gives major improvements over other regularization methods.

...

Regularization: Dropout



(a) Standard Neural Net (b) After applying dropout.

Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

Regularization: Dropout

Training

- If p is the probability of keeping unit ($1-p$: dropout),

$$a = (e_1 x_1 w_1 + \cdots + e_n x_n w_n)/p,$$

  where $e_i = 1$ or 0 with probability p or $1-p$.
- e_i is fixed during one epoch of weight update.
- $p = 1/2$ works best.

Regularization: Test/Prediction

- Use all units without dropout:

$$a = x_1 w_1 + \cdots + x_n w_n,$$

  which is understood as the average over the all possible (2^n) cases.