Supervised Learning: Artificial Neural Networks

Some slides adapted from Dan Klein et al. (Berkeley) and Percy Liang (Stanford)

Plan

- Project 5: http://www.mathcs.emory.edu/~eugene/cs325/p5/
- Review: Perceptron, MIRA
- New: Neural Networks (Supervised version)
  – +Hidden Layer
  – +Training algorithms (forward-, backword-propagation)

MIRA*: Fixing the Perceptron

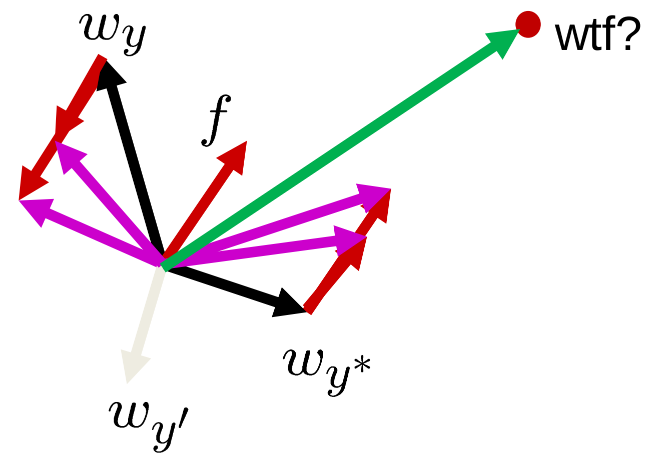
- What about "outliers" – correct or incorrect data points with abnormal feature values?
- MIRA: choose an update size that fixes the current mistake…
- … but, minimizes the change to w

$$\min_{w} \frac{1}{2} \sum_{y} ||w_y - w'_y||^2$$

$$w_{y^*} \cdot f(x) \geq w_y \cdot f(x) + 1$$

- The +1 helps to generalize

* Margin Infused Relaxed Algorithm



Guessed y instead of y^* on example x with features $f(x)$

$$w_y = w'_y - \tau f(x)$$
$$w_{y^*} = w'_{y^*} + \tau f(x)$$

Minimum Correcting Update

$$\min_w \frac{1}{2}\sum_y ||w_y - w'_y||^2$$
$$w_{y^*} \cdot f \geq w_y \cdot f + 1$$

$$\Downarrow$$

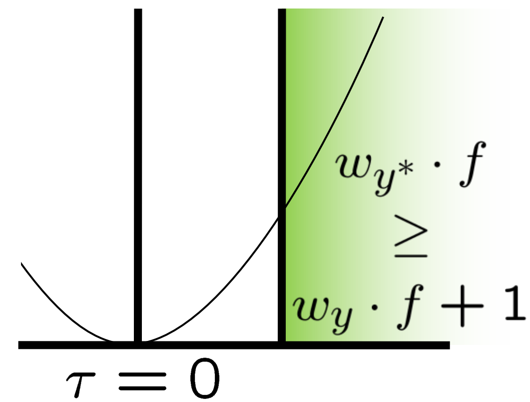
$$\min_\tau ||\tau f||^2$$
$$w_{y^*} \cdot f \geq w_y \cdot f + 1$$

$$\Downarrow$$

$$(w'_{y^*} + \tau f) \cdot f = (w'_y - \tau f) \cdot f + 1$$

$$\boxed{\tau = \frac{(w'_y - w'_{y^*}) \cdot f + 1}{2 f \cdot f}}$$

$$\boxed{\begin{aligned} w_y &= w'_y - \tau f(x) \\ w_{y^*} &= w'_{y^*} + \tau f(x) \end{aligned}}$$



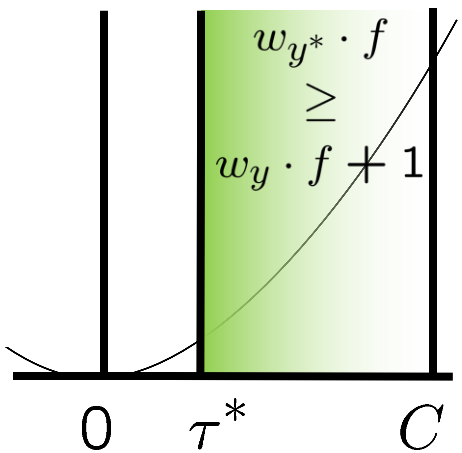
min not τ=0, or would not have made an error, so min will be where equality holds

Maximum Step Size

- In practice, it's also bad to make updates that are too large
  - Example may be labeled incorrectly
  - You may not have enough features
  - Solution: cap the maximum possible value of τ with some constant C

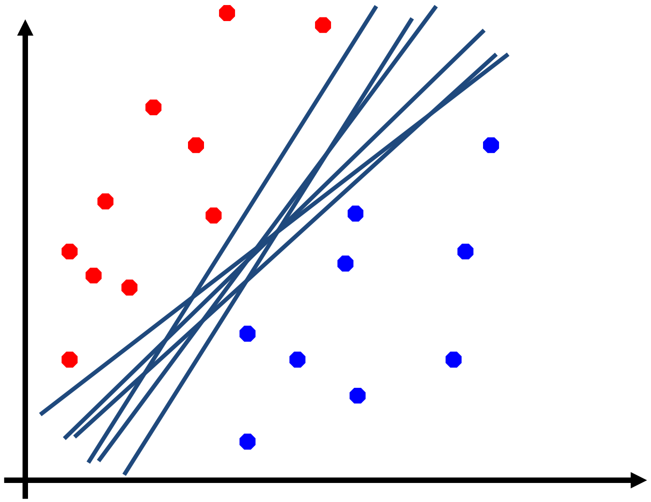
$$\tau^* = \min\left(\frac{(w'_y - w'_{y^*}) \cdot f + 1}{2f \cdot f}, C\right)$$

  - Corresponds to an optimization that assumes non-separable data
  - Usually converges faster than perceptron
  - Usually better, especially on noisy data



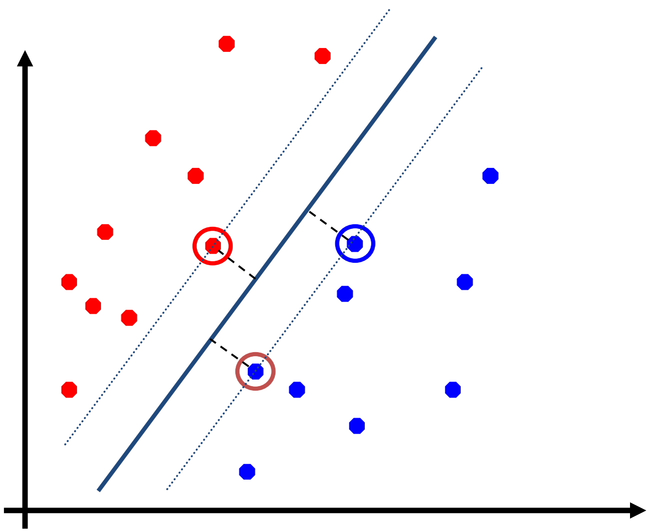
Linear Separators

- Which of these linear separators is optimal?



Support Vector Machines

- Maximizing the margin: good according to intuition, theory, practice
- Only support vectors matter; other training examples are ignorable
- Support vector machines (SVMs) find the separator with max margin
- Basically, SVMs are MIRA where you optimize over all examples at once



MIRA

$$\min_w \frac{1}{2}||w - w'||^2$$

$$w_{y^*} \cdot f(x_i) \geq w_y \cdot f(x_i) + 1$$

SVM

$$\min_w \frac{1}{2}||w||^2$$

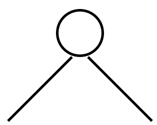
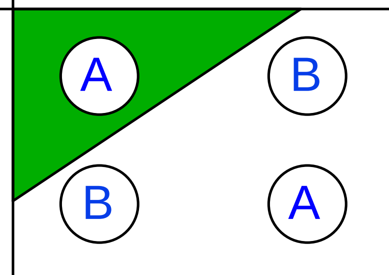
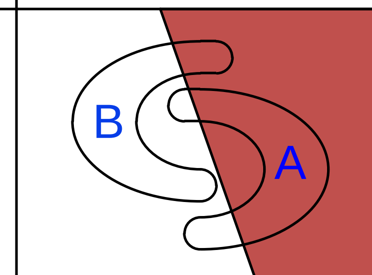
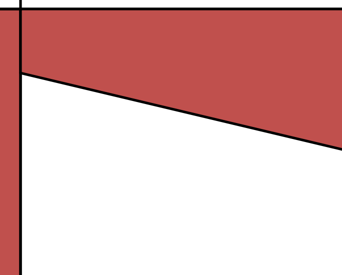
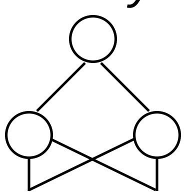
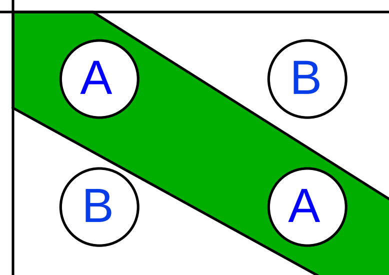
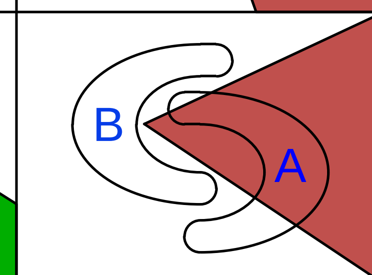
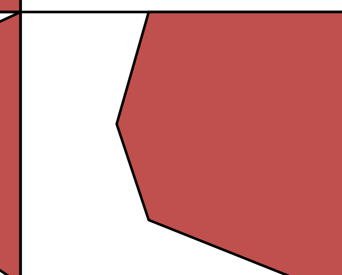
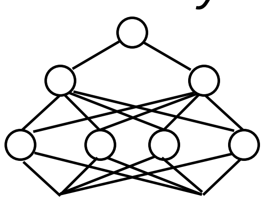
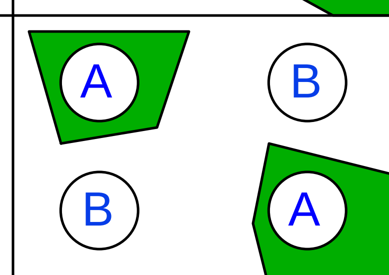
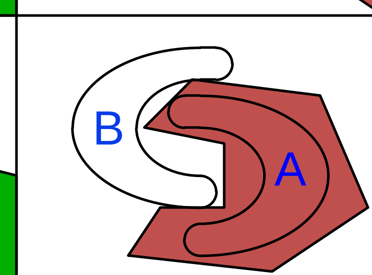
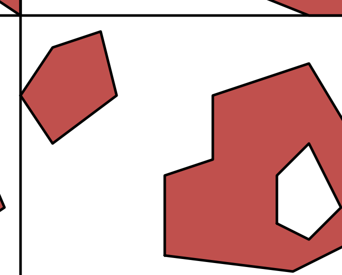
$$\forall i, y \; w_{y^*} \cdot f(x_i) \geq w_y \cdot f(x_i) + 1$$

Classification: Comparison

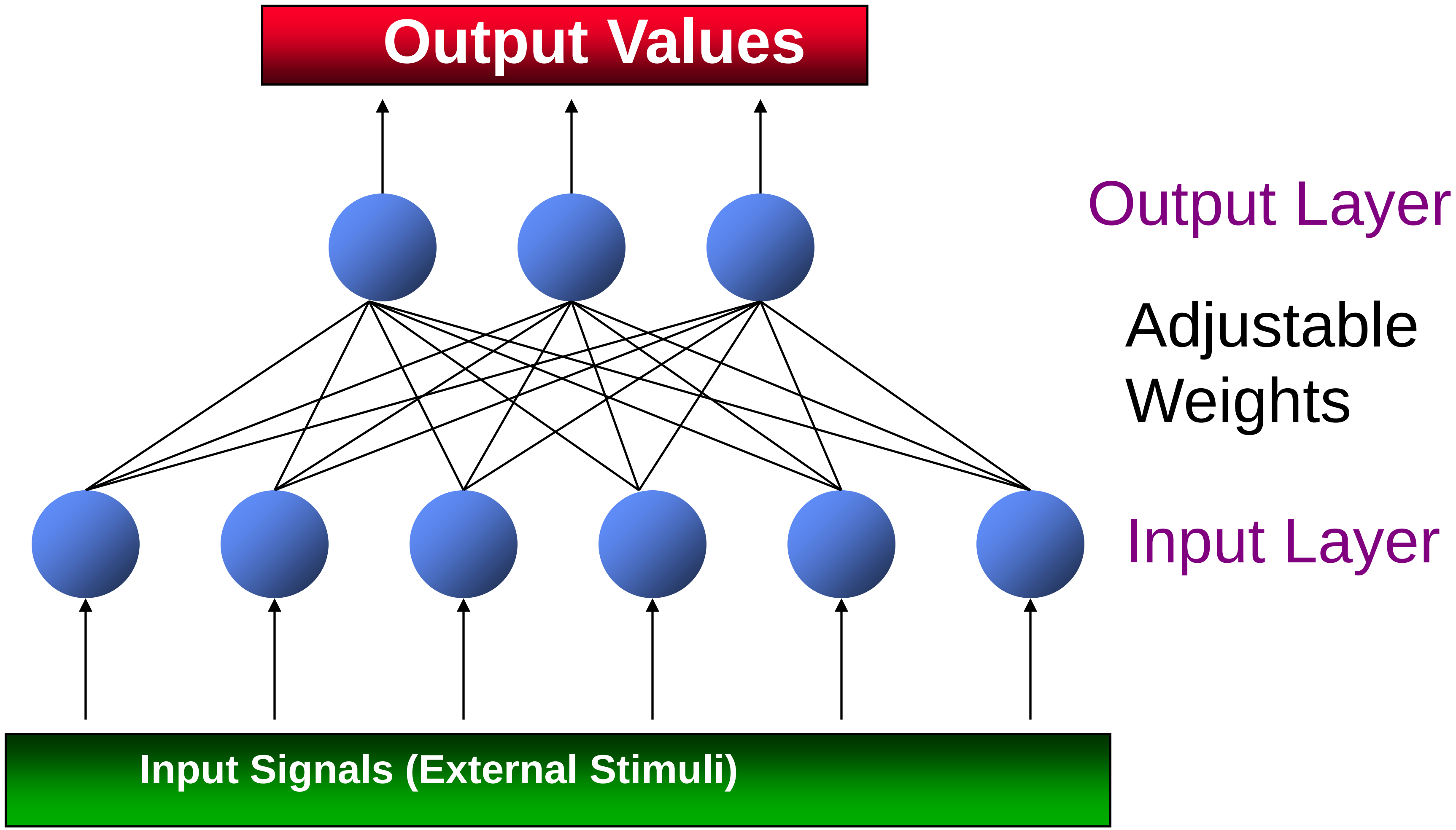
- Naïve Bayes
  - Builds a model training data
  - Gives prediction probabilities
  - Strong assumptions about feature independence
  - One pass through data (counting)

- Perceptrons / MIRA:
  - Makes less assumptions about data
  - Mistake-driven learning
  - Multiple passes through data (prediction)
  - Often more accurate

Different Non-Linearly Separable Problems

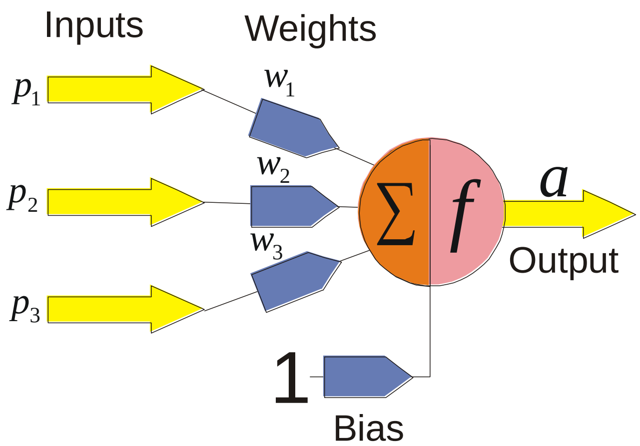
Structure	Types of Decision Regions	Exclusive-OR Problem	Classes with Meshed regions	Most General Region Shapes
Single-Layer 	*Half Plane Bounded By Hyperplane*			
Two-Layer 	*Convex Open Or Closed Regions*			
Three-Layer 	Arbitrary (Complexity Limited by No. of Nodes)			

Multilayer Perceptron (MLP)



The Key Elements of Neural Networks

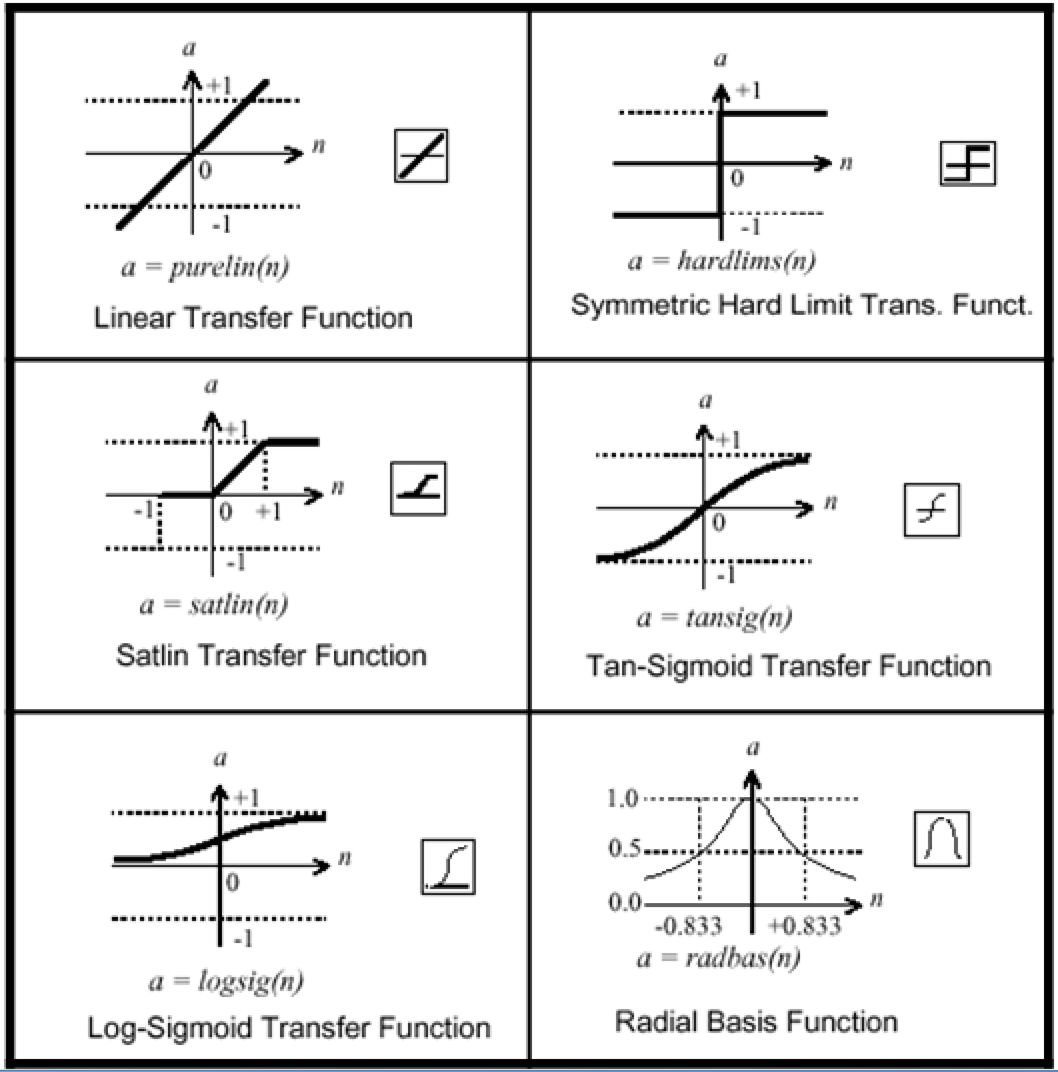
- Neural computing requires a number of neurons, to be connected together into a neural network. Neurons are arranged in layers.



$$a = f\left(p_1 w_1 + p_2 w_2 + p_3 w_3 + b\right) = f\left(\sum p_i w_i + b\right)$$

- Each neuron within the network is usually a simple processing unit which takes one or more inputs and produces an output. At each neuron, every input has an associated weight which modifies the strength of each input. The neuron simply adds together all the inputs and calculates an output to be passed on.

Activation functions



Hidden Layers

- Intermediate hidden units: **learned features**
  - activation vector *h* behaves a like our feature vector *f(x)* for linear classifier. But, *h* is “learned” automatically.

Key idea: feature learning

Before: manually specify features

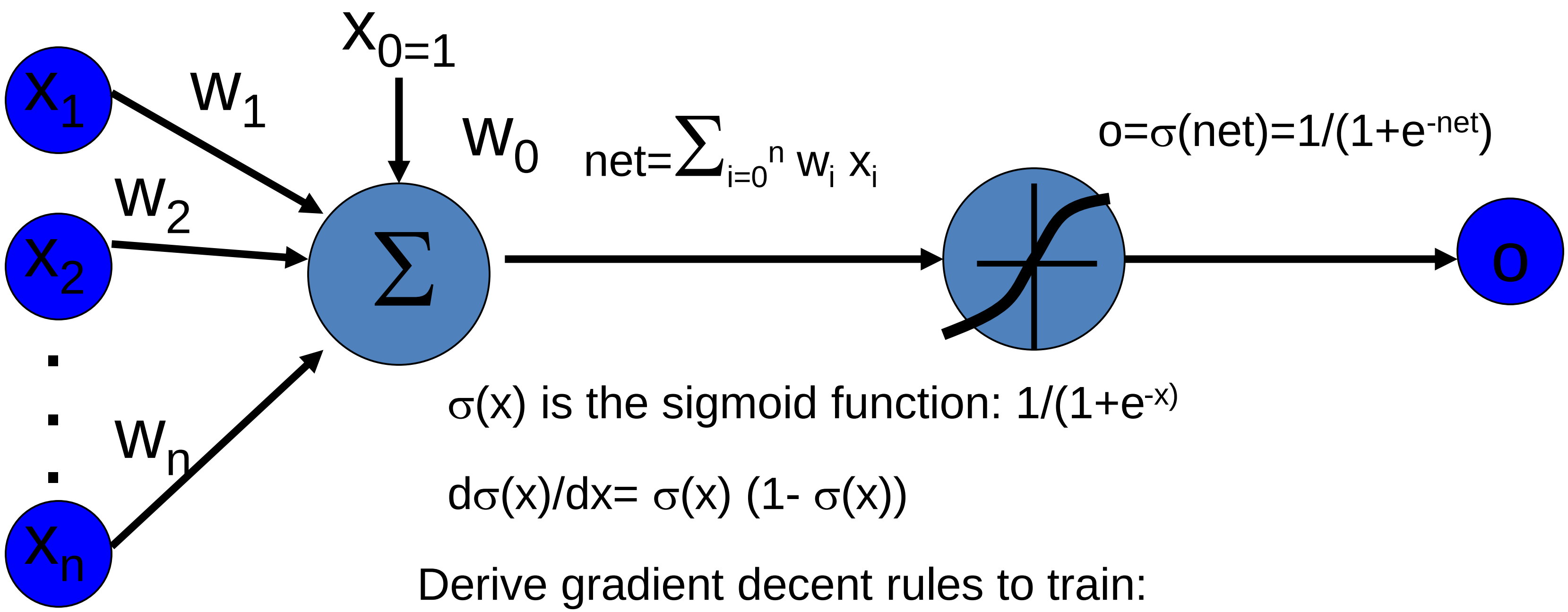
$$\phi(x)$$

Now: automatically learn them from data

$$h(x) = [h_1(x), \ldots, h_k(x)]$$

- What kind of features that can be learned?
  - must be of the form x→ $\sigma(\mathbf{v}_j \cdot \phi(x))$ $\sigma(z) = (1 + e^{-z})^{-1}$

Sigmoid Unit



$\sigma(x)$ is the sigmoid function: $1/(1+e^{-x})$

$d\sigma(x)/dx = \sigma(x)\,(1 - \sigma(x))$

Derive gradient decent rules to train:

- one sigmoid function

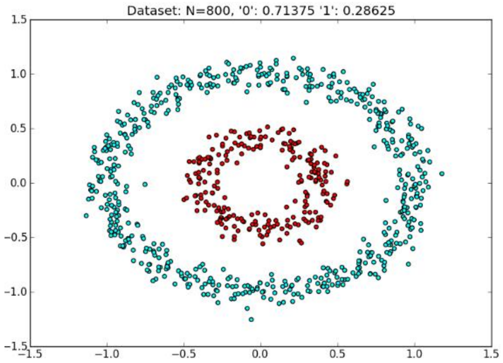
  $\partial E/\partial w_i = -\sum_d (t_d - o_d)\, o_d\, (1 - o_d)\, x_i$

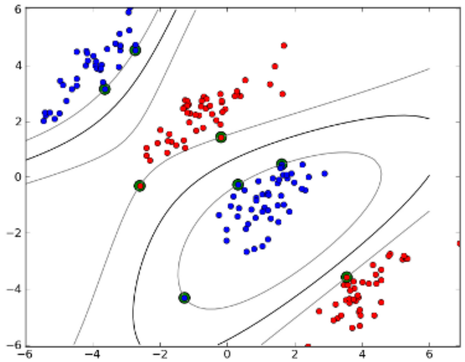
- Multilayer networks of sigmoid units
  backpropagation:

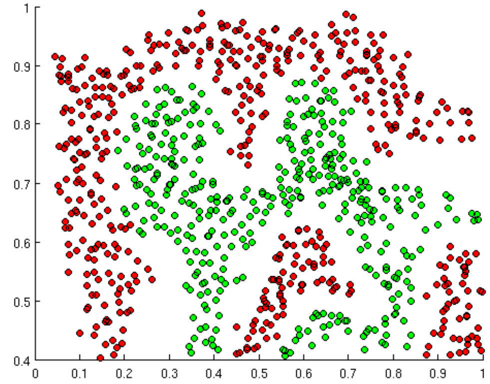
Demo (TensorFlow)

- http://playground.tensorflow.org
- - 0 hidden layers: perceptron (linear)
- 1 hidden layer: multi-layer perceptron (next)

Code (ConvNet.JS)







- Demo: http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html

```
layer_defs = [];
layer_defs.push({type:'input', out_sx:1, out_sy:1, out_depth:2});
layer_defs.push({type:'fc', num_neurons:1, activation: 'sigmoid'});
layer_defs.push({type:'softmax', num_classes:2});

net = new convnetjs.Net();
net.makeLayers(layer_defs);

trainer = new convnetjs.SGDTrainer(net, {learning_rate:0.01, momentum:0.1,
                                   batch_size:10, l2_decay:0.001});
```

Online Demo: Add 2nd Layer

- Demo: http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html

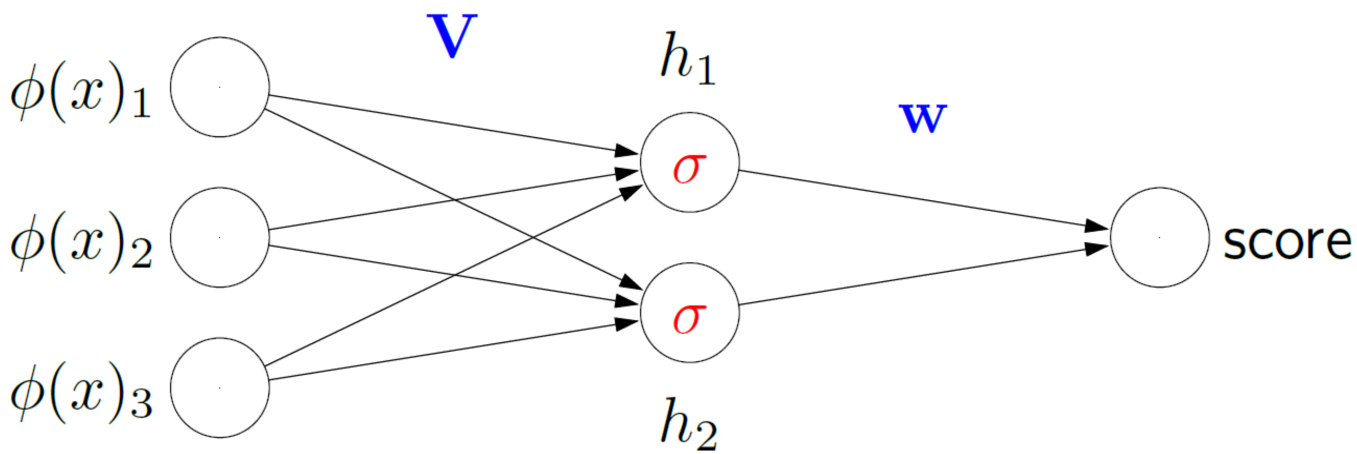
```
layer_defs = [];
layer_defs.push({type:'input', out_sx:1, out_sy:1, out_depth:2});
layer_defs.push({type:'fc', num_neurons:5, activation: 'sigmoid'});
layer_defs.push({type:'softmax', num_classes:2});

net = new convnetjs.Net();
net.makeLayers(layer_defs);

trainer = new convnetjs.SGDTrainer(net, {learning_rate:0.01, momentum:0.1,
            batch_size:10, l2_decay:0.001});
```

2-Layer Neural Network

Neural network:



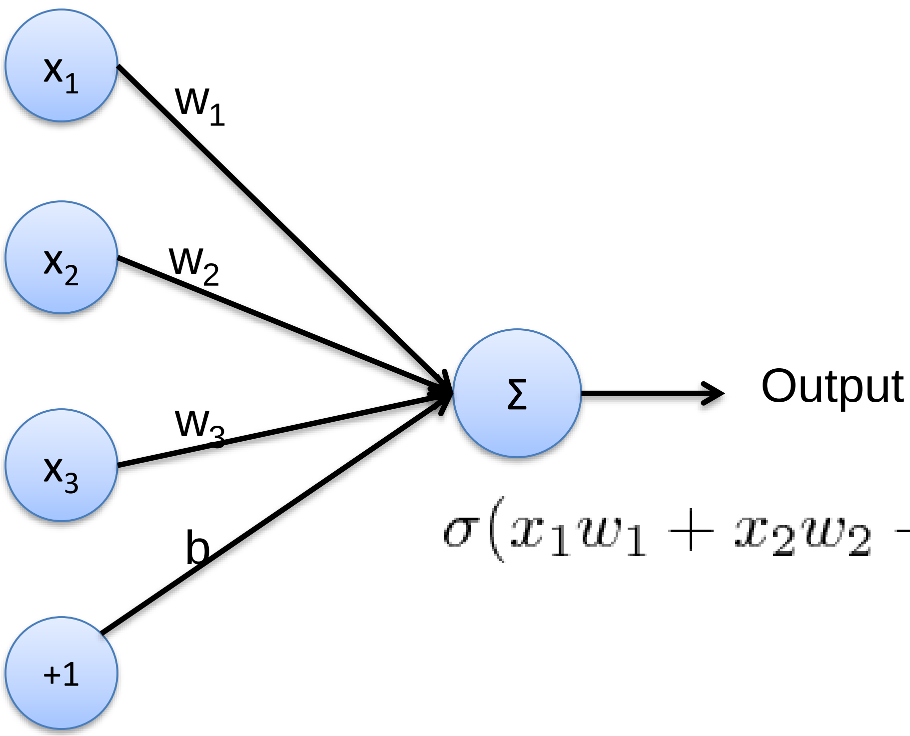
Intermediate hidden units:

$$h_j = \sigma(\mathbf{v}_j \cdot \phi(x)) \qquad \sigma(z) = (1 + e^{-z})^{-1}$$

Output:

$$\text{score} = \mathbf{w} \cdot \mathbf{h}$$

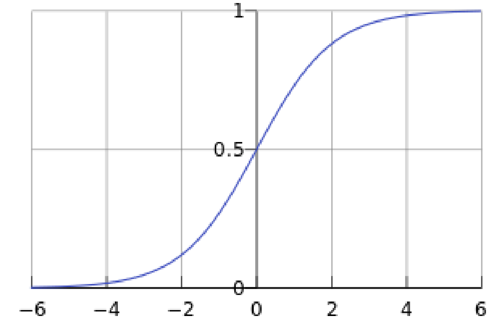
(Feed-forward) Neural Network



$$\sigma(x_1 w_1 + x_2 w_2 + x_3 w_3 + b) = \sigma(Wx + b)$$

(Feed-forward) Neural Network

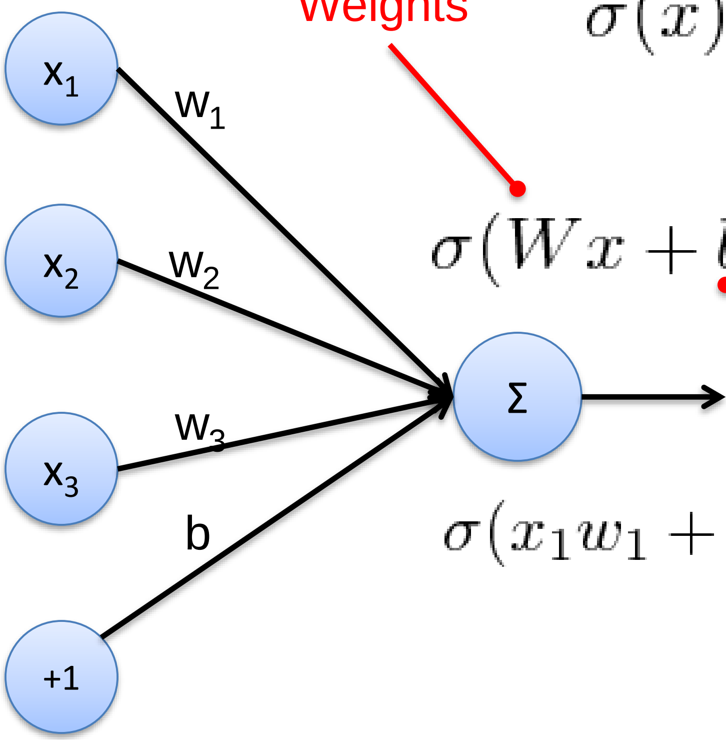
$$\sigma(x) = \frac{1}{1+e^{-x}}$$



Weights

$$\sigma(Wx+b)$$

bias

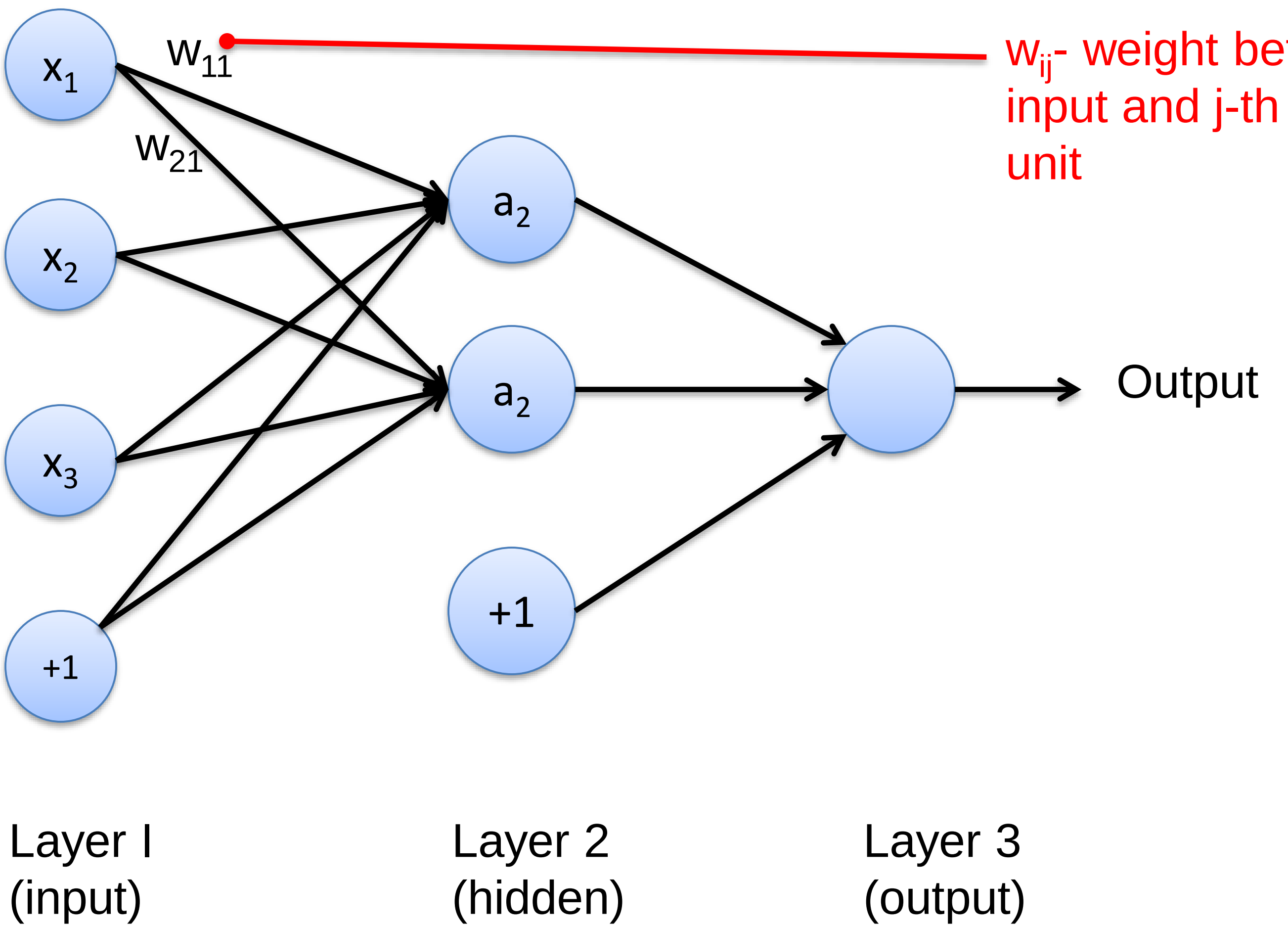


Output

$$\sigma(x_1w_1 + x_2w_2 + x_3w_3 + b) = \sigma(Wx+b)$$

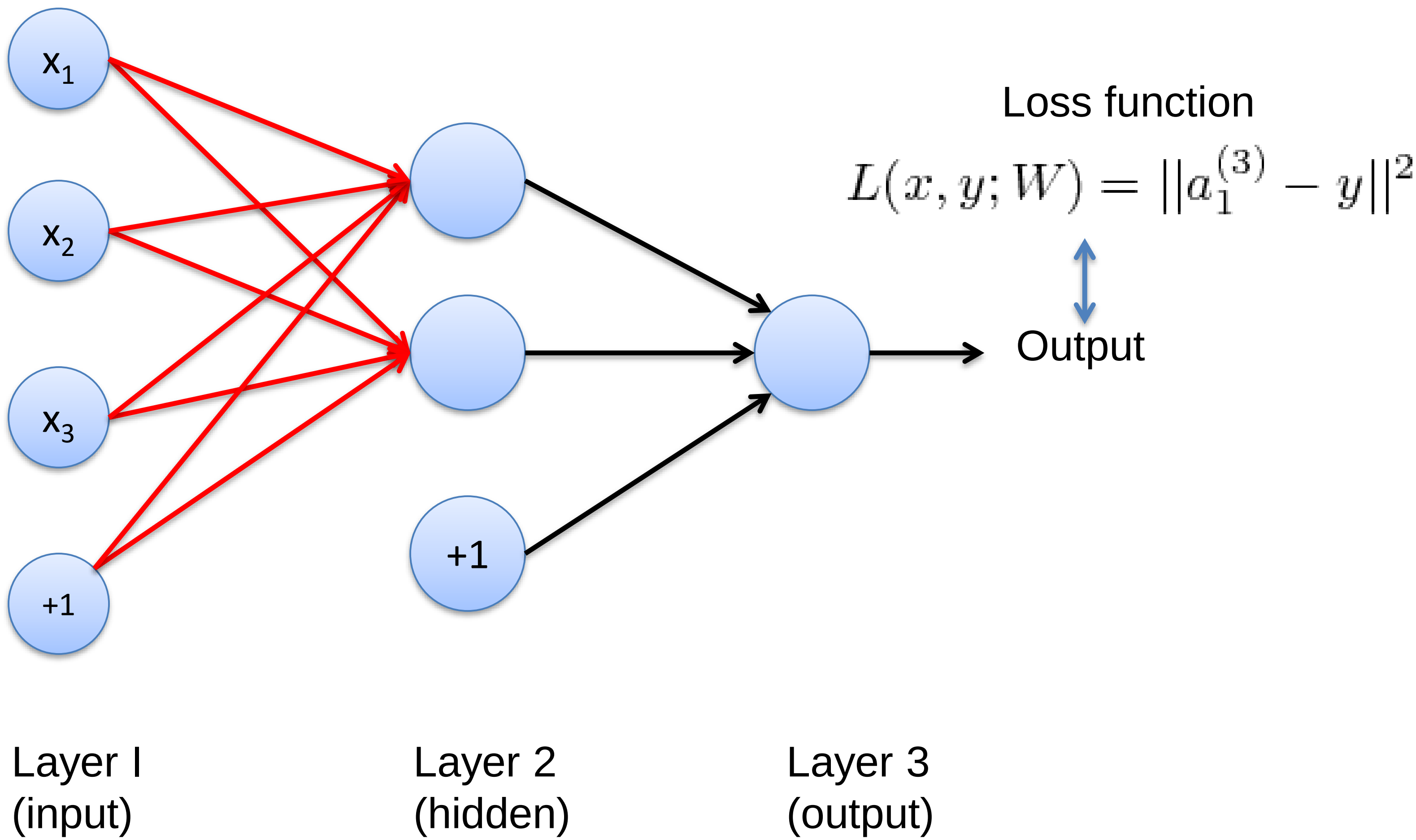
(Feed-forward) Neural Network

Stack many non-linear units (neurons)

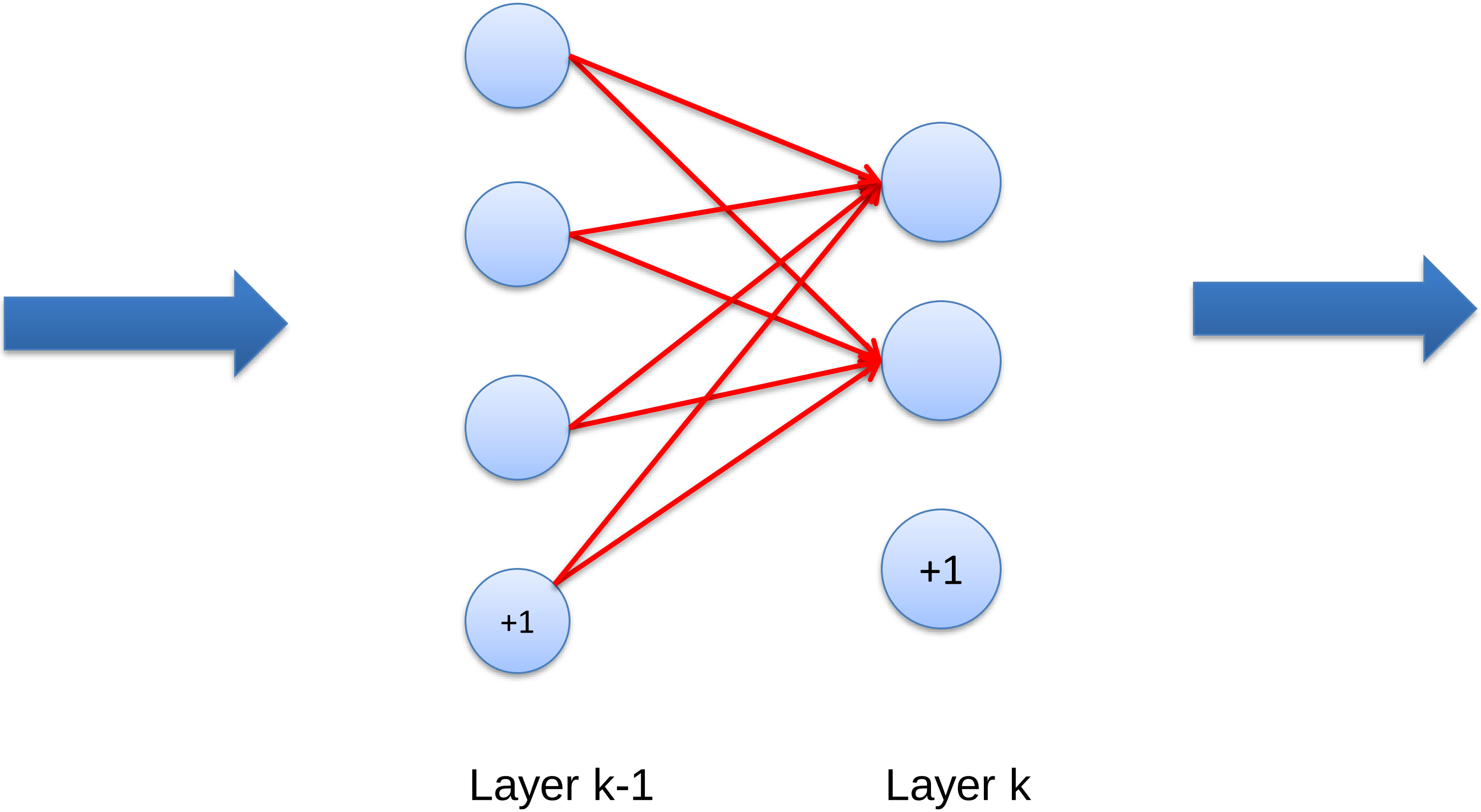


w_{ij}- weight between i-th input and j-th hidden unit

Training: Forward Pass

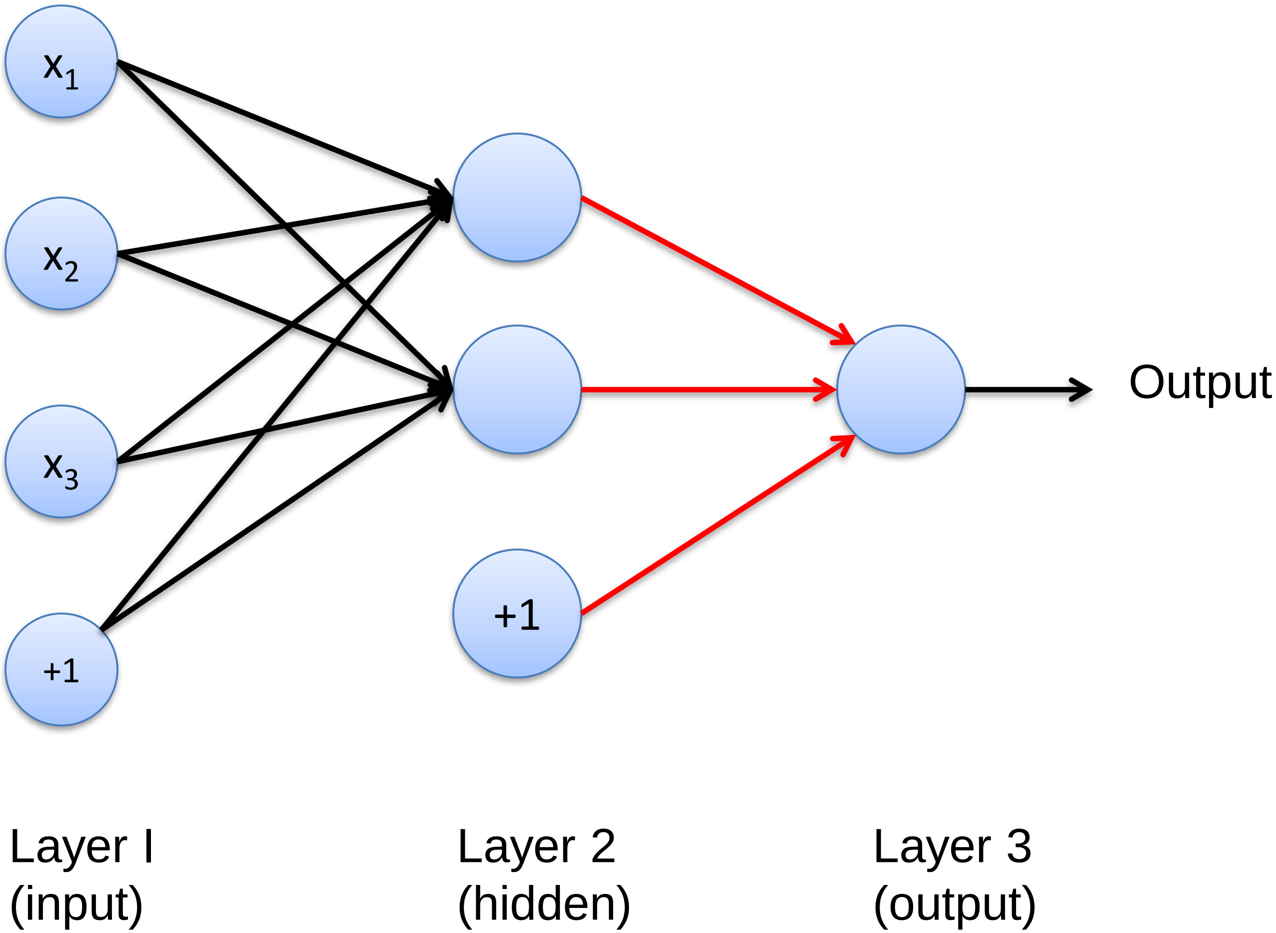


Training: Forward Pass



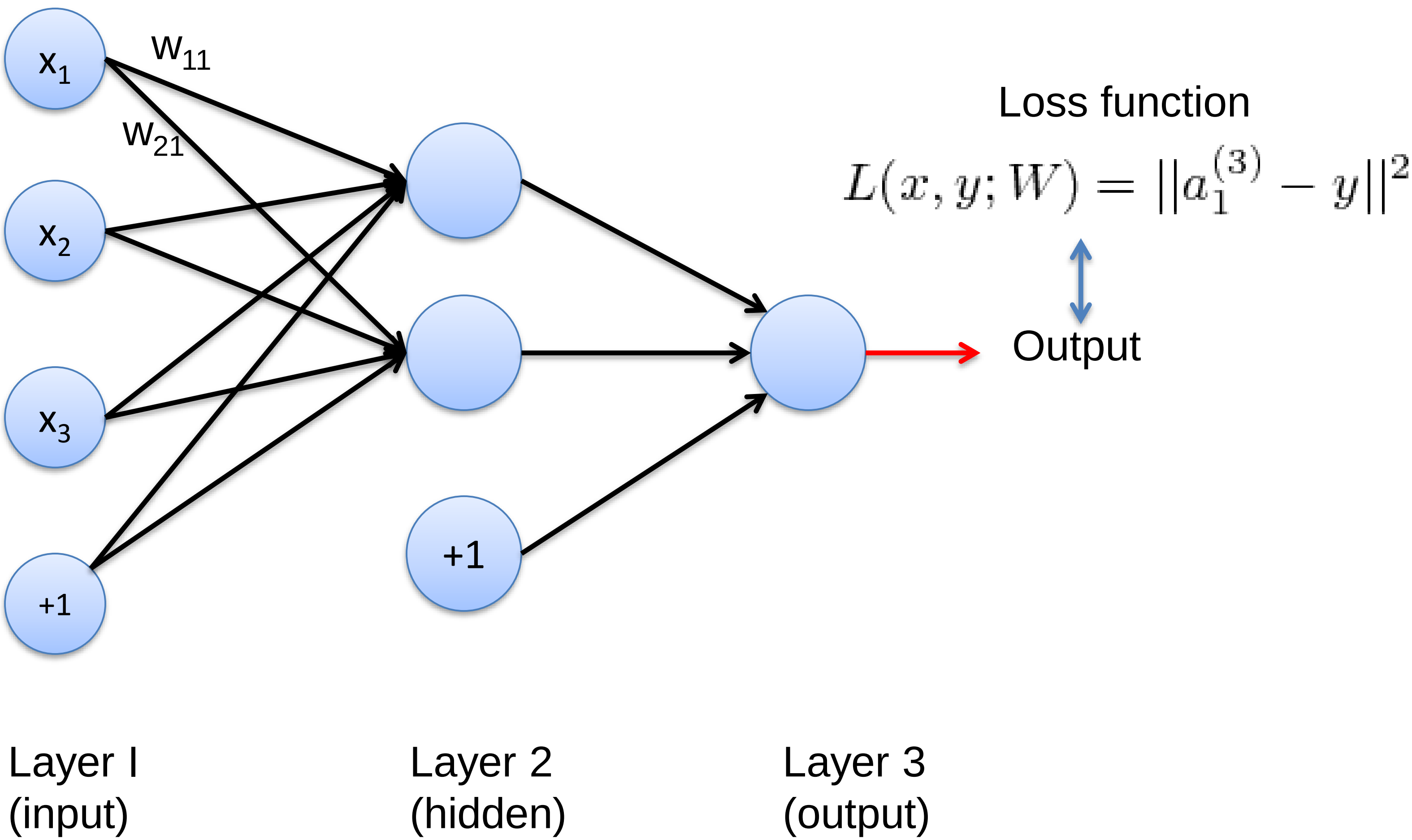
Training: Forward Pass

Stack many non-linear units (neurons)

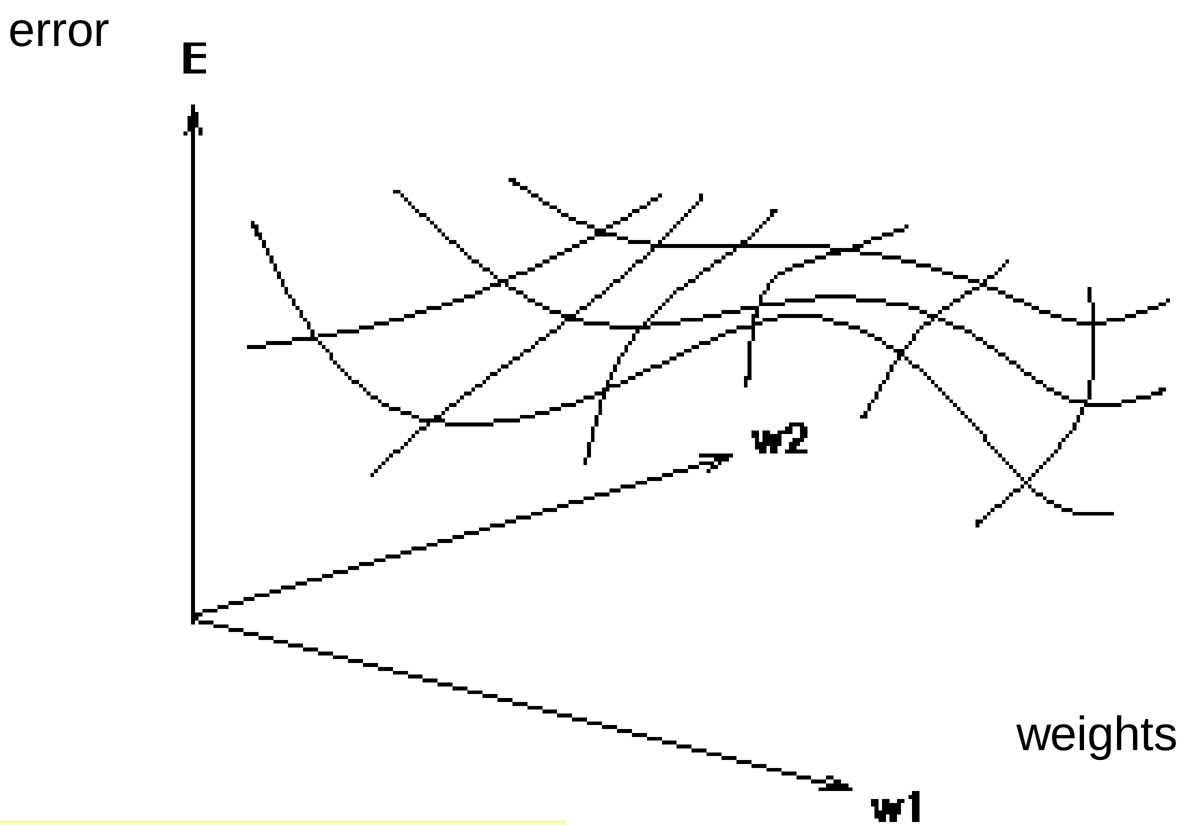


Training: Forward Pass

Stack many non-linear units (neurons)



Error Surface



Error as function of weights in multidimensional space

Gradient

Compute deltas

- Trying to make error decrease the fastest
- Compute:
  - $Grad_E$ = [dE/dw1, dE/dw2, . . ., dE/dwn]
- Change i-th weight by
  - $delta_{wi}$ = -***alpha*** * dE/dwi

Derivatives of error for weights

- We need a derivative!
- Activation function must be **continuous**, differentiable, non-decreasing, and easy to compute

Can’t use LTU

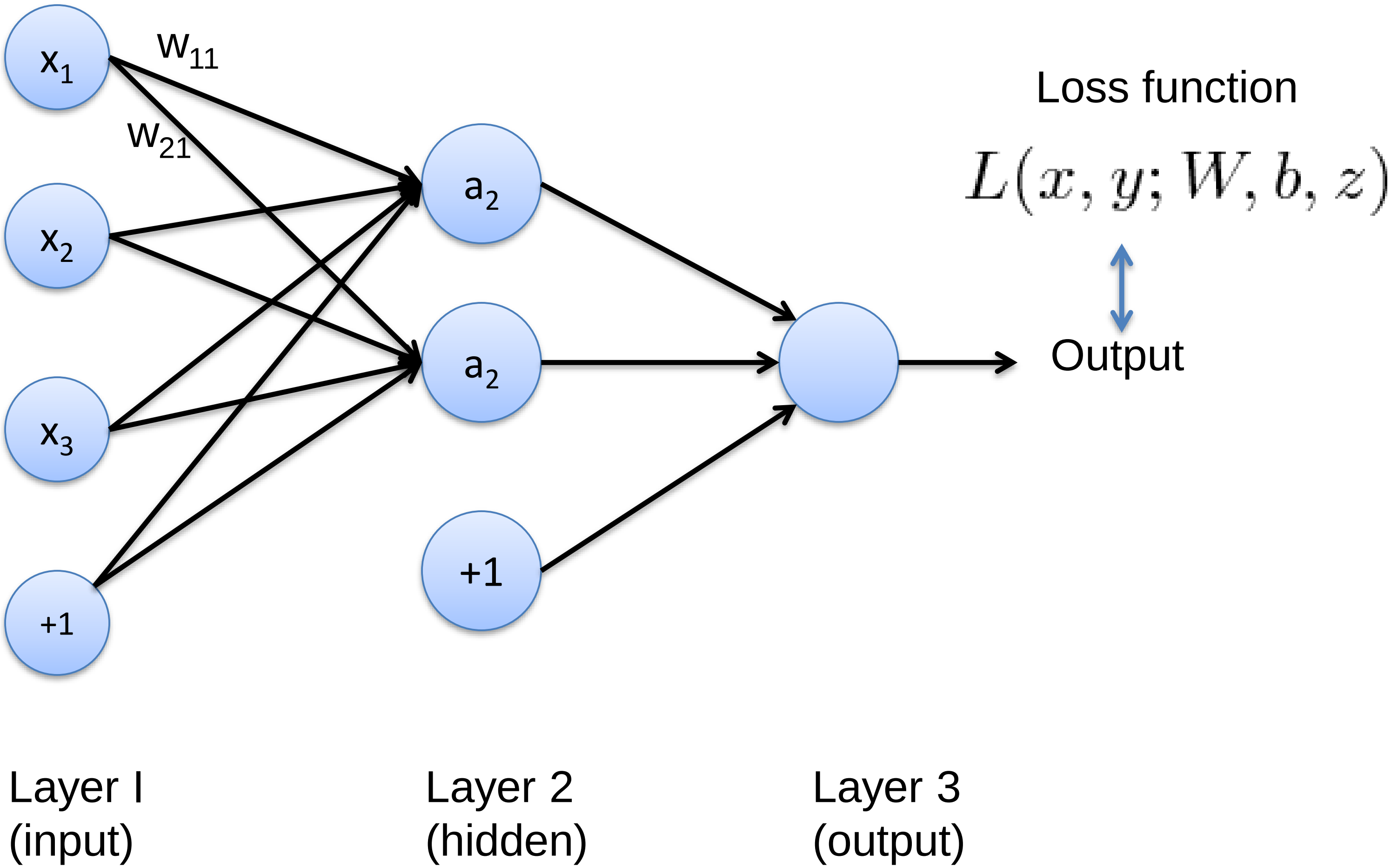
- To effectively assign credit / blame to units in hidden layers, we want to look at the first derivative of the activation function

- Sigmoid function is easy to differentiate and easy to compute forward

~~Linear Threshold Units~~ → Sigmoid function

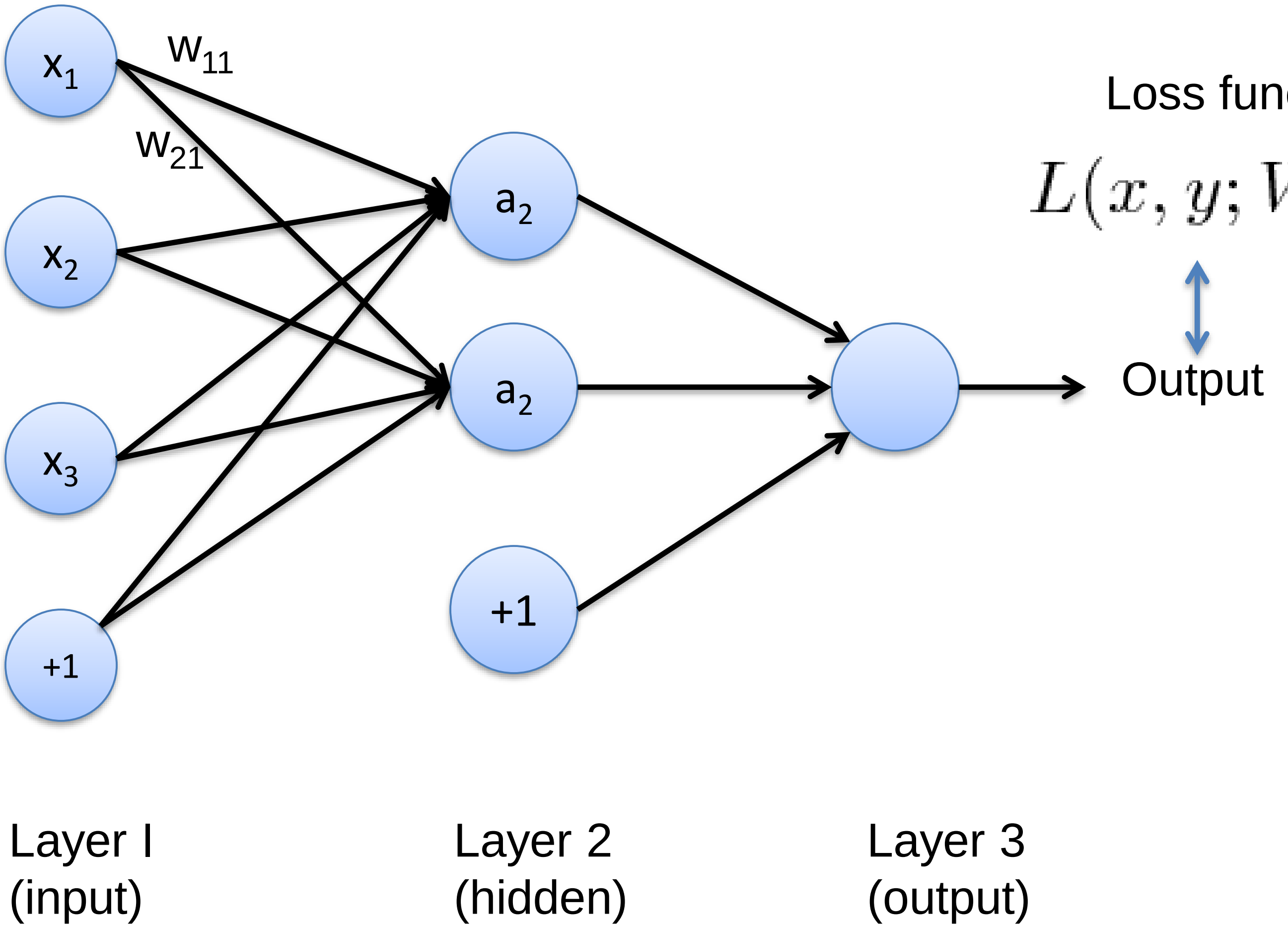
Training: Loss Function

Stack many non-linear units (neurons)



Training: Loss Function

Stack many non-linear units (neurons)



Loss function

$$L(x, y; W, b, z)$$

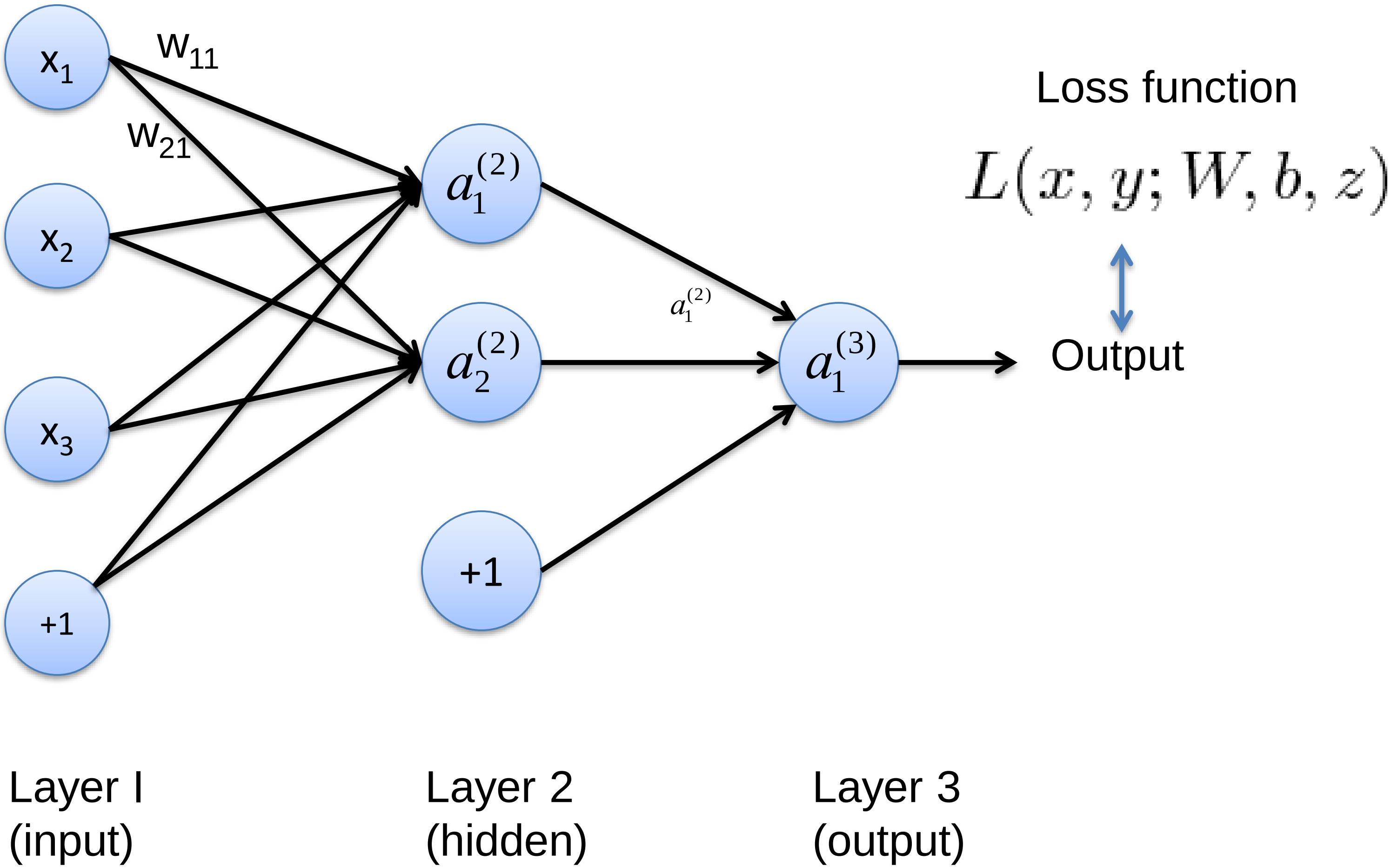
Loss functions

- Classification
  - Cross entropy (binary and multiclass)

$$crossentropy(t, o) = -(t \cdot log(o) + (1 - t) \cdot log(1 - o))$$

- Regression
  - Squared deviation (mean squared error)

Training: Loss Function



Backpropagation Algorithm – Main Idea – error in hidden layers

The ideas of the algorithm can be summarized as follows :

1. Computes the **error term for the output units** using the observed error.

2. From output layer, repeat
   - propagating the error term back to the previous layer and
   - updating the weights between the two layers

   until the earliest hidden layer is reached.

Backpropagation Algorithm

- Initialize weights (typically random!)
- Keep doing epochs
  - For each example **e** in training set do
    - **forward pass** to compute
      - O = neural-net-output(network,e)
      - miss = (T-O) at each output unit
    - **backward pass** to calculate deltas to weights
    - update all weights
  - end
- until tuning set error stops improving

Forward pass explained earlier

Backward pass explained in next slide

Backward Pass

- Compute deltas to weights
  - from hidden layer
  - to output layer

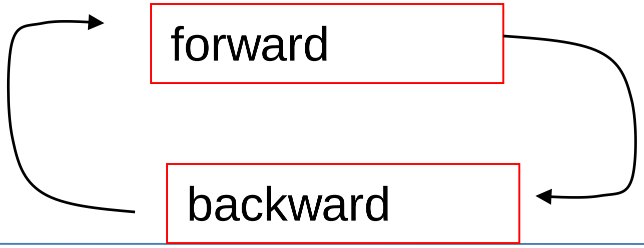
- Without changing any weights (yet), compute the actual contributions
  - within the hidden layer(s)
  - and compute deltas

Training Learning Details

- Method for learning weights in feed-forward (FF) nets

- Can't use Perceptron Learning Rule
  - no teacher values are possible for hidden units

- Use gradient descent to minimize the error
  - propagate deltas to adjust for errors
    backward from outputs
      to hidden layers
        to inputs



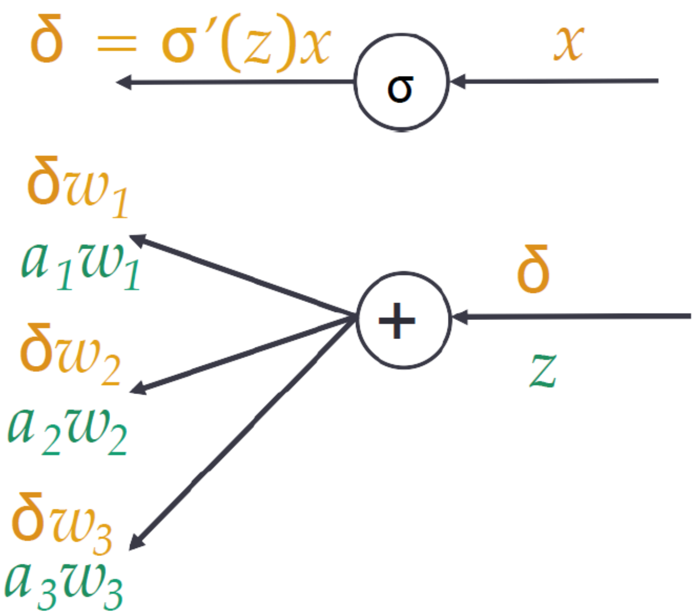
Backprop: another intuition

The chain rule of differentiation just boils down very simple patterns in error backpropagation:

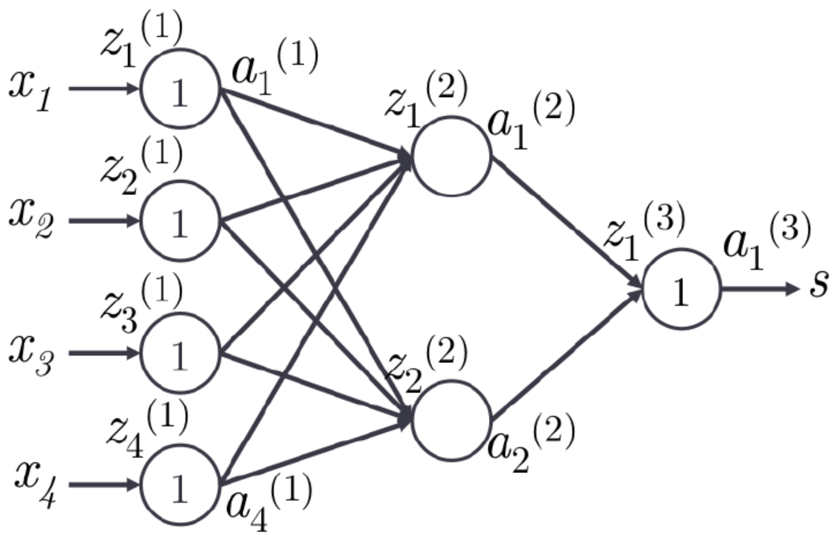
Orange = Backprop.
Green = Fwd. Pass

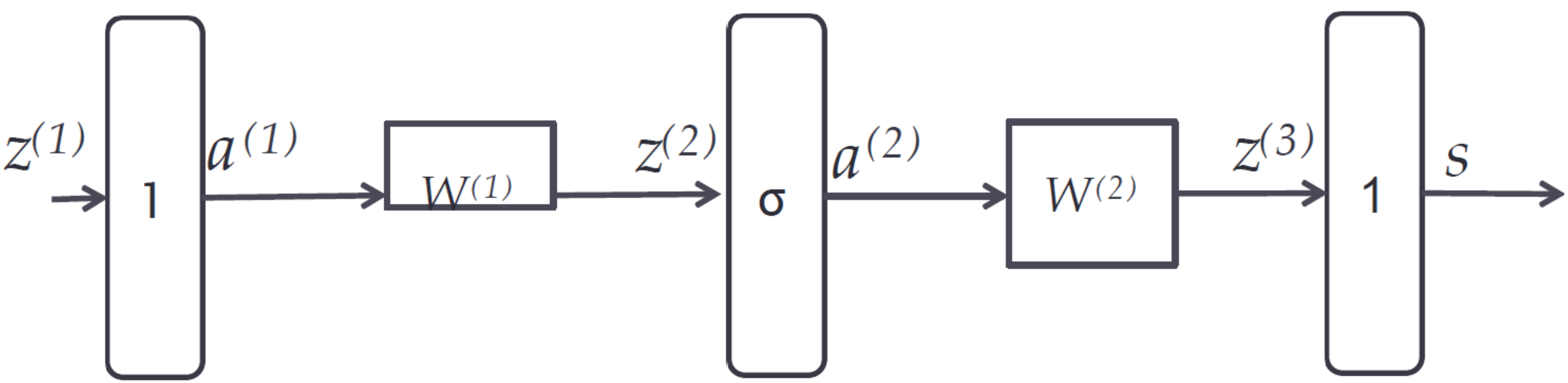
1. An error x flowing backwards passes a neuron by getting amplified by the local gradient.

2. An error δ that needs to go through an affine transformation distributes itself in the way signal combined in forward pass.

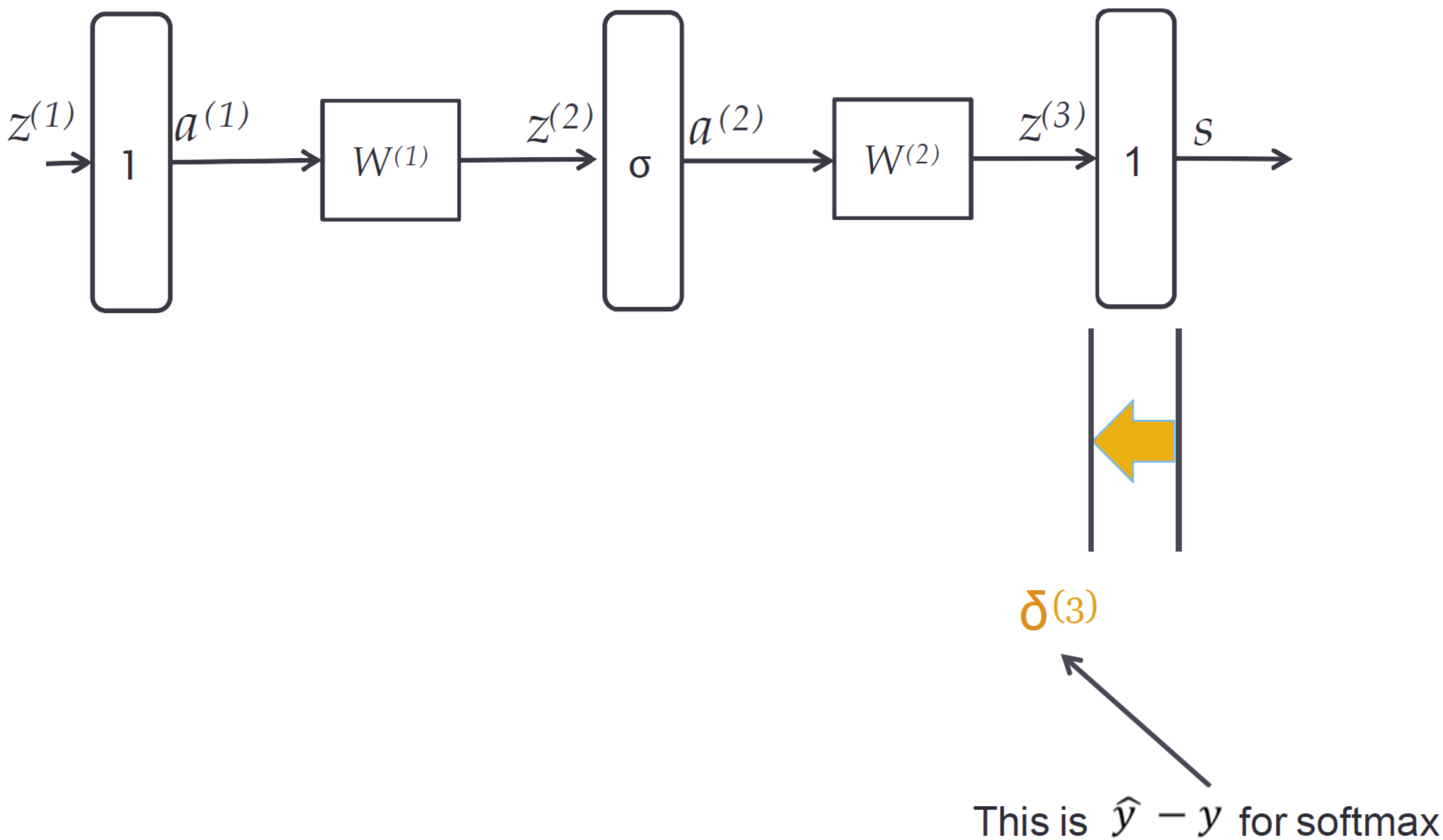


BackProp: Error Vectors

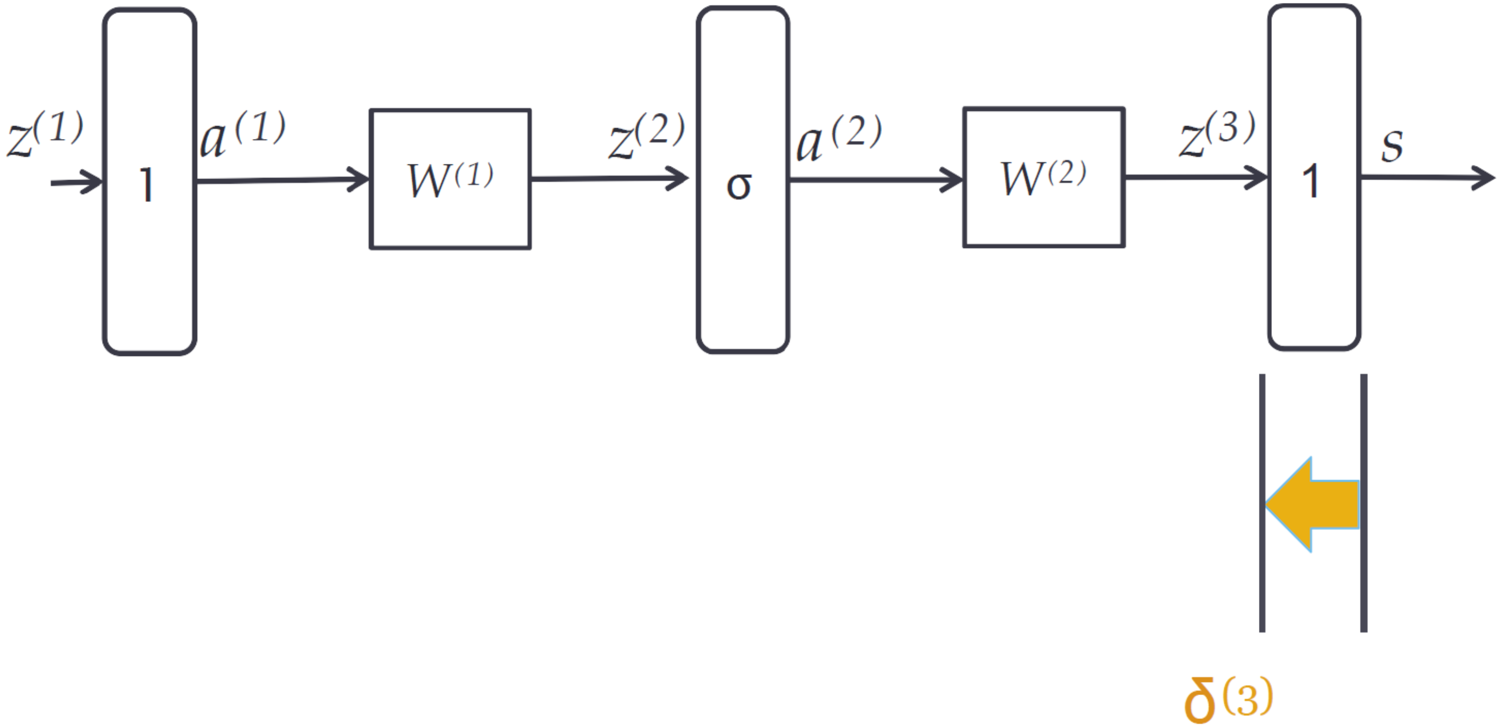




BackProp: blame propagation

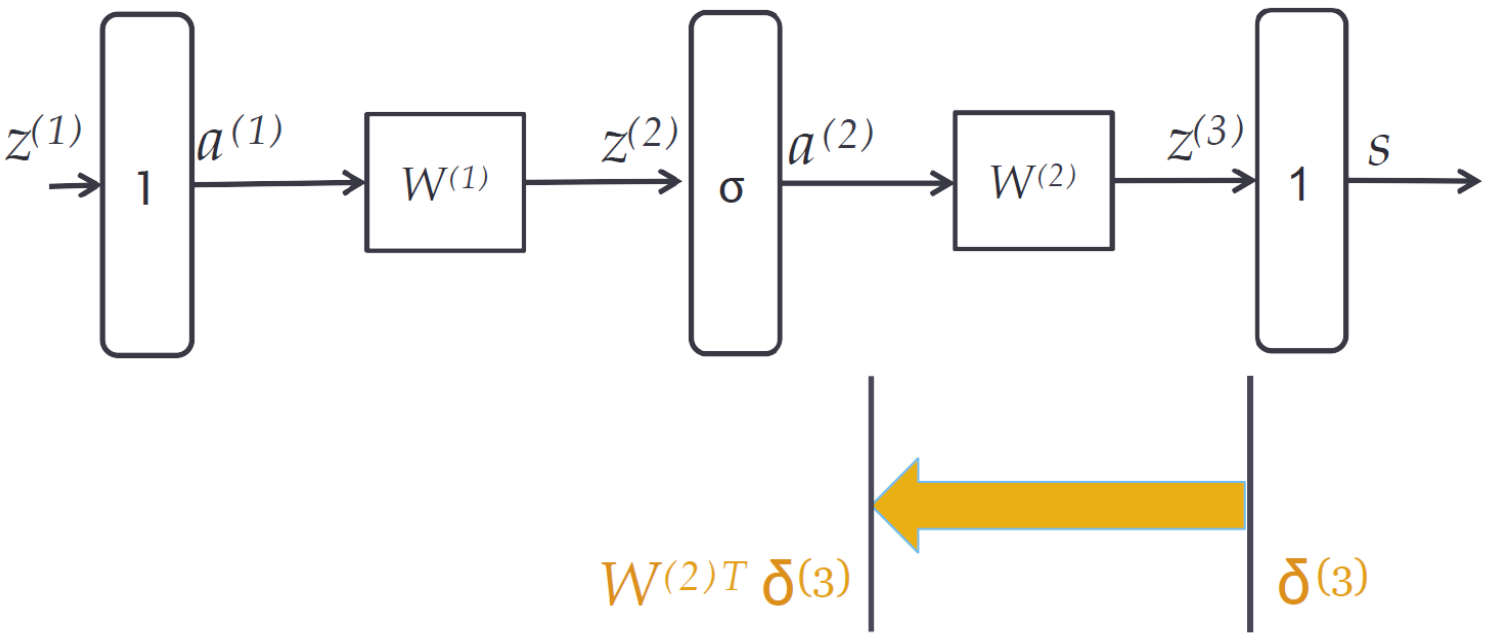


BackProp: gradient propagation



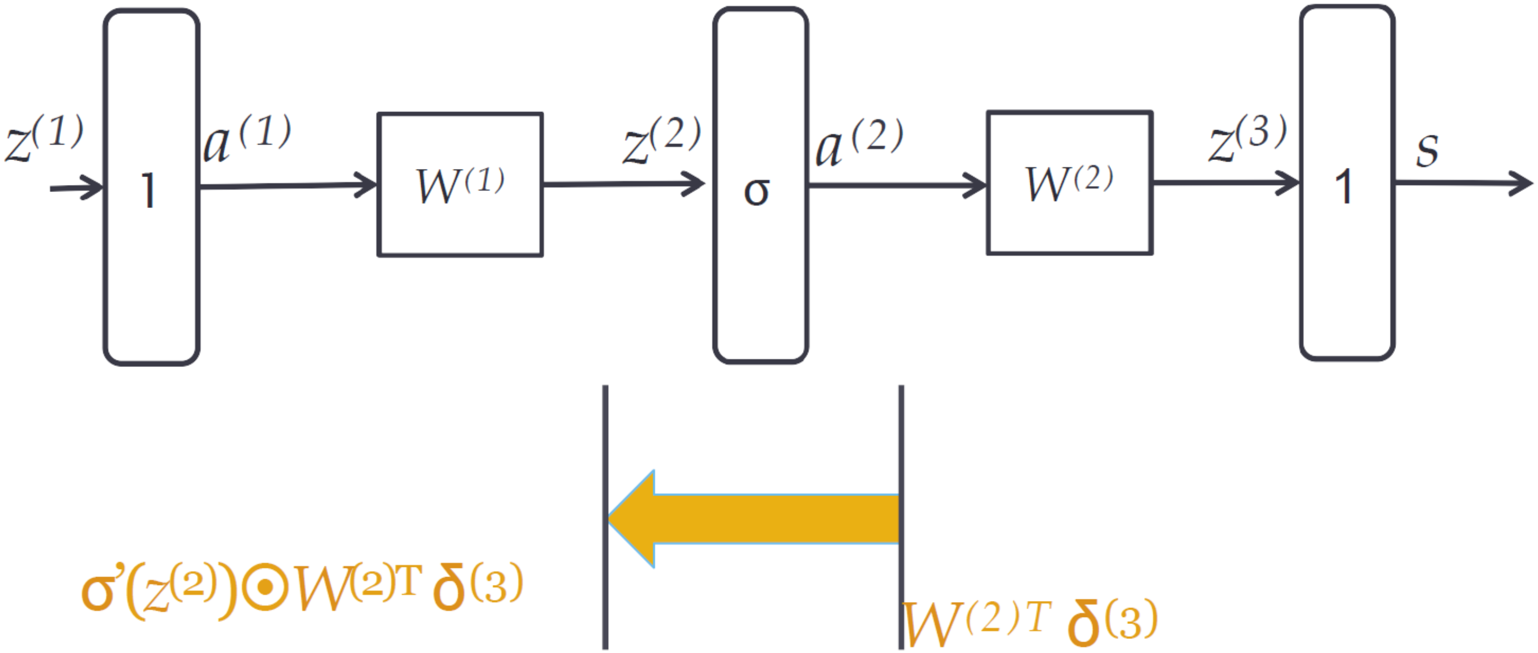
Gradient w.r.t $W^{(2)}$ = $\delta^{(3)} a^{(2)T}$

BackProp: blame propagation



--Reusing the $\delta^{(3)}$ for downstream updates.

BackProp: blame propagation



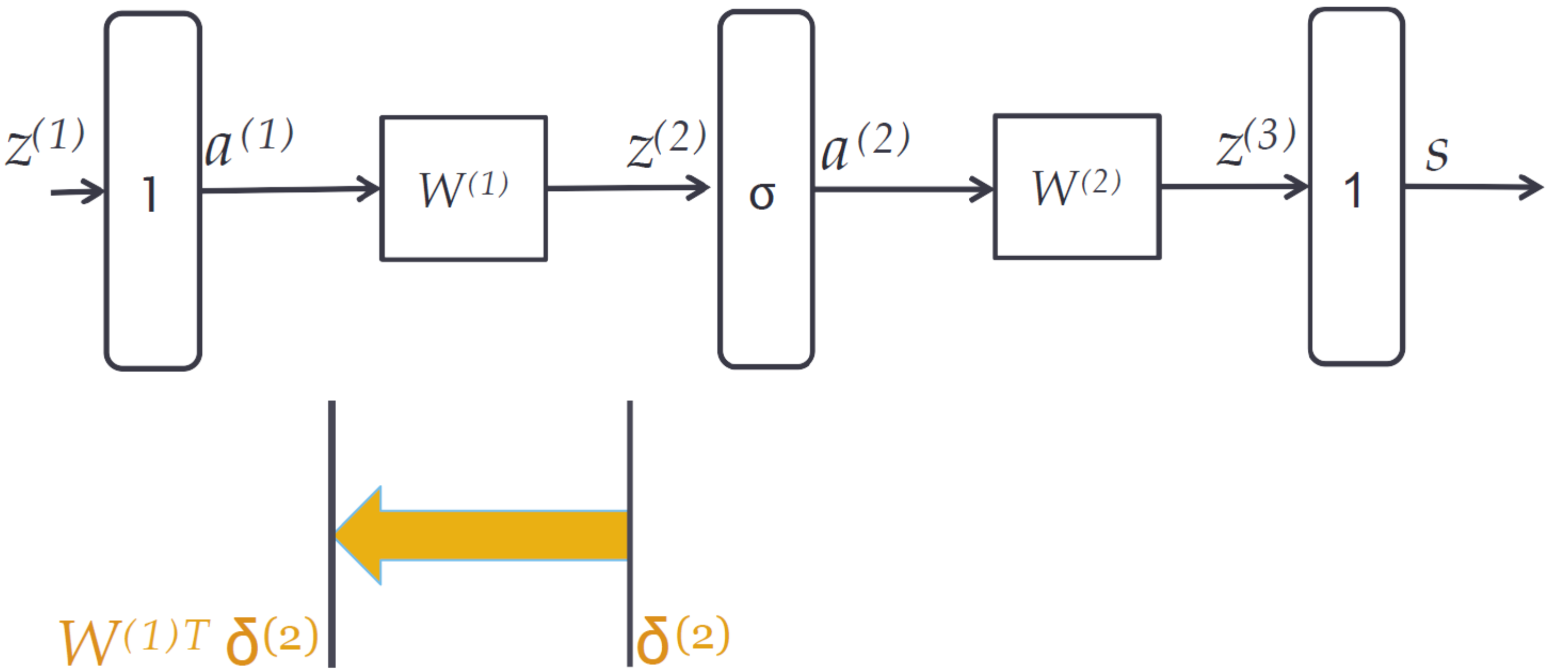
$W^{(2)T}\,\delta^{(3)}$

$\sigma'(z^{(2)}) \odot W^{(2)T}\,\delta^{(3)}$

$= \delta^{(2)}$

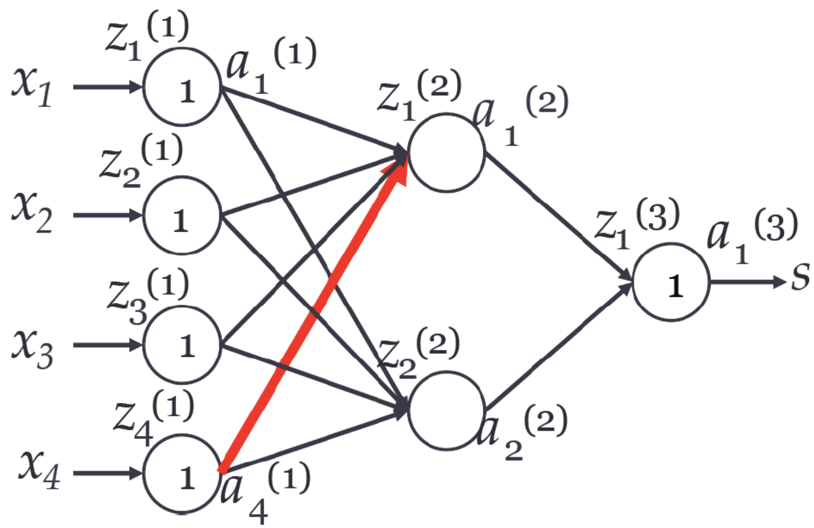
--Moving error vector across point-wise non-linearity requires point-wise multiplication with local gradient of the non-linearity

BackProp: blame propagation



Gradient w.r.t $W^{(1)}$ =
$\delta^{(2)} a^{(1)T}$

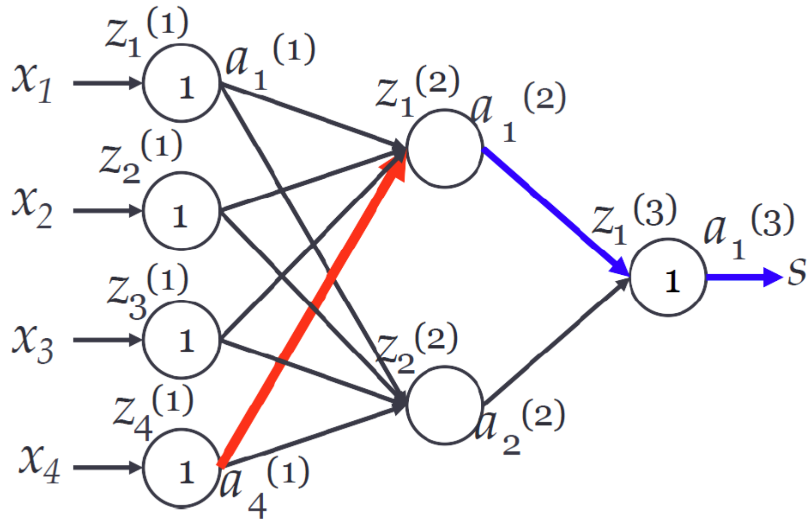
NN Training: Backpropagation



Let us try to calculate the error gradient wrt $W_{14}^{(1)}$
Thus we want to find:

$$\frac{\partial s}{\partial W_{14}^{(1)}}$$

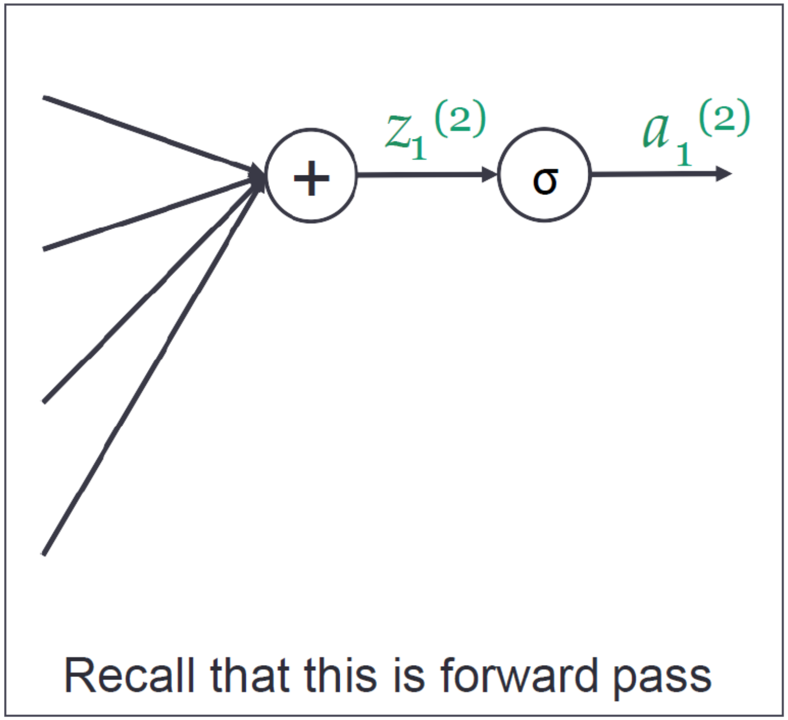
Backpropagation: Chain Rule

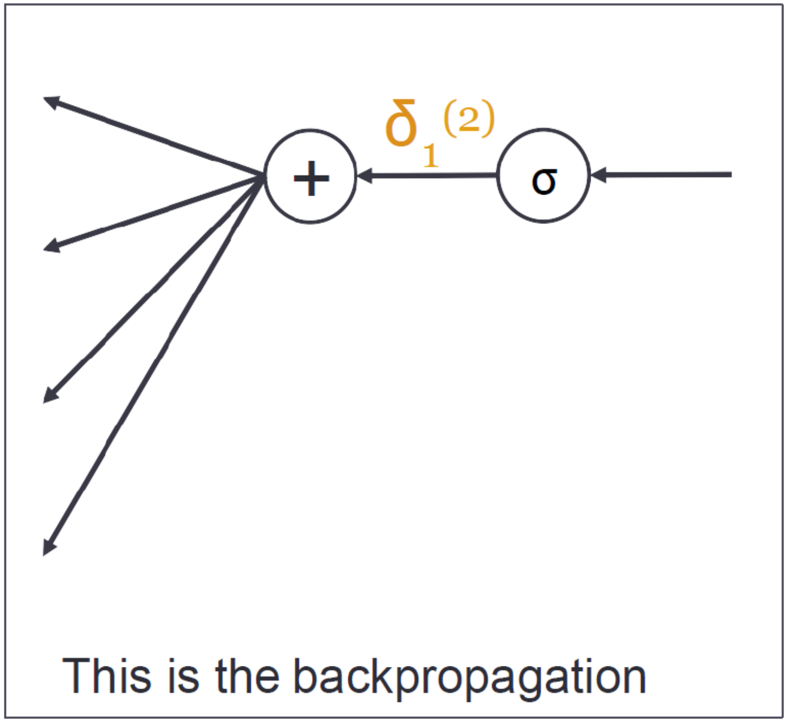


Let us try to calculate the error gradient wrt $W_{14}^{(1)}$
Thus we want to find:

$$\frac{\partial s}{\partial z_1^{(3)}} \frac{\partial z_1^{(3)}}{\partial a_1^{(2)}} \frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial W_{14}^{(1)}}$$

BackProp: Loss Gradient

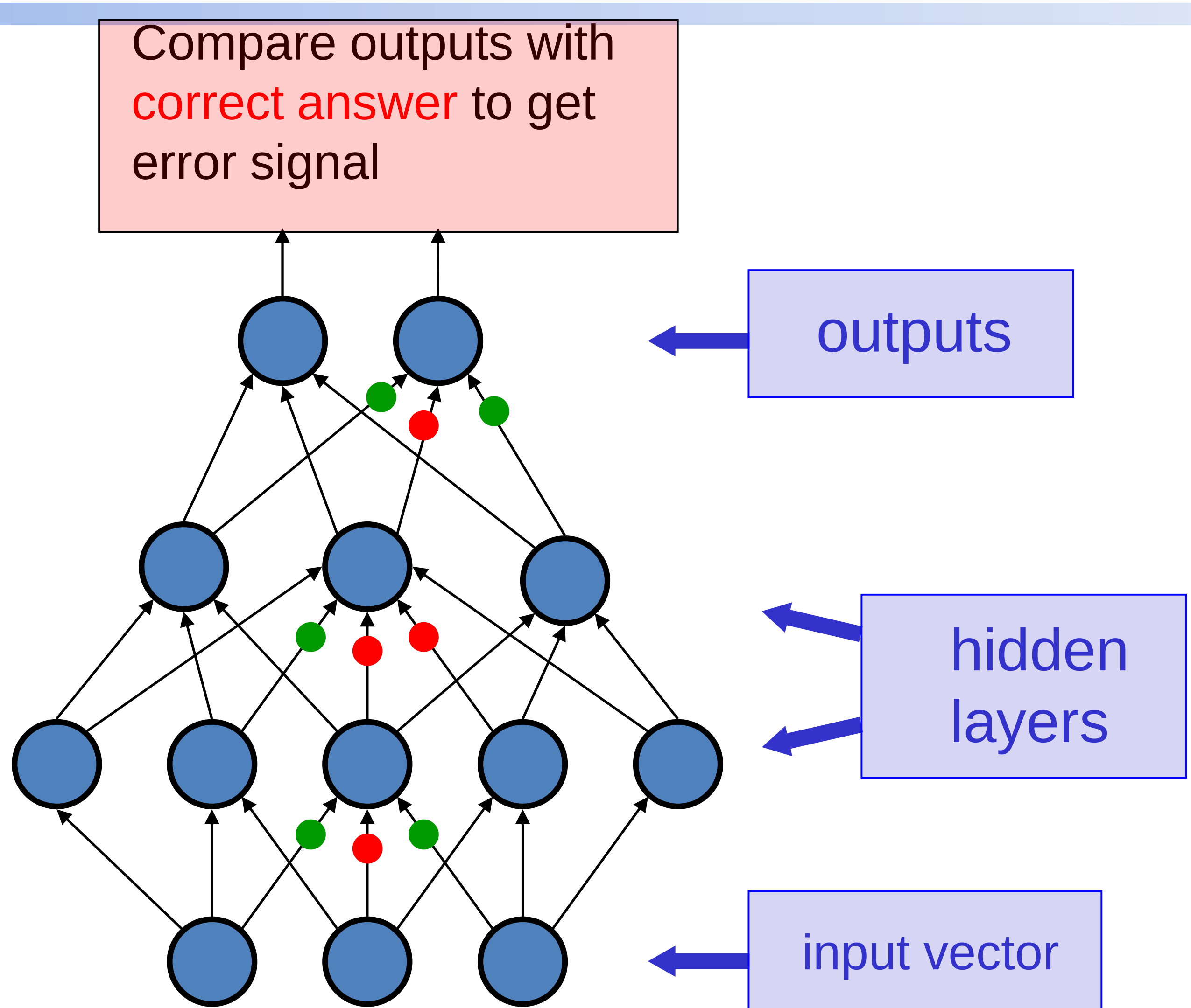




$\delta_1^{(2)}$ is the error flowing backwards at the same point where $z_1^{(2)}$ passed forwards. Thus it is simply the gradient of the error wrt $z_1^{(2)}$.

Backpropagation: Recap

Back-propagate error signal to get derivatives for learning



Backpropagation: Code and Demos

- http://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/
- Code: /home/eugene/cs325/inclass/04042016/
- Online demo:

  http://www.emergentmind.com/neural-network

- Exercise for the reader == Extra credit for P4:
  - Incorporate NN (with Backprop) as another classifier in Project 4 framework

How many hidden layers?

- Usually just one (i.e., a 2-layer net)

- How many hidden units in the layer?
  - Too few ==> can't learn
  - Too many ==> poor generalization

Overfitting

- Complexity = memorize training data!
- Solutions:
  - Simplify model
  - Enforce “small” weights for smoother surface:

  http://neuralnetworksanddeeplearning.com/chap3.html#overfitting_and_regularization