Neural Networks

AI534

Key concepts:

Neuron and activation functions

Multilayer Perceptron (MLP) neural networks

Universal function approximator

Back-propogation training

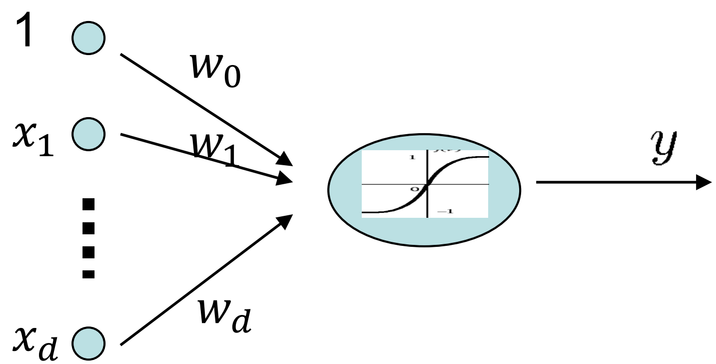
Basics of neural network training

Motivations

- Analogy to biological systems, which are the best examples of robust learning systems
- Consider human brain:
  - Neuron "switching time" ~ 10^{-3} S
  - Scene recognition can be done in 0.1 S
  - There is only time for about a hundred serial steps for performing such tasks
- We need to exploit massive parallelism!

Neural Network Neurons

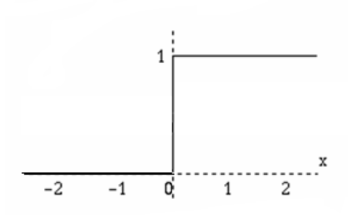
$$y = f(w_0 + w_1 x_1 + \cdots + w_d x_d) = f(\mathbf{w}^T \mathbf{x})$$



- Receives d inputs (plus a bias term)
- Multiplies each input by its weight
- Applies activation function f (typically nonlinear) to the sum of results to generate output

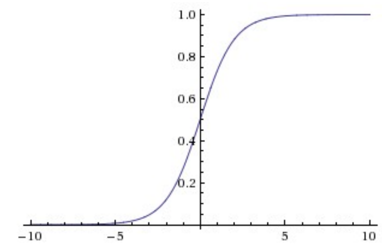
Commonly Used Activation Functions

- **Step function:** $f(x) = \begin{cases} 1 & x > 0 \\ 0 & x \le 0 \end{cases}$



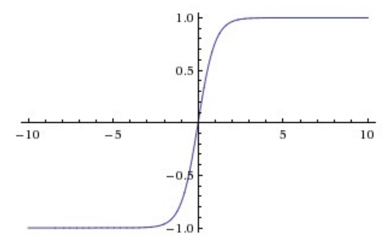
- **Sigmoid function:**

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



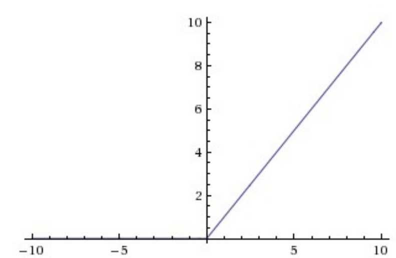
- **Tanh function:**

$$\tanh(x) = 2\sigma(2x) - 1$$

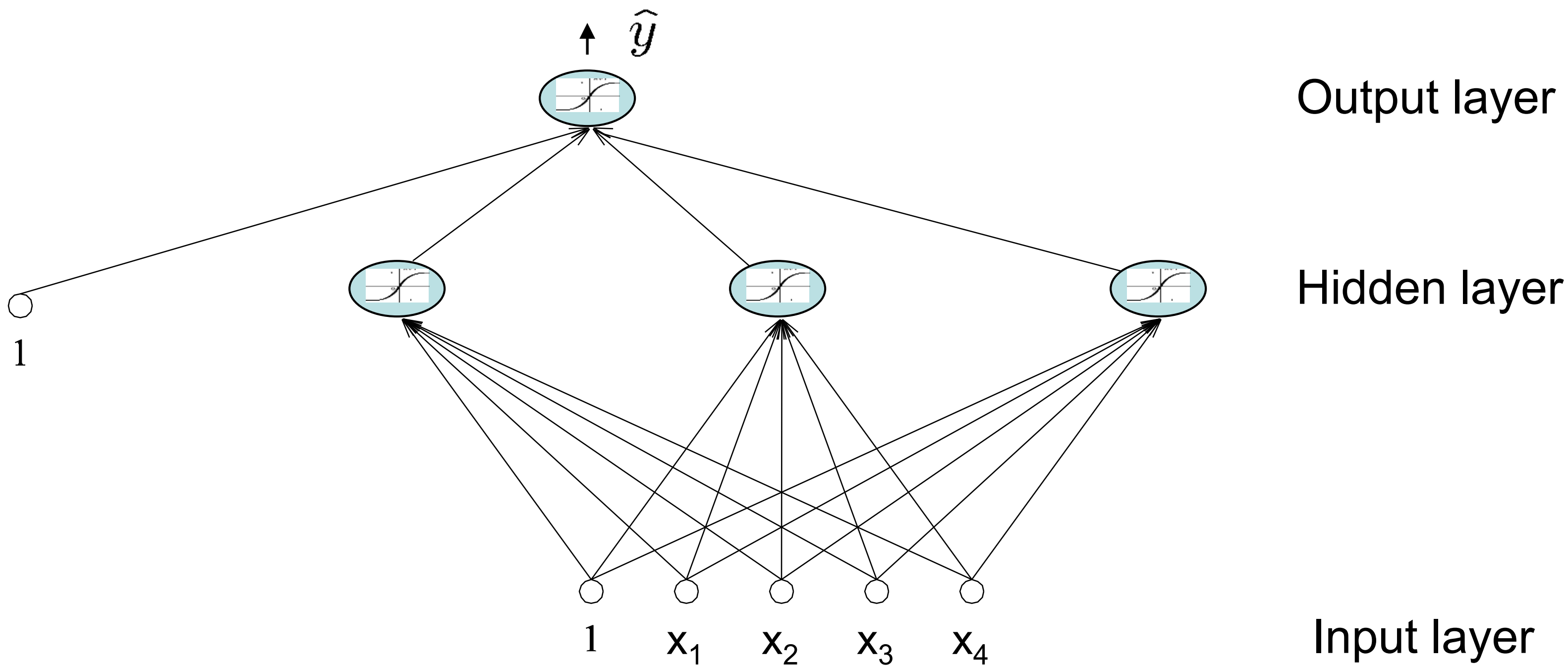


- **Rectified Linear Unit (ReLu)**:

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



Basic Multilayer Neural Network



- Each layer receives its inputs from the previous layer and forwards its outputs to the next – <u>feed forward</u> structure
- Output layer: often use sigmoid activation function for classification, and linear activation function for regression
- Referred to as a two-layer network (2 layer of weights)

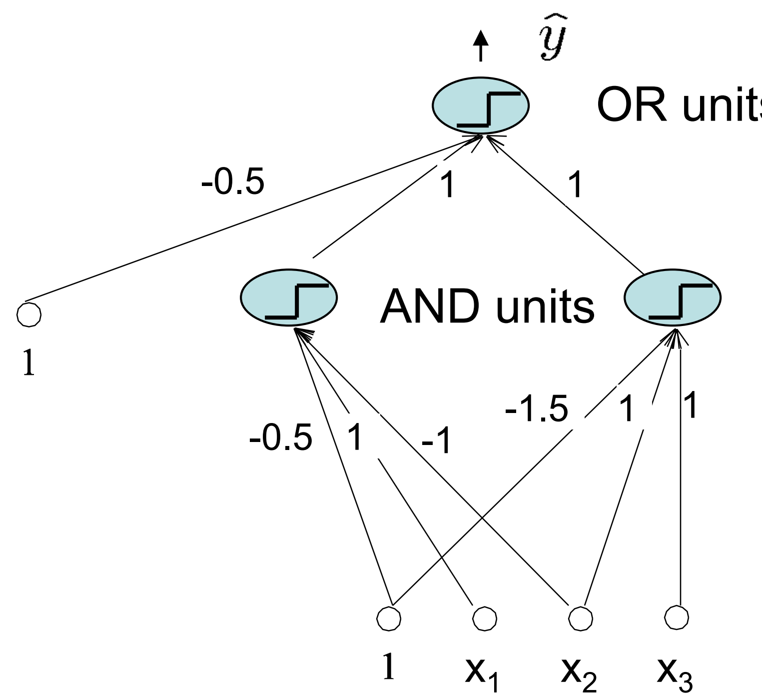
Representational Power

<u>Boolean Formula</u>

- A Boolean function can be transformed into a disjunctive normal form
- Formula in disjunctive normal form can be easily represented using a two layer neural network using step function as activation

For example:

$$(x_1 \wedge \neg x_2) \vee (x_2 \wedge x_3)$$



Representational Power (cont.)

- Continuous functions
  - Any continuous functions can be approximated arbitrarily closely by a sum of (possibly infinite) basis functions
  - Suppose we implement the hidden units to represent the basis functions, and give the output node a linear activation function. Any bounded continuous function can be approximated to arbitrary accuracy with enough hidden units.

Training: Backpropagation

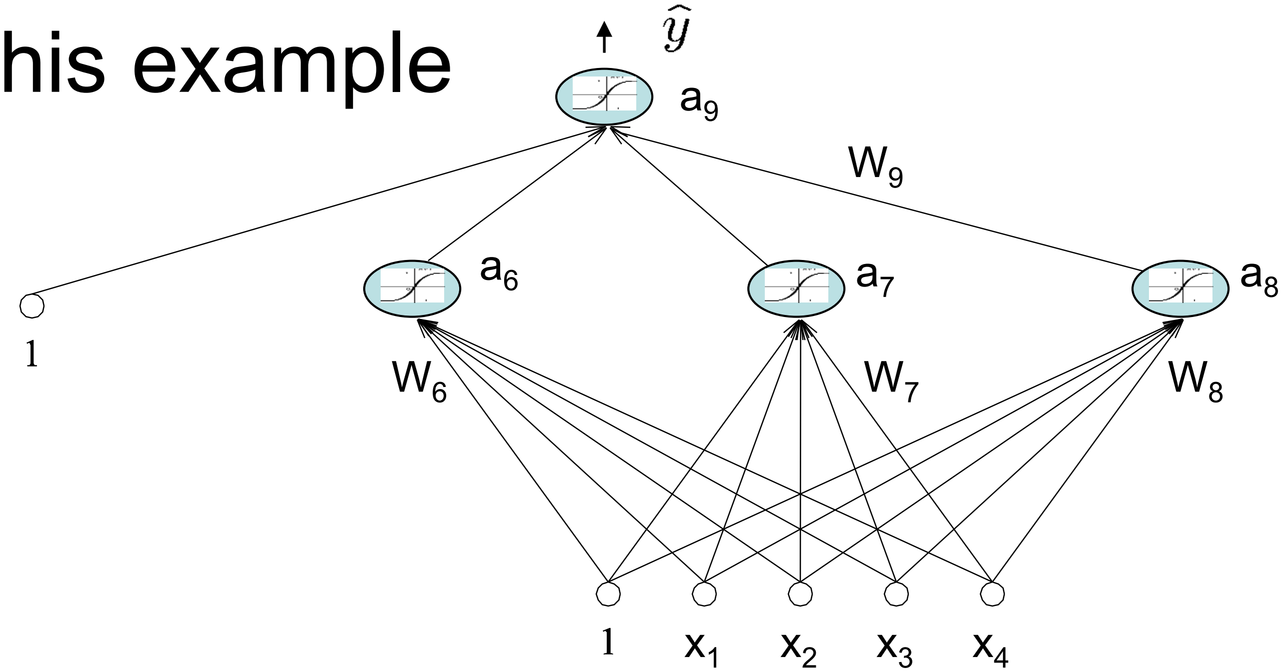
- Training of the neural net aims to find weights that minimize some loss function
- For example, for regression problem, denoting the network output for input $\mathbf{x}$ as $\hat{y}(\mathbf{x})$

$$L(\mathrm{w}) = \sum_{i=1}^{N} (\hat{y}(\mathbf{x}_i, w) - y_i)^2$$

- For classification problems the loss can be different, e.g., **negative log-likelihood** (same as logistic regression)
- Use **gradient descent** to iteratively improve the weights
- This is done from layer to layer, applying the **chain rule** to compute the gradient for each layer

Chain rule for gradient: $\frac{df\big(y(x)\big)}{dx} = \frac{df}{dy}\frac{dy}{dx}$

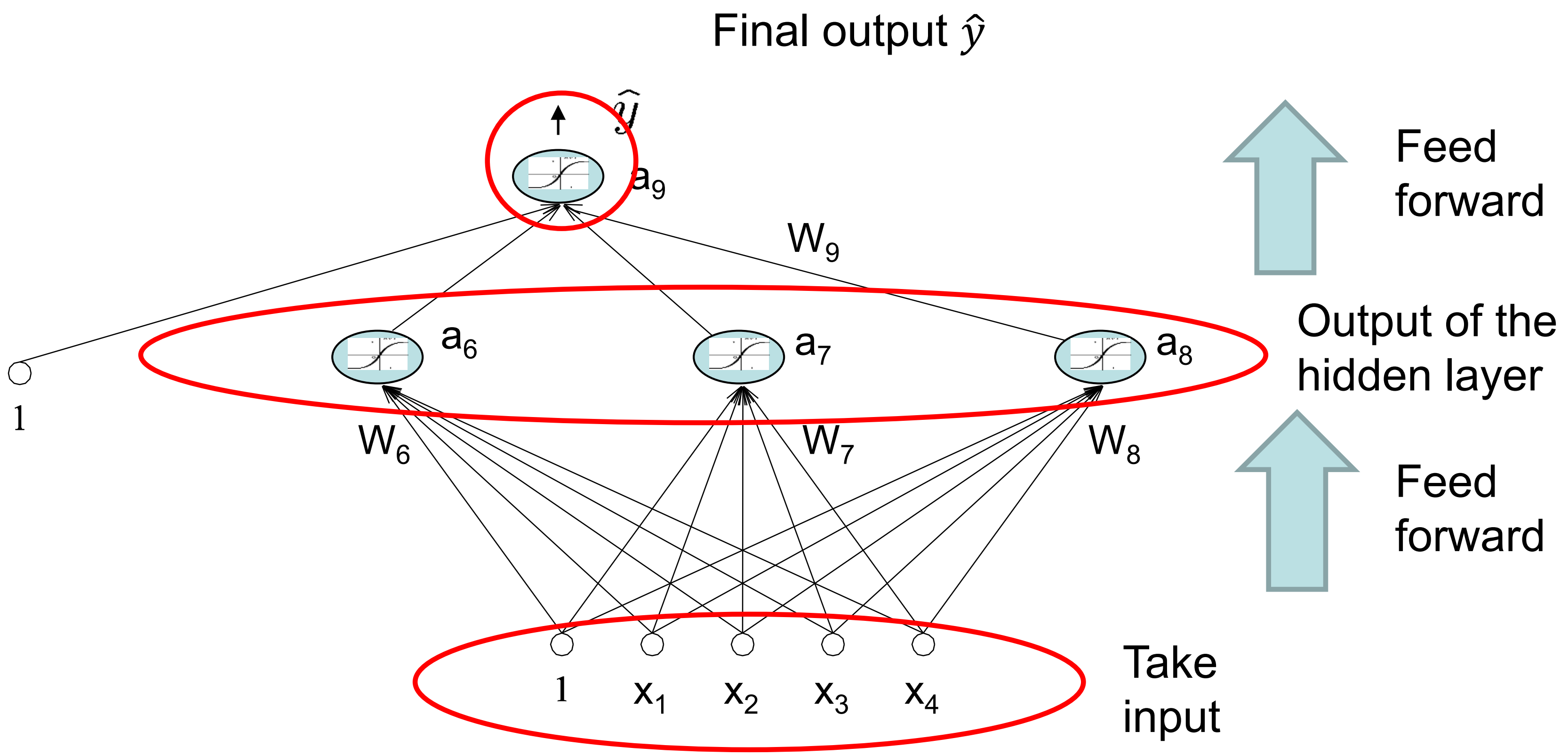
Notation for this example



- $\mathbf{X} = [1, x_1, x_2, x_3, x_4]^T$ – the input vector with the bias term
- $A = [1, a_6, a_7, a_8]^T$ – the output of the hidden layer with the bias term
- $\boldsymbol{W}_i$ represents the weight vector leading to node i
- $w_{i,j}$ represents the weight connecting from the j-th node to the i-th node
  - $w_{9,6}$ is the weight connecting between a_6 and a_9
- We will use σ to represent the activation function, so

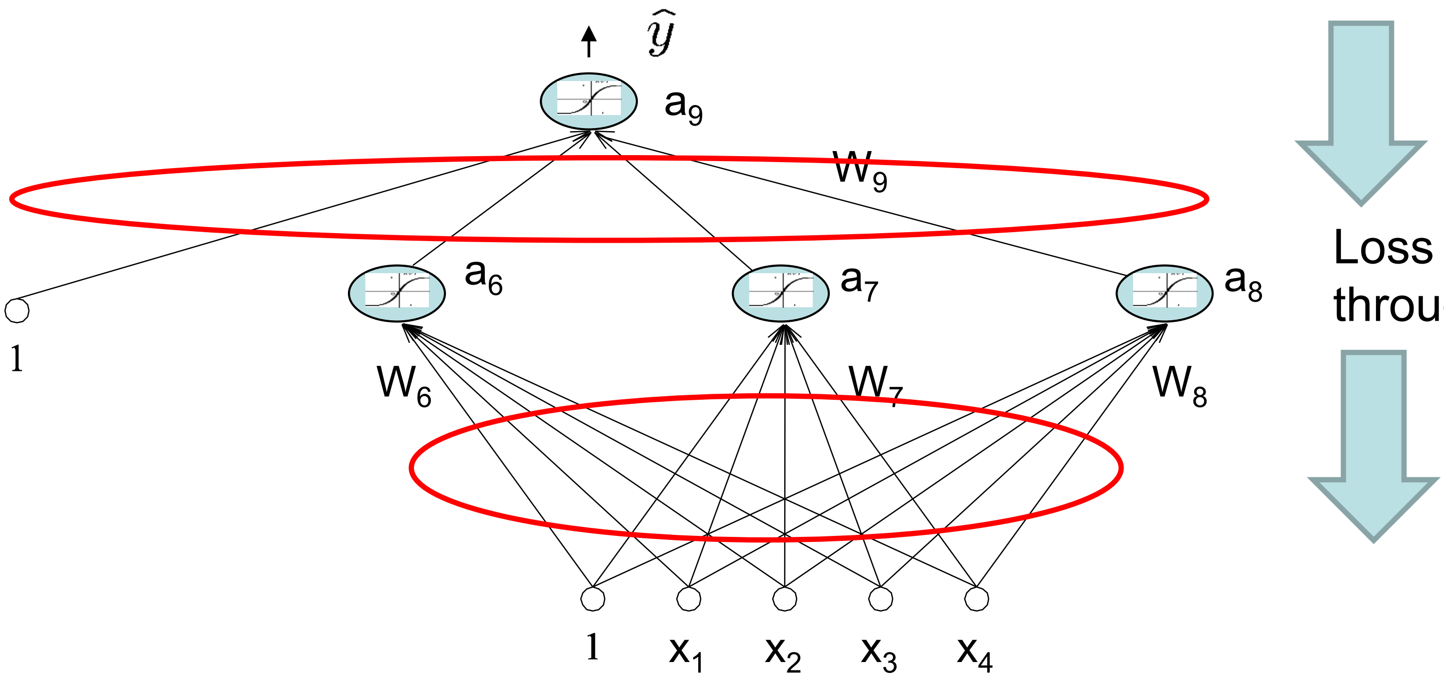
$$\hat{y} = \sigma(W_9 \cdot [1, a_6, a_7, a_8]^T) = \sigma(W_9 \cdot [1, \sigma(W_6 \cdot X), \sigma(W_7 \cdot X), \sigma(W_8 \cdot X)]^T)$$

Training: the forward pass



Training: the backward pass

Compute Loss $L(\hat{y}(x), y)$



Feed gradient back, update weights

Loss propagated through hidden nodes

Feed gradient back, update weights

The calculation of the gradient will depend on the loss function and the activation function – but often it is not complicated

E.g., if we use the same loss as logistic regression, we have the same update rule for updating the outer most weight layer

Example: Sum Squared Error

- We adjust the weights of the neural network to minimize the Sum squared error (SSE) on training set.

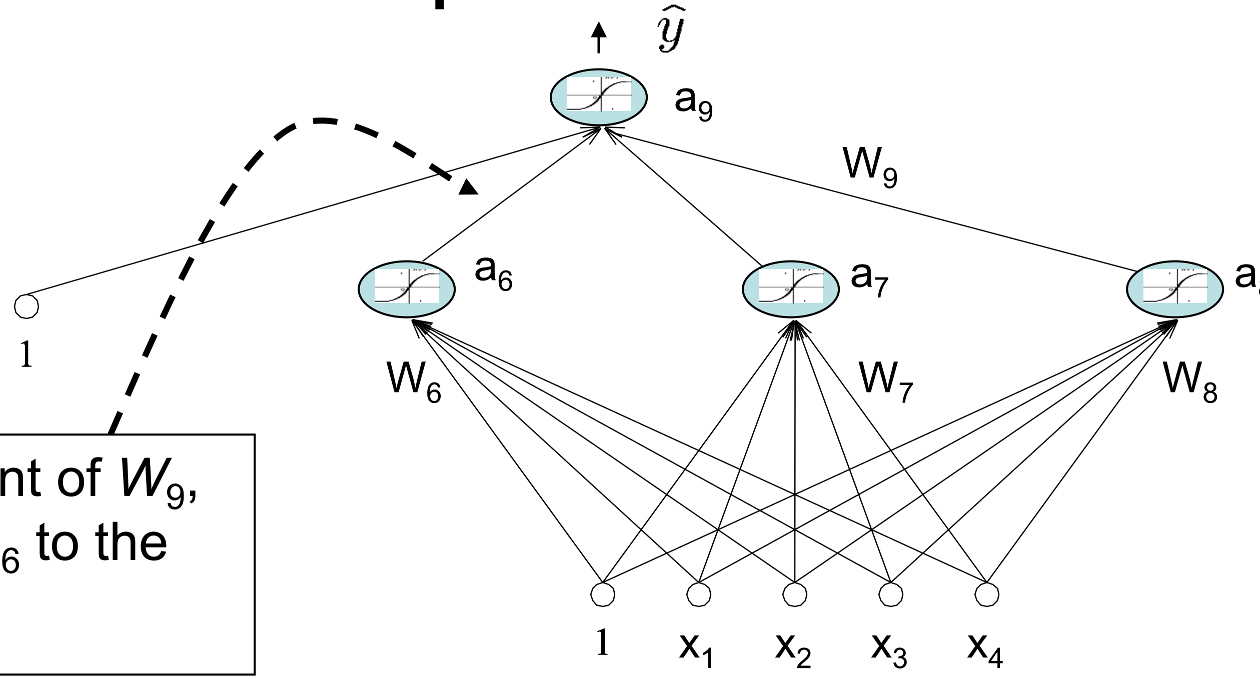
$$J(W) = \frac{1}{2}\sum_{i=1}^{N}(\widehat{y}^i - y^i)^2$$

$$J_i(W) = \frac{1}{2}(\widehat{y}^i - y^i)^2$$

- **<u>Useful fact</u>:** the derivative of the sigmoid activation function is

$$\frac{d\sigma(x)}{dx} = \sigma(x)\big(1 - \sigma(x)\big)$$

Gradient Descent: Output Unit

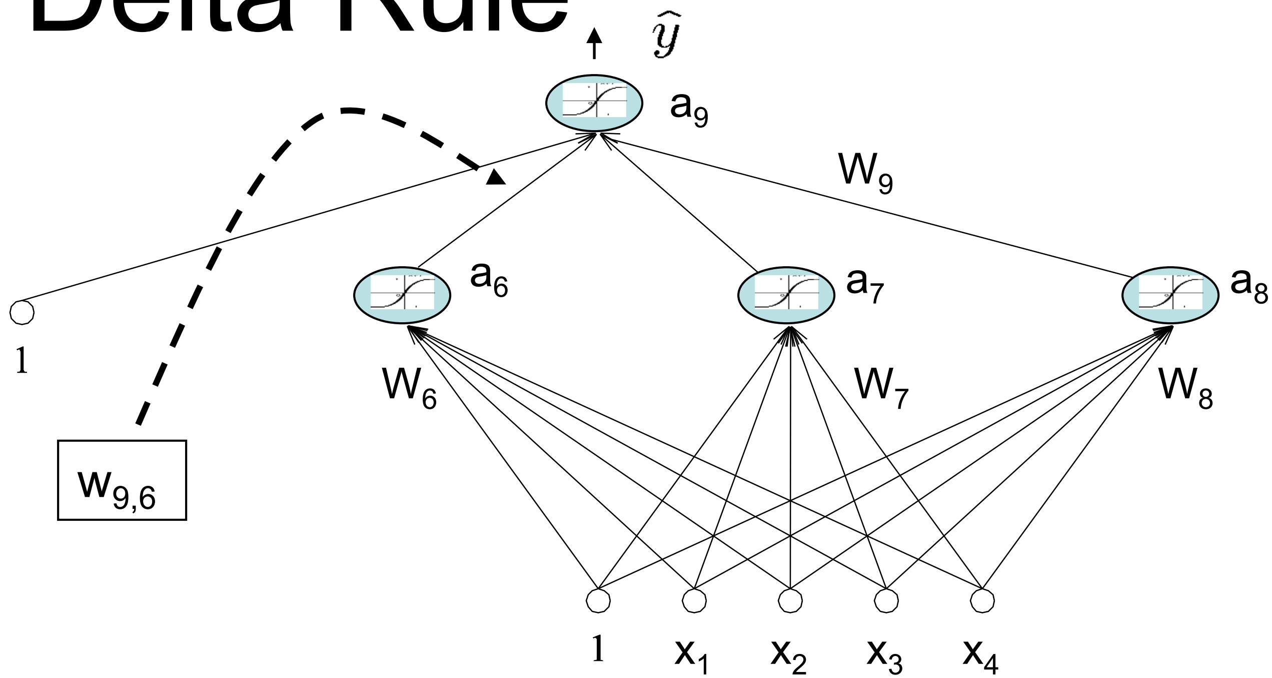


$w_{9,6}$ is a component of W_9, connecting from a_6 to the output node.

$$\frac{d\sigma(x)}{dx} = \sigma(x)\big(1 - \sigma(x)\big)$$

$$\begin{aligned}
\frac{\partial J_i(W)}{\partial w_{9,6}} &= \frac{\partial}{\partial w_{9,6}} \frac{1}{2}(\widehat{y}^i - y^i)^2 \\
&= \frac{1}{2} \cdot 2 \cdot (\widehat{y}^i - y^i) \cdot \frac{\partial}{\partial w_{9,6}} \sigma(W_9 \cdot A^i) \\
&= (\widehat{y}^i - y^i) \cdot \sigma(W_9 \cdot A^i)(1 - \sigma(W_9 \cdot A^i)) \cdot \frac{\partial}{\partial w_{9,6}} W_9 \cdot A^i \\
&= (\widehat{y}^i - y^i)\widehat{y}^i(1 - \widehat{y}^i) \cdot a_6^i
\end{aligned}$$

The Delta Rule



- Define $\delta_9^i = (\widehat{y}^i - y^i)\widehat{y}^i(1 - \widehat{y}^i)$

  then
$$\frac{\partial J_i(W)}{\partial w_{9,6}} = (\widehat{y}^i - y^i)\widehat{y}^i(1 - \widehat{y}^i) \cdot a_6^i$$
$$= \delta_9^i \cdot a_6^i$$

Extending to the whole vector W_9: $\dfrac{\partial J_i}{\partial W_9} = \delta_9^i \ A^i$

Di-secting the delta rule

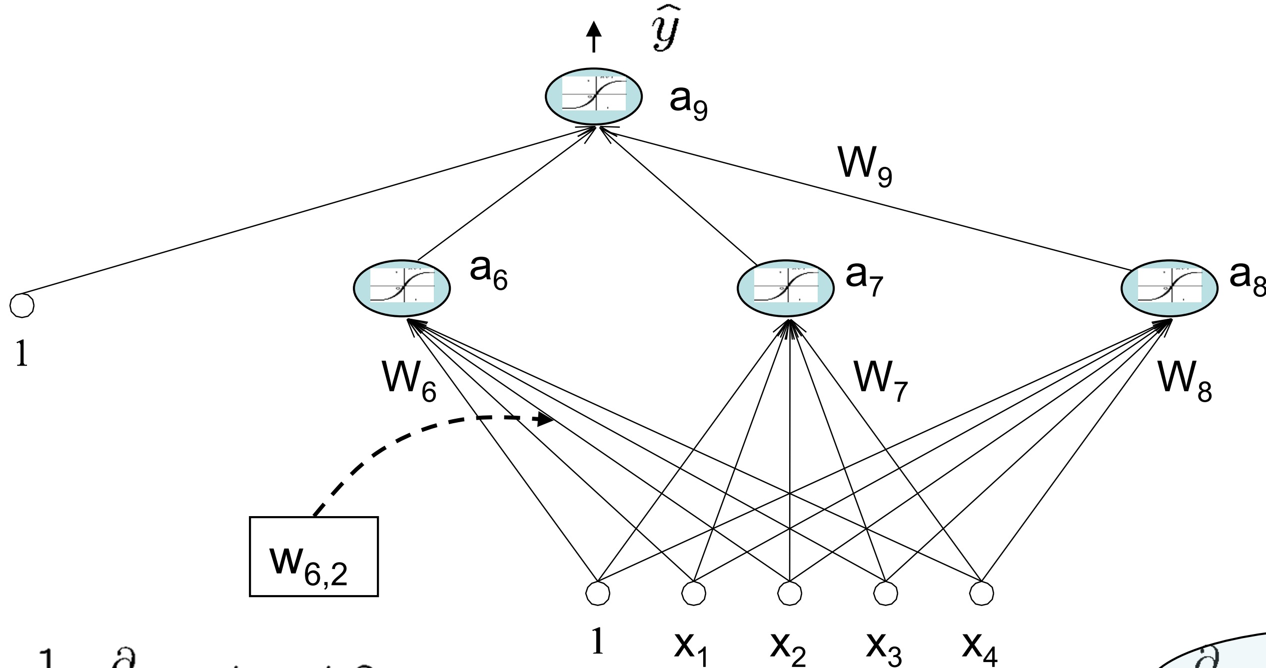
- Consider a general loss function defined on $\hat{y}^i$:

$$L(\hat{y}^i)$$

Where $\hat{y}^i = f\left(\mathbf{W}_9^T \mathrm{A}^i\right)$, f is the activation function

$$\frac{dL}{d\mathrm{W}_\mathbf{9}} = \frac{dL(\hat{y}^i)}{d\hat{y}^i} \times \frac{d\hat{y}^i}{d\mathrm{W}_\mathbf{9}} = (\underbrace{L' \cdot f'}_{\delta_9^i})\mathrm{A}^i$$

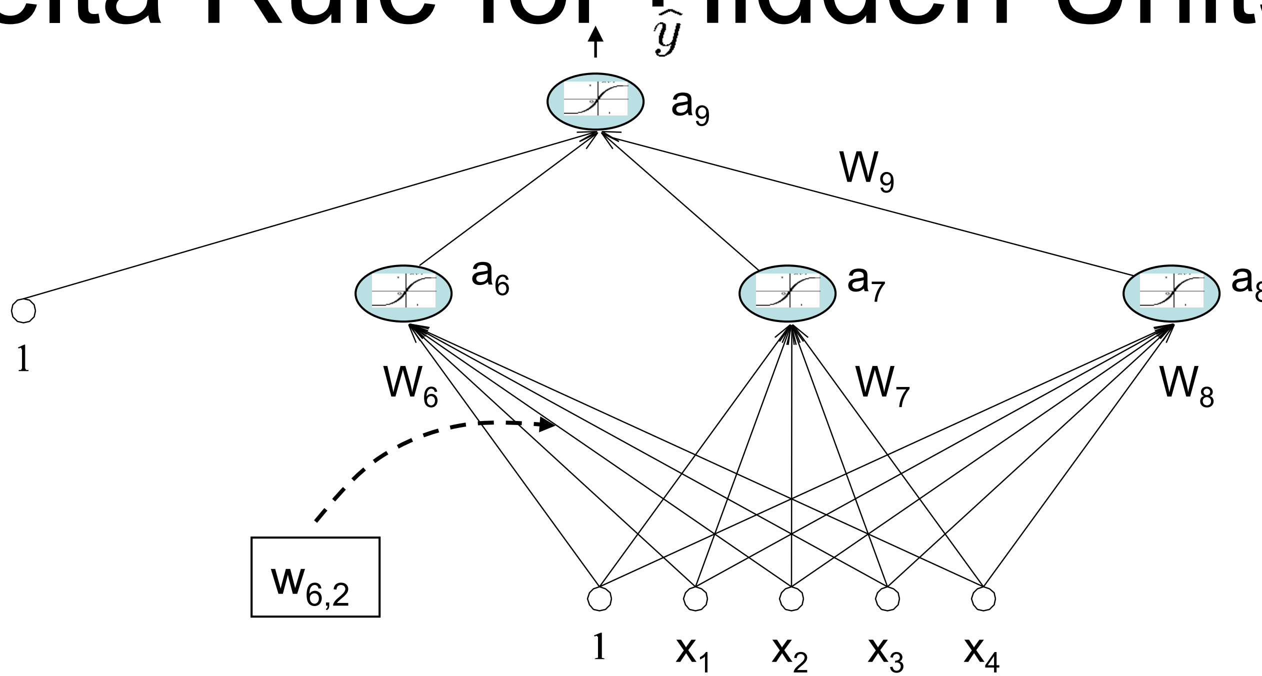
Derivation: Hidden Units



$$\begin{aligned}
\frac{\partial J_i(W)}{\partial w_{6,2}} &= \frac{1}{2}\frac{\partial}{\partial w_{6,2}}(\widehat{y}^i - y^i)^2 \\
&= (\widehat{y}^i - y^i)\cdot\sigma(W_9\cdot A^i)(1-\sigma(W_9\cdot A^i))\cdot\frac{\partial}{\partial w_{6,2}}(W_9\cdot A^i) \\
&= \delta_9^i\cdot w_{9,6}\cdot\frac{\partial}{\partial w_{6,2}}\sigma(W_6\cdot X^i) \\
&= \delta_9^i\cdot w_{9,6}\cdot\sigma(W_6\cdot X^i)(1-\sigma(W_6\cdot X^i))\cdot\frac{\partial}{\partial w_{6,2}}(W_6\cdot X^i) \\
&= \delta_9^i\cdot w_{9,6}\cdot a_6(1-a_6)\cdot x_2^i
\end{aligned}$$

$$\frac{\partial}{\partial w_{6,2}}(W_9\cdot A^i) = \frac{\partial}{\partial w_{6,2}}(w_{9,6}\cdot a_6^i)$$

Delta Rule for Hidden Units



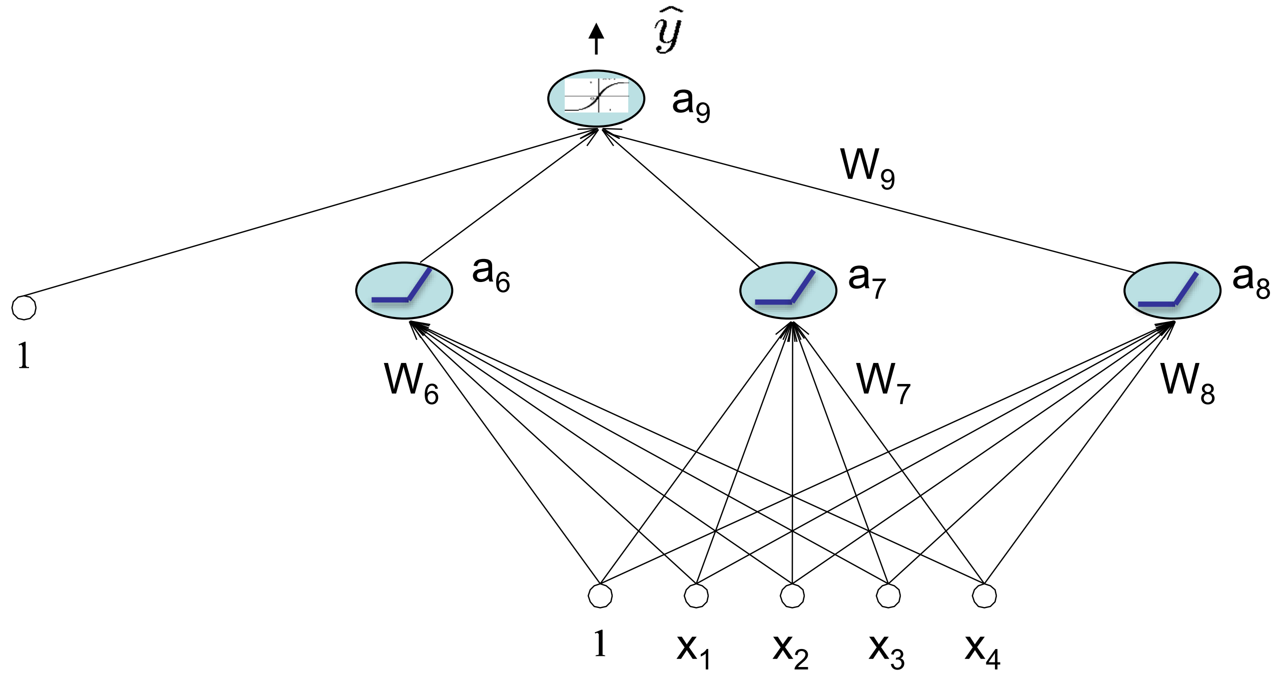
Define $\delta_6^i = \delta_9^i \cdot w_{9,6} \cdot a_6^i(1 - a_6^i)$

and rewrite as

$$\frac{\partial J_i(W)}{\partial w_{6,2}} = \delta_6^i \cdot x_2^i.$$

$$\frac{\partial J_i}{\partial W_6} = \delta_6^i \cdot X, \qquad \frac{\partial J_i}{\partial W_7} = \delta_7^i \cdot X, \qquad \frac{\partial J_i}{\partial W_8} = \delta_8^i \cdot X$$

ReLu in the hidden layer?



What will δ_9, δ_6, δ_7 and δ_8 be?

Impact of ReLu

- It reduces the issue of vanishing gradient
- It introduces sparsity in the hidden layer outputs
  - Randomly initialized weights leads to approximately 50% of the hidden nodes to output zero
- Such hidden nodes blocks the backpropagation
  - But this has proven to be not a substantial problem, as long as back-prop is not blocked on some paths

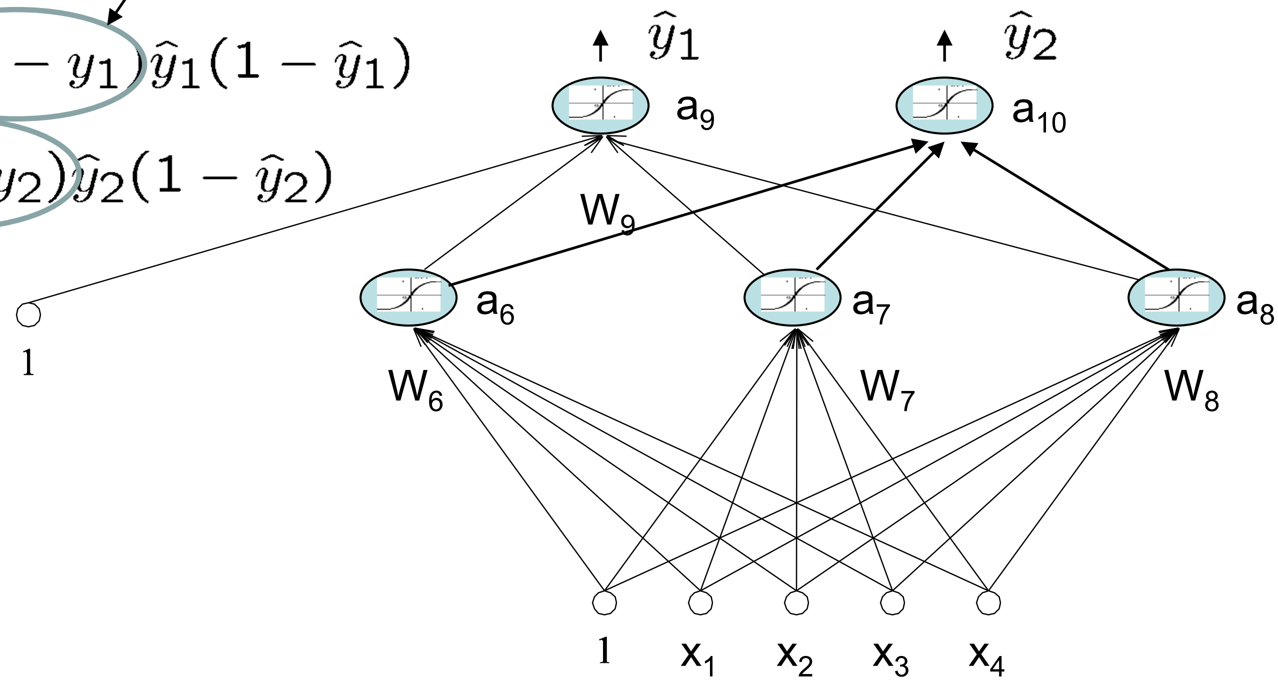
$$\frac{\partial L}{\partial \hat{y}_1}$$

Networks with Multiple Output Units

$$\delta_9 = (\hat{y}_1 - y_1)\hat{y}_1(1 - \hat{y}_1)$$

$$\delta_{10} = (\hat{y}_2 - y_2)\hat{y}_2(1 - \hat{y}_2)$$

$$\frac{\partial L}{\partial \hat{y}_2}$$



- We get a separate contribution to the gradient from each output unit.
- Hence, for input-to-hidden weights, we must sum up the contributions:

$$\delta_6 = a_6(1 - a_6)(w_{9,6}\delta_9 + w_{10,6}\delta_{10})$$

Backpropagation Training

- Initialize all the weights with small random values
- Repeat for T iterations

    Begin Epoch

    For each training example do*

    - Compute the network output
    - Compute loss
    - Backpropagate this loss from layer to layer and adjust weights to decrease this loss using gradient descent

$$\mathbf{w}_9 \leftarrow \mathbf{w}_9 + \gamma\ \delta_9^i \boldsymbol{A}^i$$

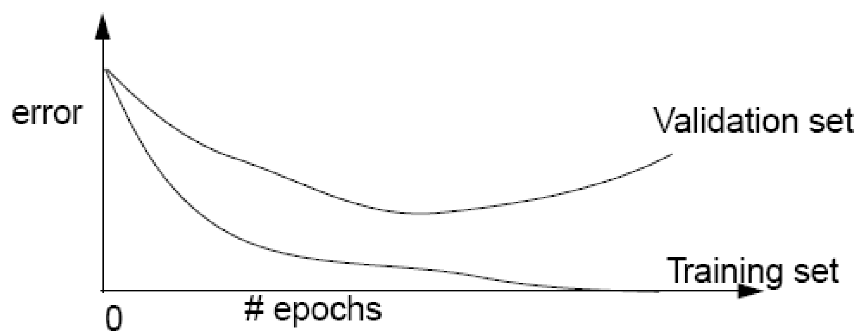
$$\mathbf{w}_6 \leftarrow \mathbf{w}_6 + \gamma \delta_6^i \mathbf{x}^i, \mathbf{w}_7 \leftarrow \mathbf{w}_7 + \gamma \delta_7^i \mathbf{x}^i, \mathbf{w}_8 \leftarrow \mathbf{w}_8 + \gamma \delta_8^i \mathbf{x}^i$$

    End Epoch

*This is online version of the training, where we update the network with each example. Batch or minibatch training can be easily achieved.

Remarks on Training

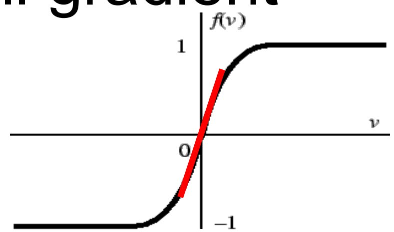
- Not guaranteed to convergence, may oscillate or reach a local minima.
- In practice many large networks can be adequately trained on large amounts of data for realistic problems, e.g.,
  - Driving a car or recognizing handwritten zip codes
  - Many epochs (thousands or more) may be needed for adequate training, large data sets may require extended training
- Termination criteria:
  - Overtraining is a real issue
  - Use a validation set to decide when to stop training



- To avoid bad local minima, run several trials with different random initialization and select the best according to the objective

Notes on Proper Initialization

- Start in the "linear" regions
  - keep all weights near zero, so that all sigmoid units are in their linear regions. This makes the whole net the equivalent of one linear threshold unit—a relatively simple function.
  - This will also avoid having very small gradient



- Break symmetry
  - If we start with all the weights equal, what would happen?
  - Ensure that each hidden unit has different input weights so that the hidden units move in different directions.

Batch, Online and Online with Momentum

- Batch. Sum up the gradient for a batch of examples and take a combined gradient step
- Online: Take a gradient step for each example
- Momentum: each update linearly combines the current gradient with the previous update direction to ensure smoother convergence

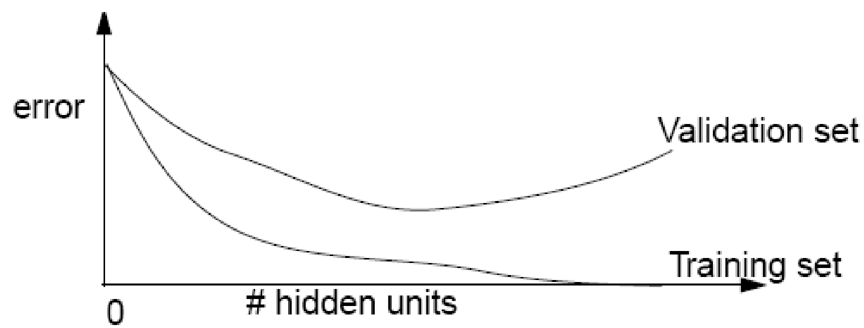
$$\Delta w_{i,j}(n) = \eta \delta_j x_{i,j} + \alpha \Delta w_{i,j}(n-1)$$

Current update Previous update

Useful resource for learning bells and whistles of SGD:
https://cs231n.github.io/neural-networks-3/#sgd

Curb Overfitting

- Too few hidden units underfit the data and fail to learn the concept.
- Too many hidden units over-fit



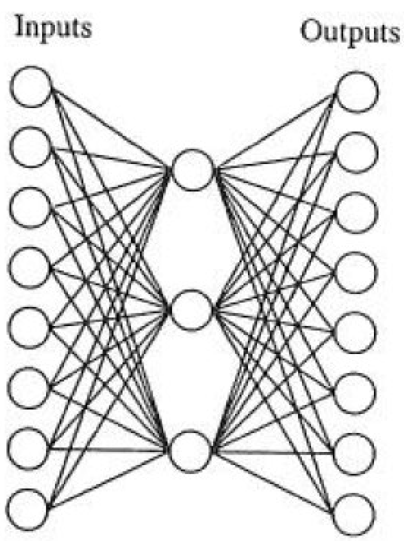
- Cross-validation can be used to decide # of hidden units.
- ***Weight decay*** multiplies all weights by some fraction between 0 and 1 after each epoch.
  - Encourages smaller weights and less overfitting
  - Equivalent to including a regularization term on the weights

Input/Output Coding

- Appropriate coding of inputs/outputs can make learning easier and improve generalization.
- Best to encode discrete multi-category features using multiple input units and include one binary unit per value
- Continuous inputs can be handled by a single input unit, but scaling them between 0 and 1
- For classification problems, best to have one output unit per class.
  - Continuous output values then represent certainty in various classes.
  - Assign test instances to the class with the highest output.
- If using MSE for objective for binary problems, use target values of 0.9 and 0.1 rather than forcing weights to grow large enough to closely approximate 0/1 outputs.
- Continuous outputs (regression) can also be handled by scaling to the range between 0 and 1 (for ease of training)

Hidden layer representation

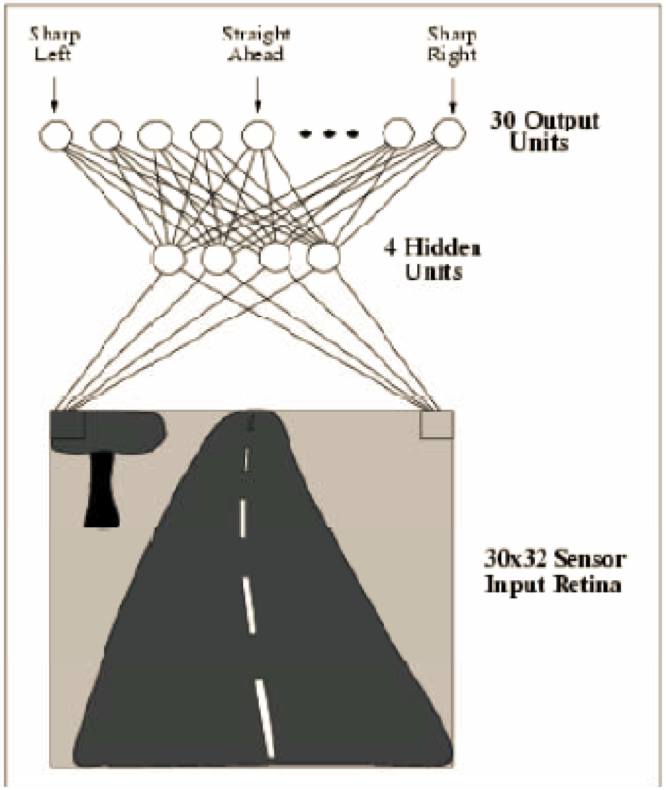
- Hidden nodes learn to discover useful intermediate representations
  - A intriguing property of multi-layer neural networks

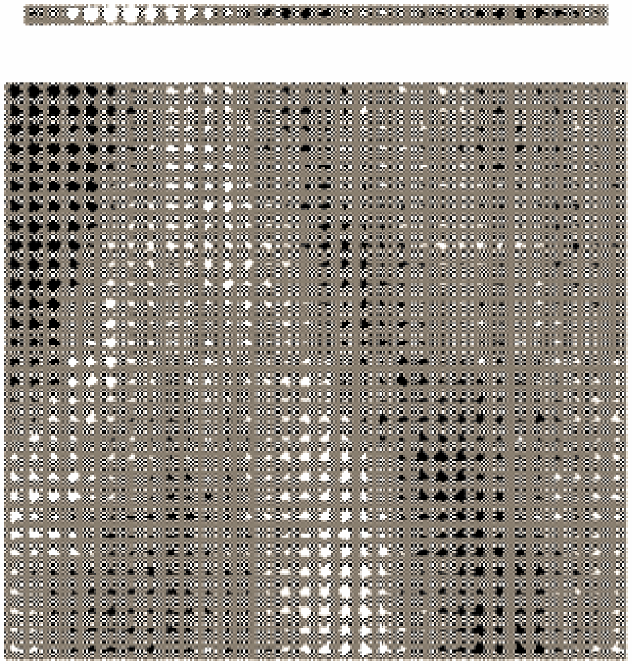


Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.15	.99	.99	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.01	.11	.88	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

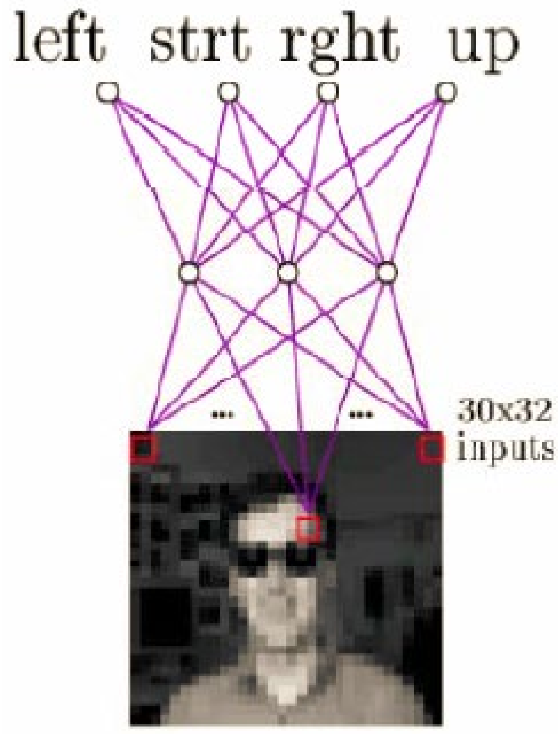
Example

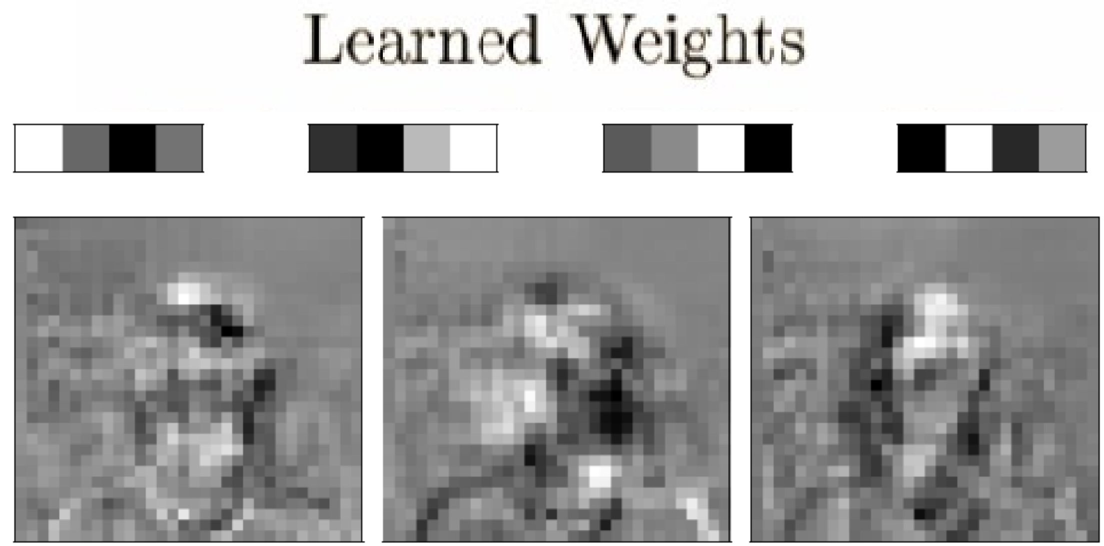
Neural net is one of the most effective methods when the data include complex sensory inputs such as images.





Example from Mitchell’s ML book pp. 84





Example from Mitchell's ML textbook pp. 113

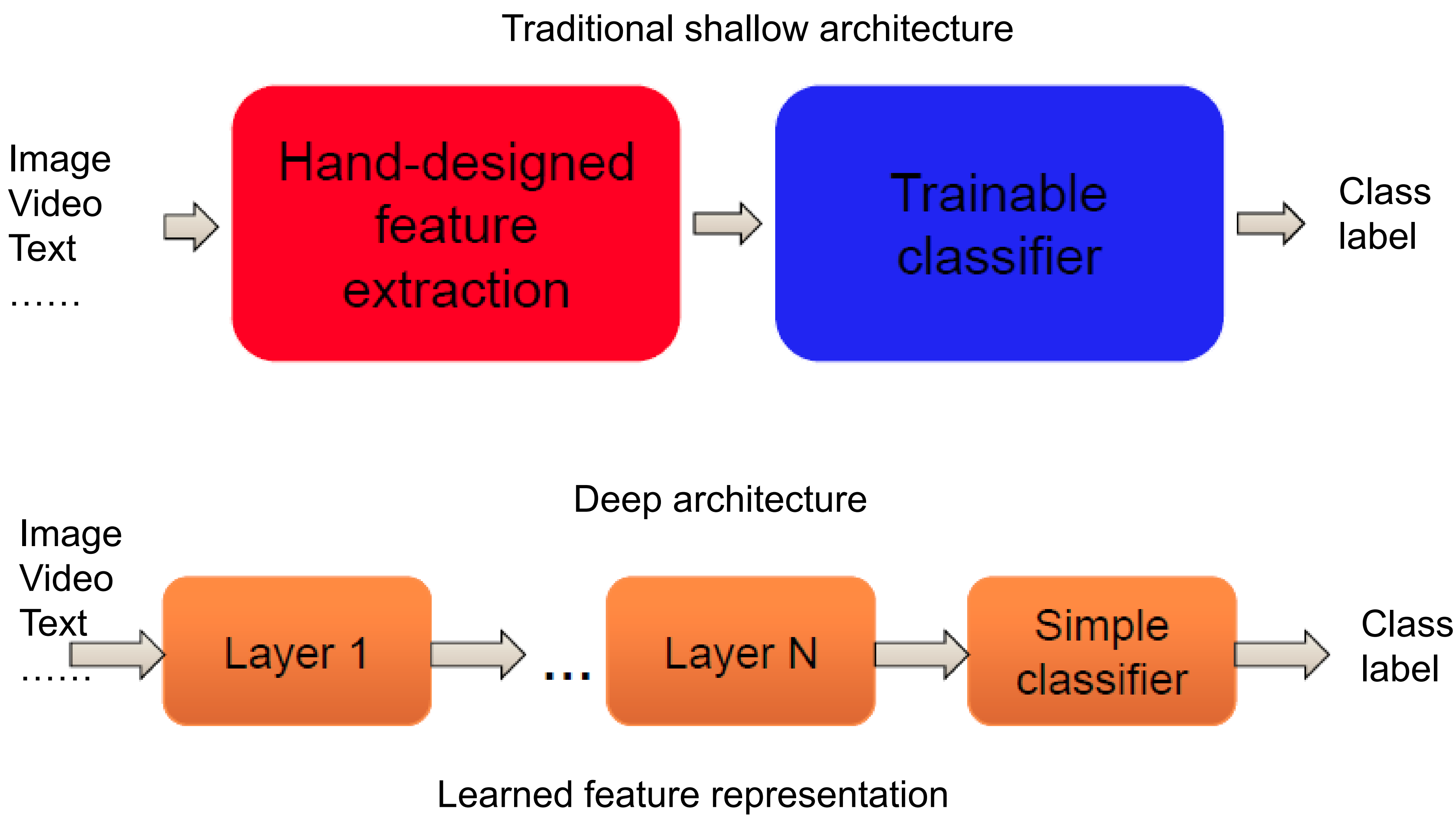
Typical input images

Such representations are not translation invariant

Recent Development

- More recent trend in ML is deep learning, which uses deep architecture and learns feature hierarchies (sometimes using large amount of unlabeled data via self-supervised learning)
- The learned feature hierarchies are expected to capture the inherent structure in the data
- Often lead to substantially improved classification when used the learned features to train with labeled data

Shallow vs Deep Architectures



Want to learn more about deep neural networks and ML in general?

- There are many types of deep learning architectures – Convolutional networks, recurrent networks, Transformer …
- Various packages: Pytorch, Tensorflow …
- Tremendous impacts in vision, speech and natural language processing
- Very fast growing area, to learn more, take the deep learning class (AI535)
- Other related AI classes:
  - PGM (AI536), NLP (AI 539 deep learning version and classic version), RL (AI 533), ML2 (AI 539 in Spring)