CS-540: INTRO TO ARTIFICIAL INTELLIGENCE

BACK PROPAGATION

Erin Winter

March 27th, 2017

Midterm Exam Grades

cannot be distributed because 2 students haven’t taken the midterm yet.
Neither can the answers for HW3 for the same reason.

Midterm Evaluation Feedback

"you should assign readings complementary to the lectures"

There have been readings this whole time on the Canvas Assignments page. Please do them.

"more examples of algorithms in lecture"

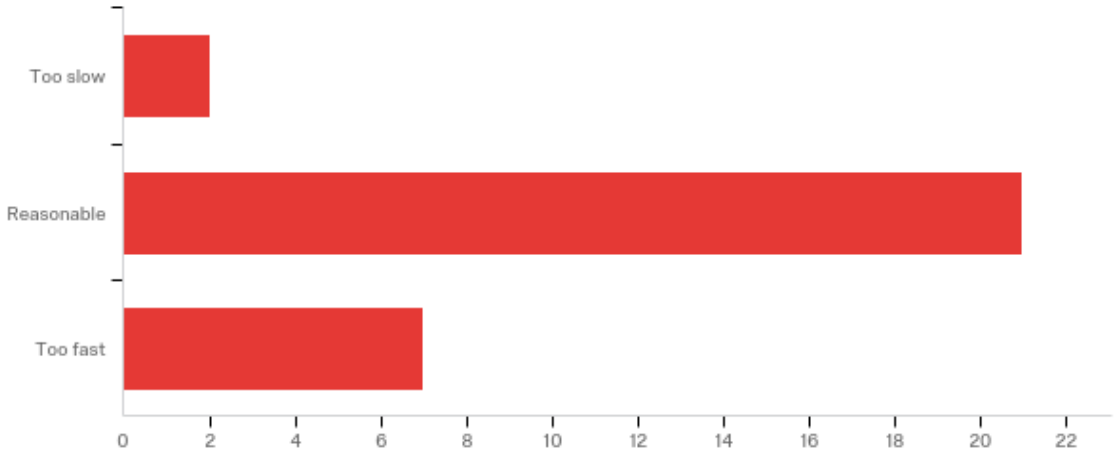
I can definitely try to do that, time and document camera permitting.

"faster grading"

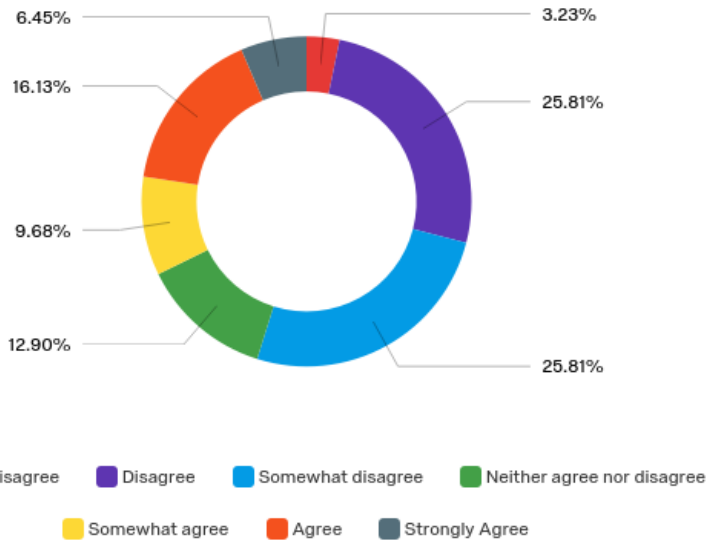
I would enjoy that too, but that is dependent on the TA's, not me. I'm working with them to make turnaround time faster.

Midterm Evaluation Feedback

Pacing
Pacing will be roughly the same, if a bit slower.

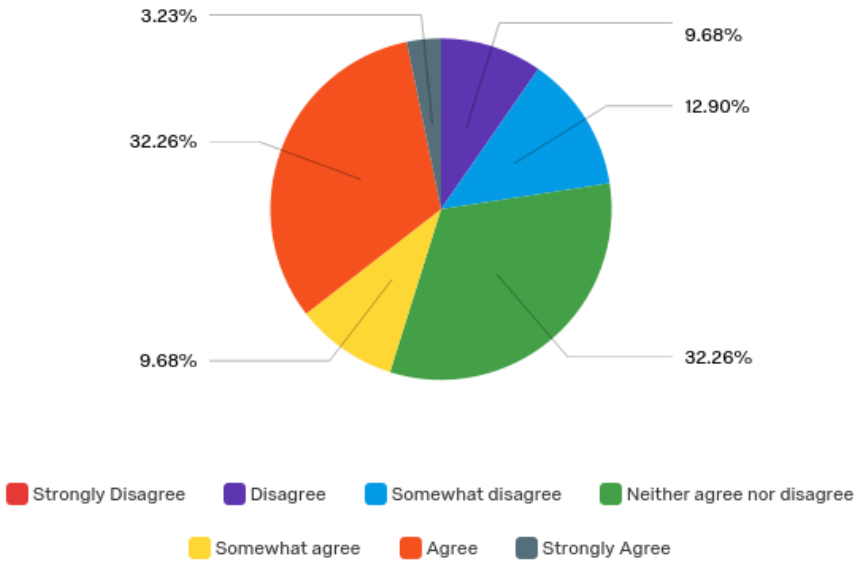


Short assignments vs. long assignments
About evenly split, so I'll keep them roughly the same

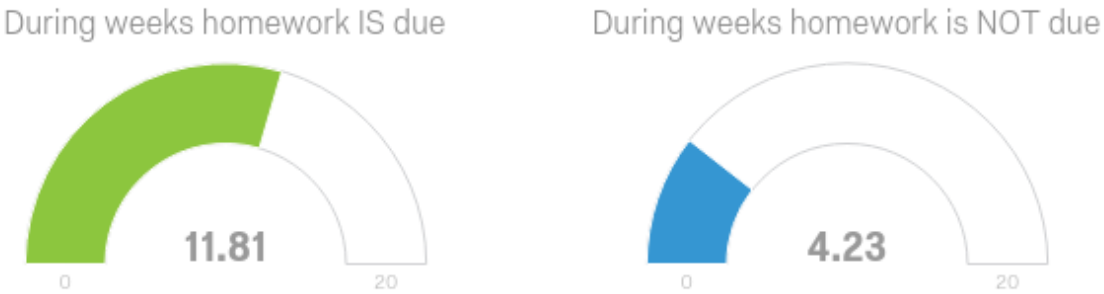


Midterm Evaluation Feedback

I'm comfortable going to office hours. Please come if you have questions. There's nothing to be afraid of.

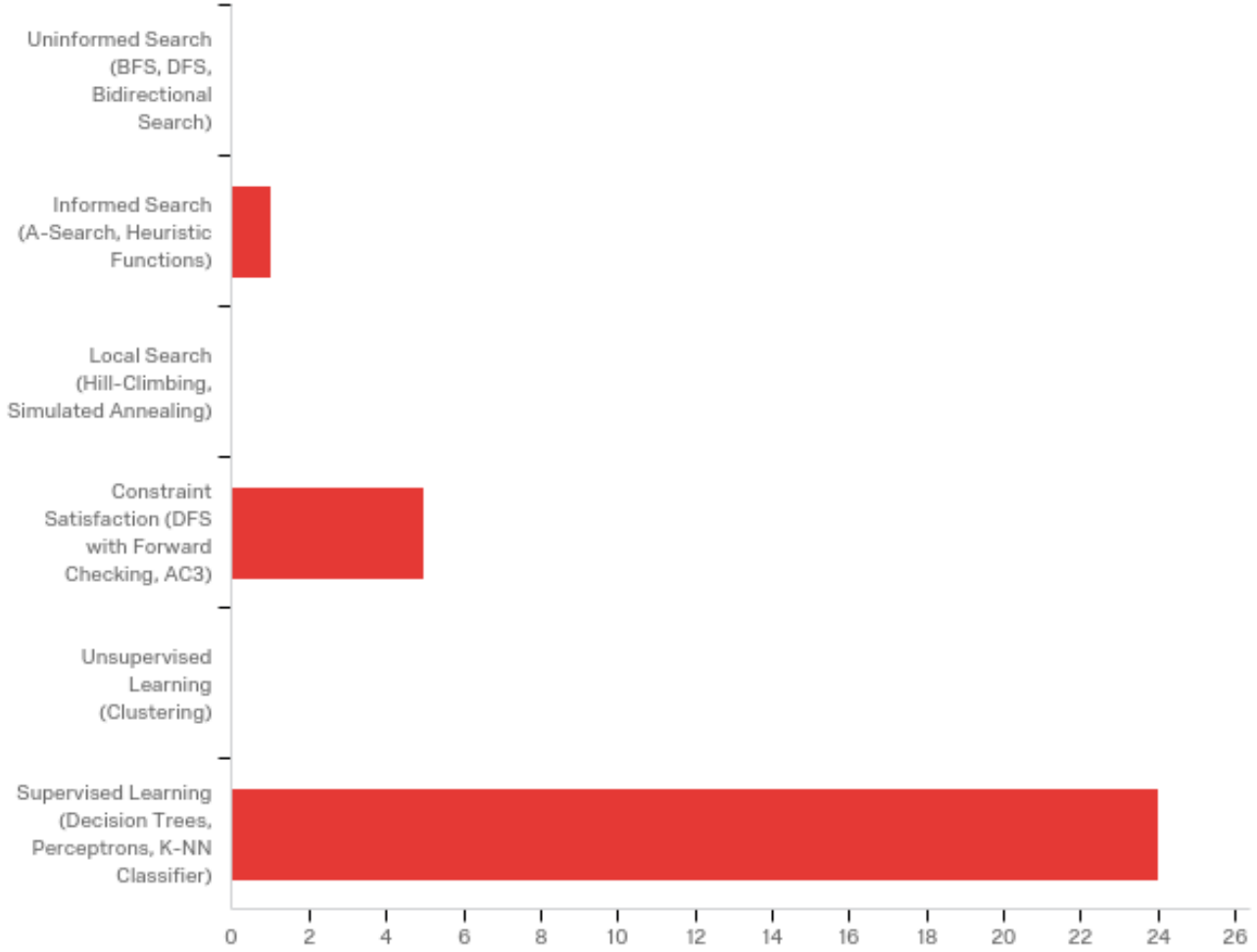


How many hours per week do you spend on this class? About what I expected.



Midterm Evaluation Feedback

Most difficult topic?



BACK-PROPAGATION ALGORITHM

Initialize the weights in the network (usually random values)

Repeat until stopping criterion is met **{**

forall p,q in network, $\Delta \boldsymbol{W}_{p,q} = 0$

foreach example ***e*** in training set **do {**

O = neural_net_output(***network***, ***e***) // forward pass

Calculate error (***T*** - **O**) at the output units // ***T*** = teacher output

Compute $\Delta w_{i,k}$ for all weights from hidden to output layer

Compute $\Delta w_{i,j}$ for all weights from inputs to hidden layer

forall p,q in network $\Delta \boldsymbol{W}_{p,q} = \Delta \boldsymbol{W}_{p,q} + \Delta w_{\boldsymbol{p,q}}$

}

for all p,q in network $\Delta \boldsymbol{W}_{p,q} = \Delta \boldsymbol{W}_{p,q}$ / num_training_examples

network = update_weights(***network***, $\Delta \boldsymbol{W}_{p,q}$)

}

backward pass

BACK-PROP USING STOCHASTIC GRADIENT DESCENT (SGD)

Most practitioners use SGD to update weights using the *average gradient computed using a small batch of examples*, and repeating this process for many small batches from the training set

In extreme case, update after *each* example

Called *stochastic* because each small set of examples gives a noisy estimate of the average gradient over *all* training examples

UPDATING THE WEIGHTS

Back-Propagation performs a ***gradient descent search*** in "weight space" to learn the network weights

Given a network with *n* weights:

- each configuration of weights is a vector, ***W***, of length *n* that defines an instance of the network
- ***W*** can be considered a point in an *n*-dimensional **weight space**, where each dimension is associated with one of the connections in the network

UPDATING THE WEIGHTS

- Given a training set of *m* examples:
  - Each network defined by the vector ***W*** has an associated total error, ***E***, on ***all*** the training data
  - ***E*** is the sum squared error (SSE) defined by

$$E = E_1 + E_2 + \ldots + E_m$$

where E_i is the squared error of the network on the i^{th} training example

- Given *n* output units in the network:

$$E_i = (T_1 - O_1)^2 + (T_2 - O_2)^2 + \ldots + (T_n - O_n)^2$$

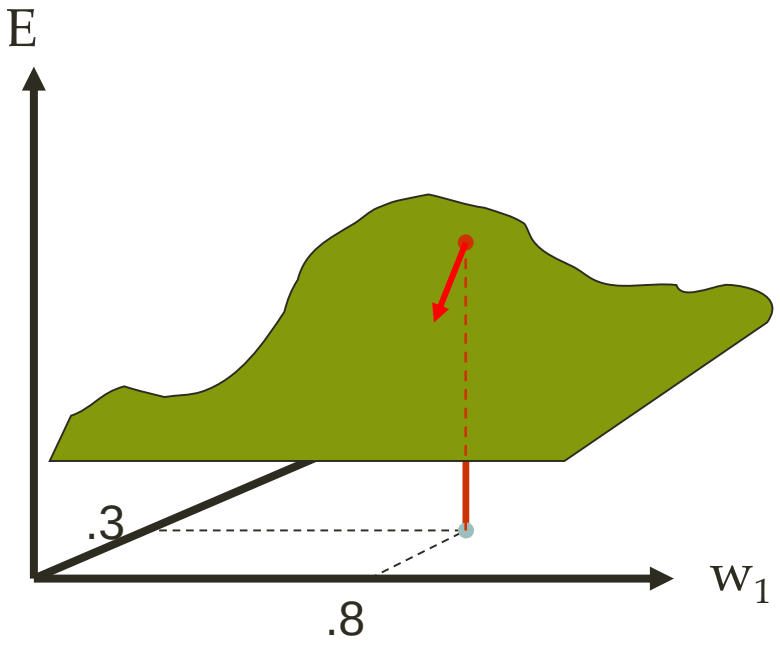
  - T_i is the target value for the i^{th} example
  - O_i is the network output value for the i^{th} example

UPDATING THE WEIGHTS

Visualized as a 2D error surface in "weight space"

- Each point in w_1 w_2 plane is a weight configuration

- Each point has a total error E
- 2D surface represents errors for all weight configurations
- Goal is to find a lower point on the error surface (local minimum)
- Gradient descent follows the direction of steepest descent, i.e., where E decreases the most



UPDATING THE WEIGHTS

The **gradient** is defined as

$$\nabla E = [\partial E / \partial w_1, \partial E / \partial w_2, ..., \partial E / \partial w_n]$$

Update the ith weight using

$$\Delta w_i = - \alpha \, \partial E / \partial w_i$$

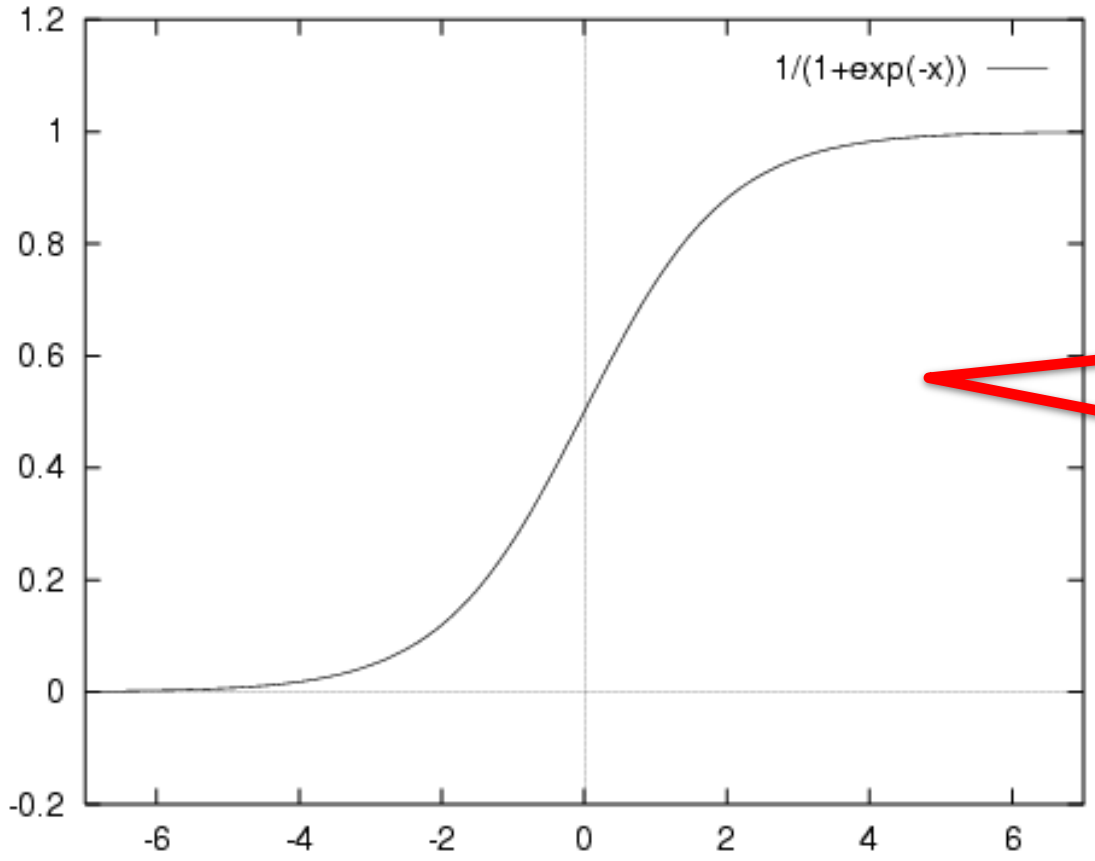
Can't use the Step function in LTU's because it's derivative is 0 everywhere

Instead, let's use (for now) the **Sigmoid function**

Sigmoid Activation Function

Solution: Replace with a smooth function such as Sigmoid function (aka Logistic Sigmoid function):

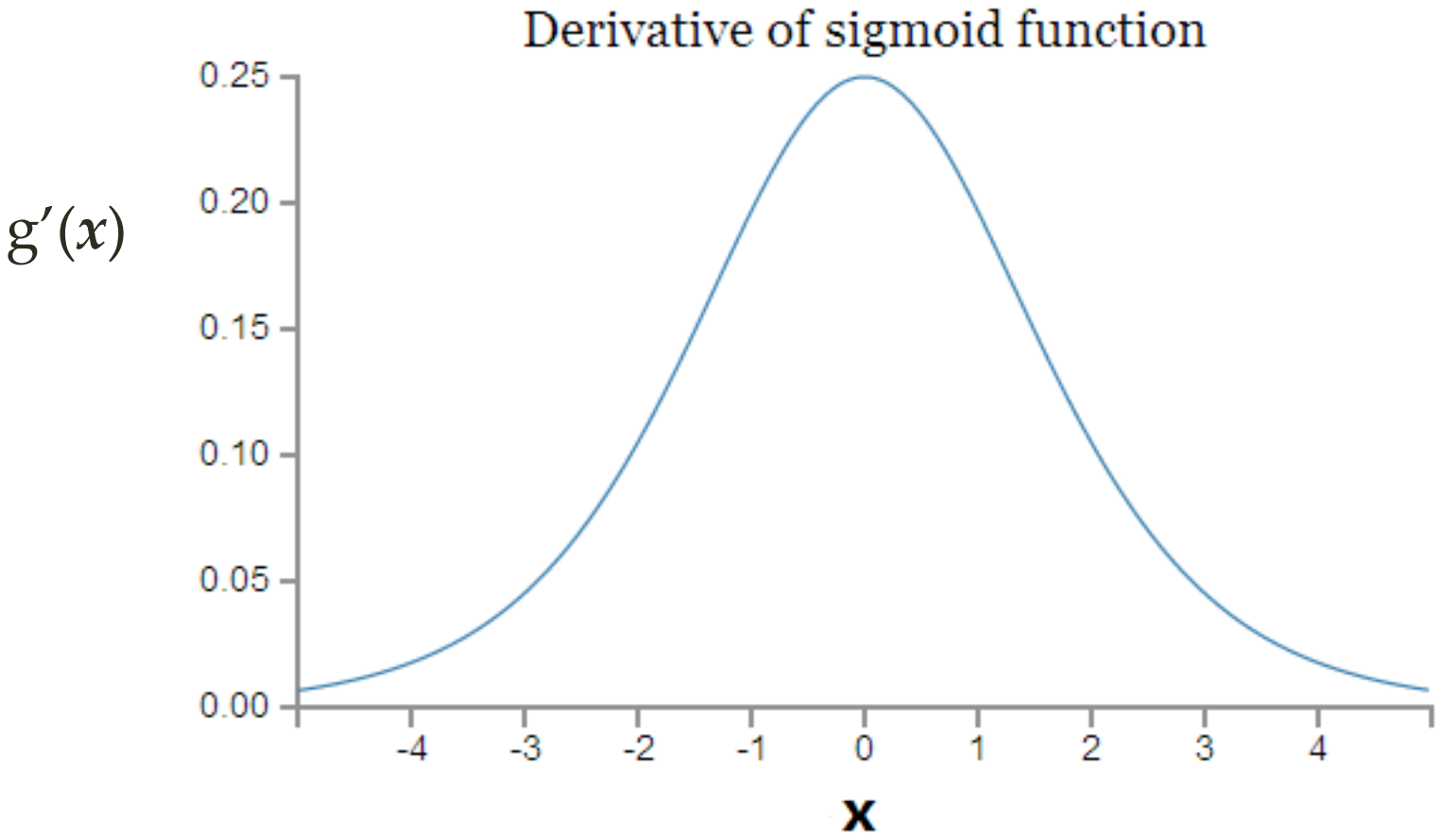
$g_w(x) = 1 / (1 + e^{-wx})$



Squashes numbers to range [0,1]

FIRST DERIVATIVE OF SIGMOID FUNCTION

$$g'(x) = g(x)\,(1 - g(x))$$



UPDATING WEIGHTS IN A 2-LAYER NEURAL NETWORK

For **weights between hidden and output units**, generalized PLR for sigmoid activation is

$$\Delta w_{j,k} = -\alpha \frac{\partial E}{\partial w_{j,k}}$$
$$= -\alpha \, -a_j (T_k - O_k) \, g'(in_k)$$
$$= \alpha \, a_j (T_k - O_k) \, O_k (1 - O_k)$$
$$= \alpha \, a_j \, \Delta_k$$

$\Delta_k = \mathrm{Err}_k \times g'(\mathrm{in}_k)$

- $w_{j,k}$ weight on link from hidden unit j to output unit k
- α learning rate parameter
- a_j activation (i.e. output) of hidden unit j
- T_k teacher output for output unit k
- O_k actual output of output unit k
- g' derivative of the sigmoid activation function, which is $g' = g(1 - g)$

UPDATING WEIGHTS IN A 2-LAYER NEURAL NETWORK

For **weights between inputs and hidden units**:

$$\Delta w_{i,j} = -\alpha \, \partial E / \partial w_{i,j}$$

$$= -\alpha(-a_i) g'(in_j) \sum_k w_{j,k}(T_k - O_k) g'(in_k)$$

$$= \alpha \, a_i \, a_j(1 - a_j) \sum_k w_{j,k}(T_k - O_k) \, O_k(1 - O_k)$$

$$= \alpha \, a_i \, \Delta_j \quad \text{where} \quad \Delta_j = g'(in_j) \sum_k w_{j,k} \Delta_k$$

$w_{i,j}$ weight on link from input i to hidden unit j

$w_{j,k}$ weight on link from hidden unit j to output unit k

α learning rate parameter

a_j activation (i.e. output) of hidden unit j

T_k teacher output for output unit k

O_k actual output of output unit k

a_i input value i

g' derivative of sigmoid activation function, which is $g' = g(1-g)$

BACK-PROPAGATION ALGORITHM

Initialize the weights in the network (usually random values)

Repeat until stopping criterion is met

foreach example, *e,* in training set **do**

{ O = neural_net_output(network, *e*)

T = desired output, i.e., **T**arget or **T**eacher's output

calculate error (T - O) at all the output units

compute $\alpha w_{j,k} = \alpha\, a_j\, \Delta_k = \alpha\, a_j\, (T_k - O_k)\, g'(in_k)$

compute $\Delta w_{i,j} = \alpha\, a_i\, \Delta_j = \alpha\, a_i\, g'(in_j) \sum w_{j,k} (T_k - O_k) g'(in_k)$

forall *p*, *q* in network $w_{p,q} = w_{p,q} + \Delta w_{p,q}$

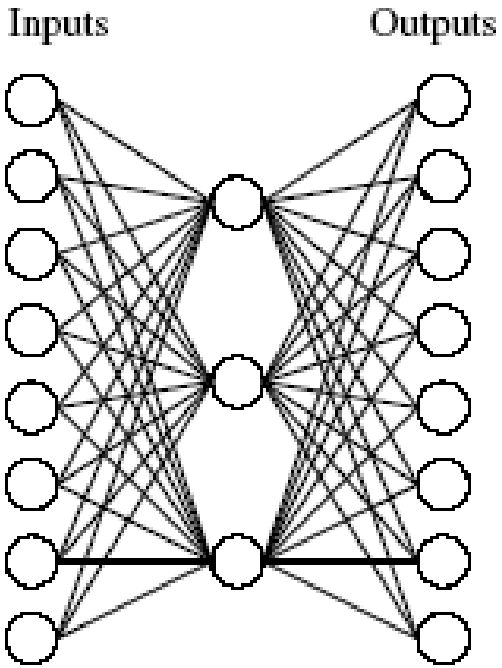
}

backward pass

forward pass

Simplistic SGD: update all weights after each example

LEARNING HIDDEN LAYER REPRESENTATION

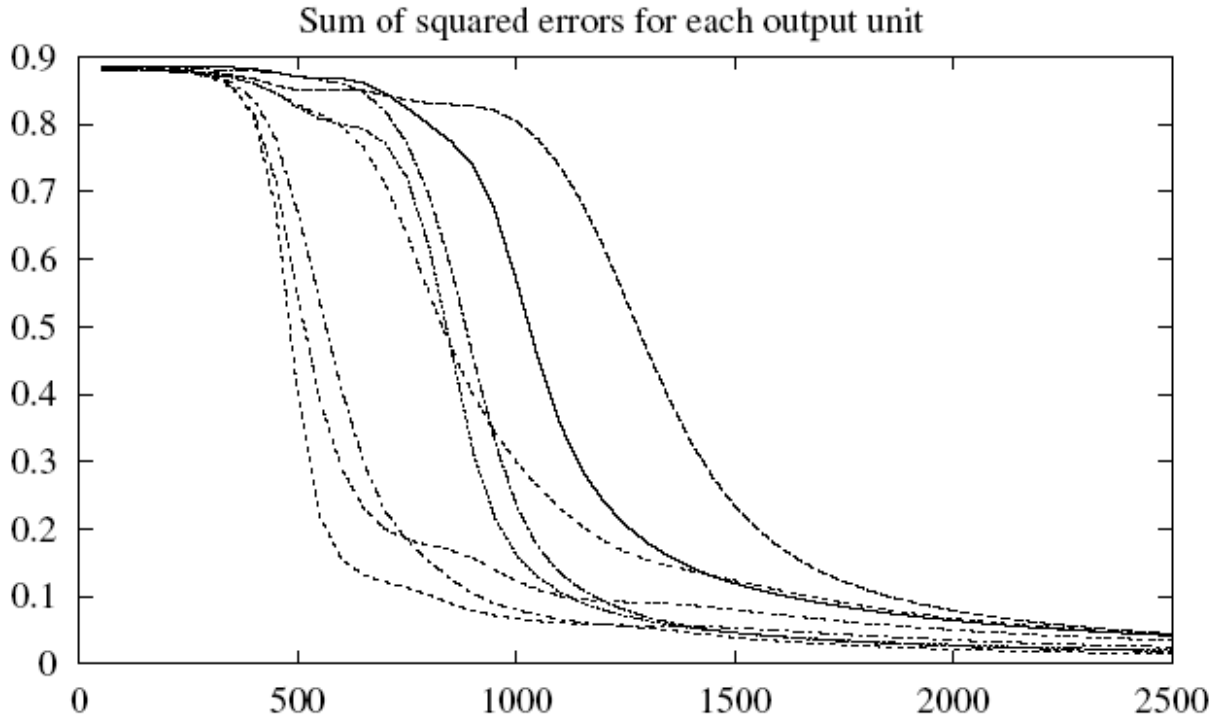


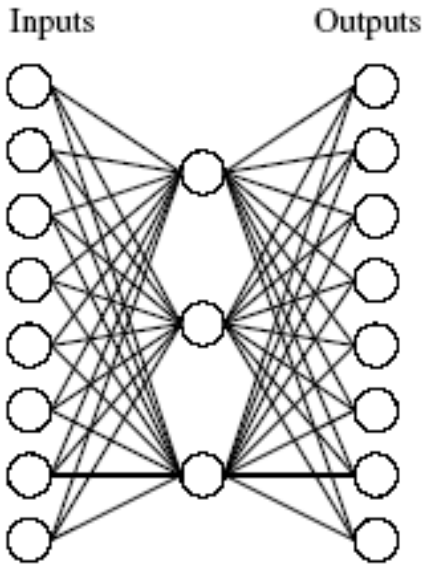
Input		Output
10000000	→	10000000
01000000	→	01000000
00100000	→	00100000
00010000	→	00010000
00001000	→	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	00000001

Can this be learned?

Slide by Guoping Qiu

LEARNING HIDDEN LAYER REPRESENTATION

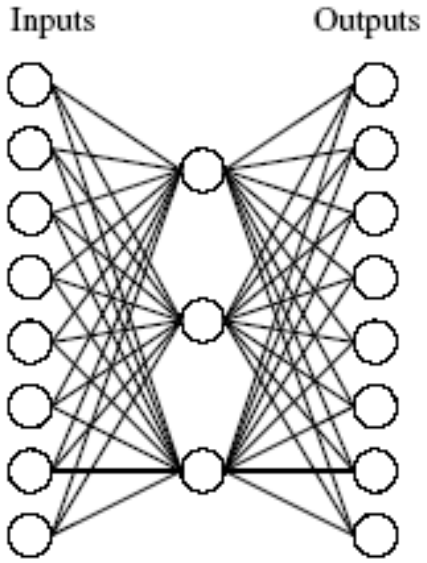




The evolving sum of squared errors for each of the eight output units

Slide by Guoping Qiu

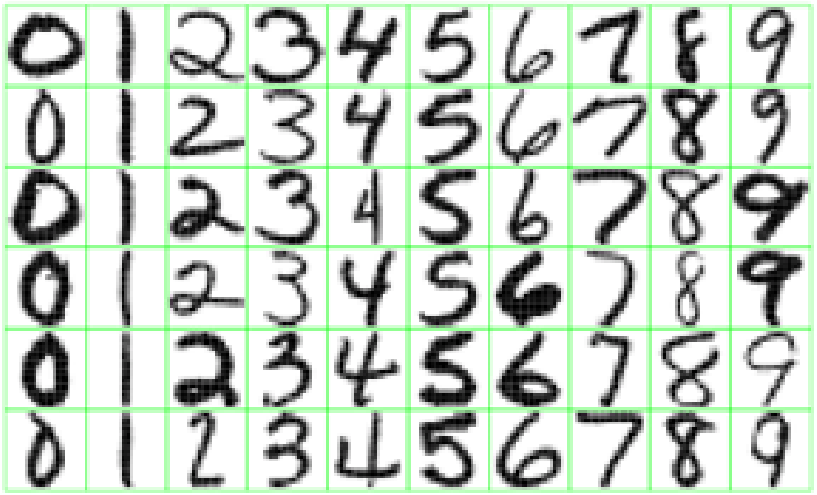
LEARNING HIDDEN LAYER REPRESENTATION



Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

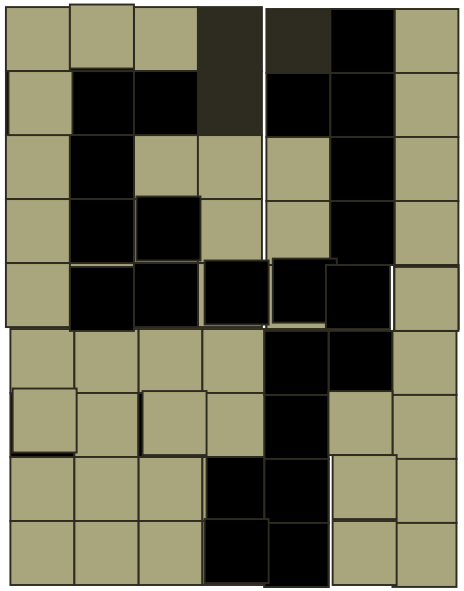
Learned hidden layer representation

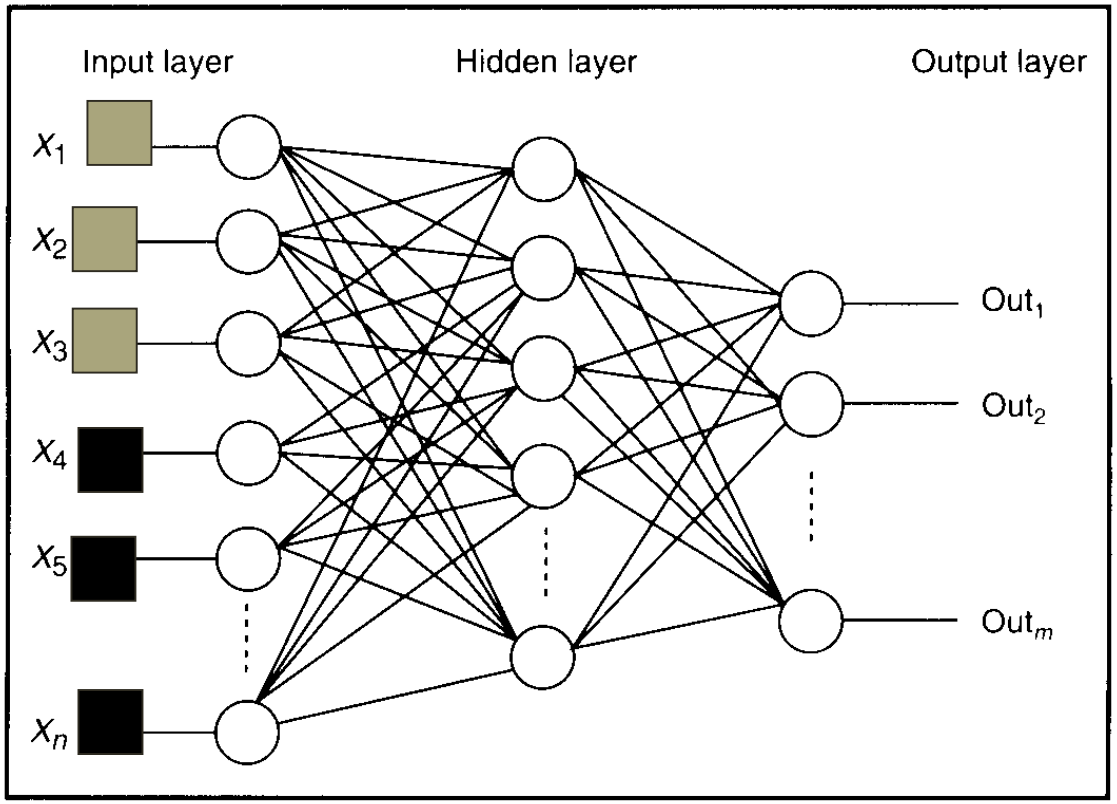
Slide by Guoping Qiu



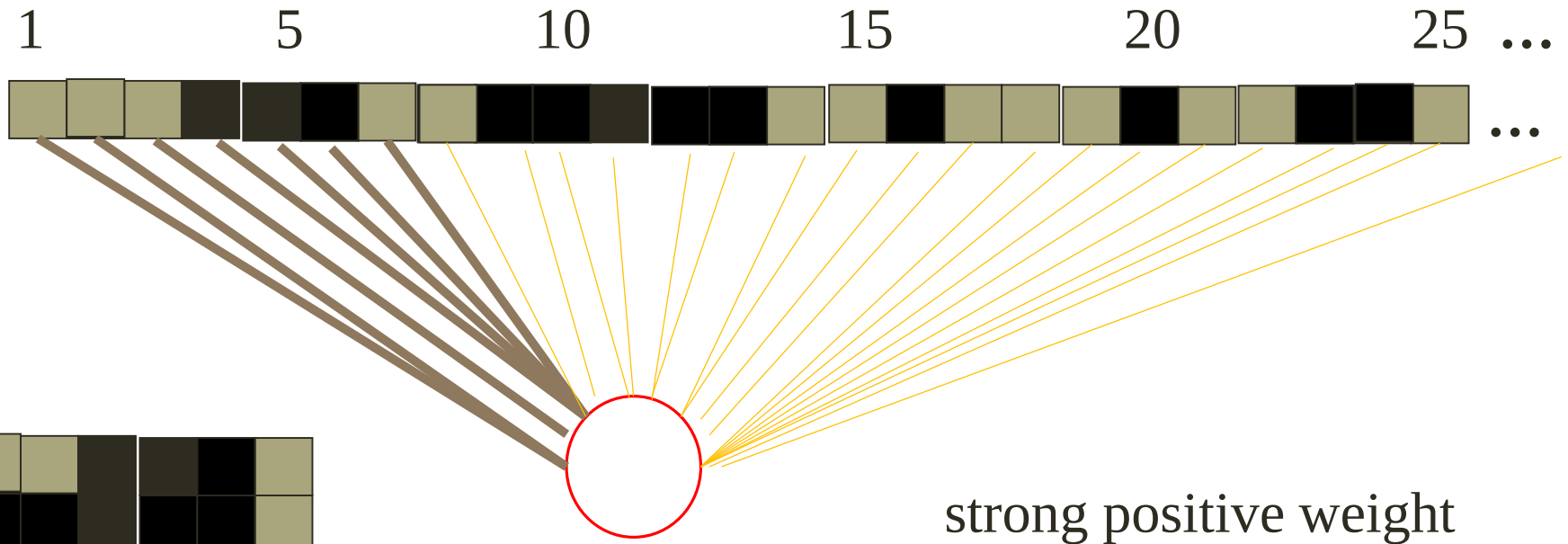
FEATURE DETECTORS

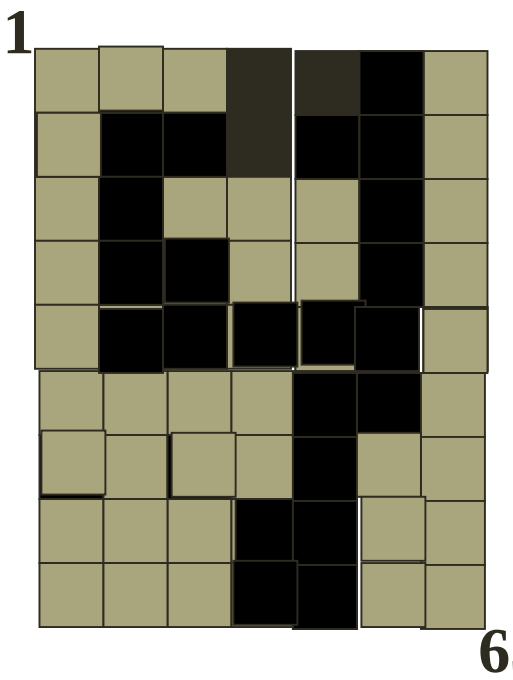
Figure 1.2: *Examples of handwritten digits from postal envelopes.*





HIDDEN-LAYER UNITS LEARN TO BE FEATURE DETECTORS

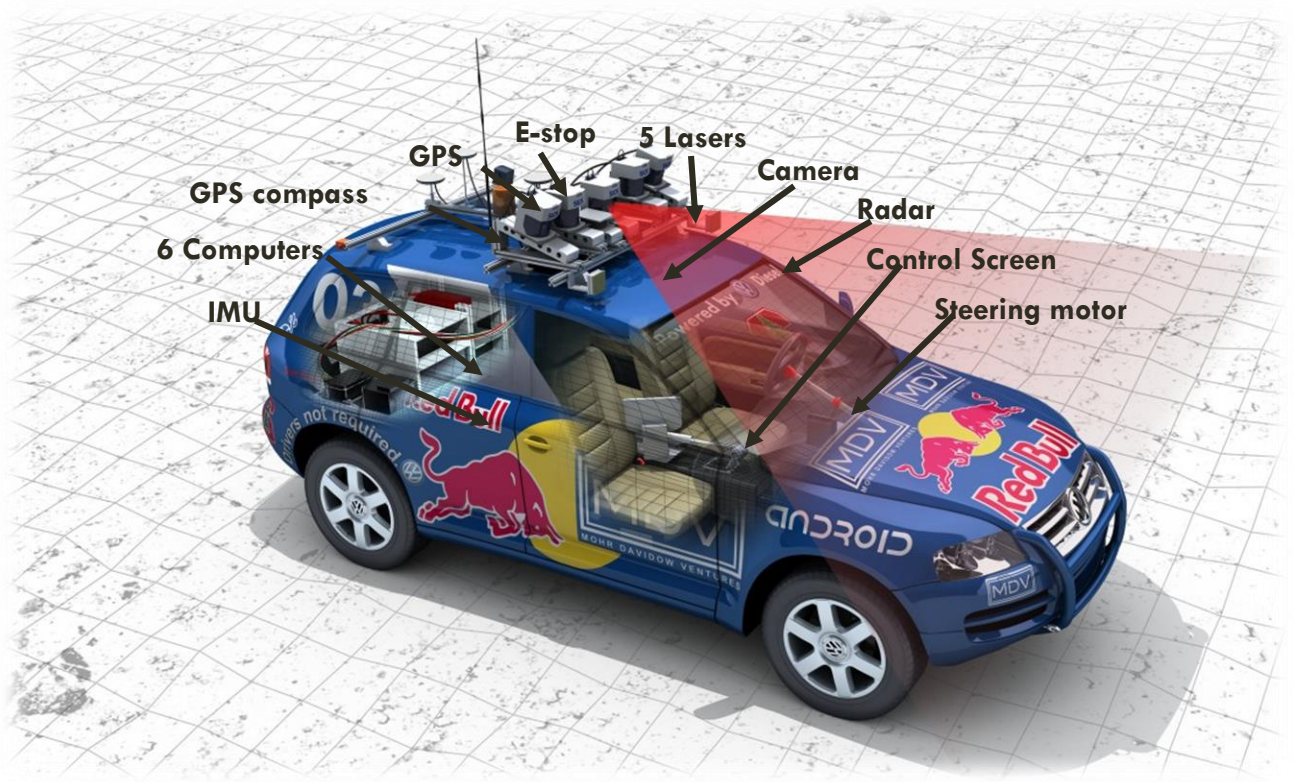




strong positive weight

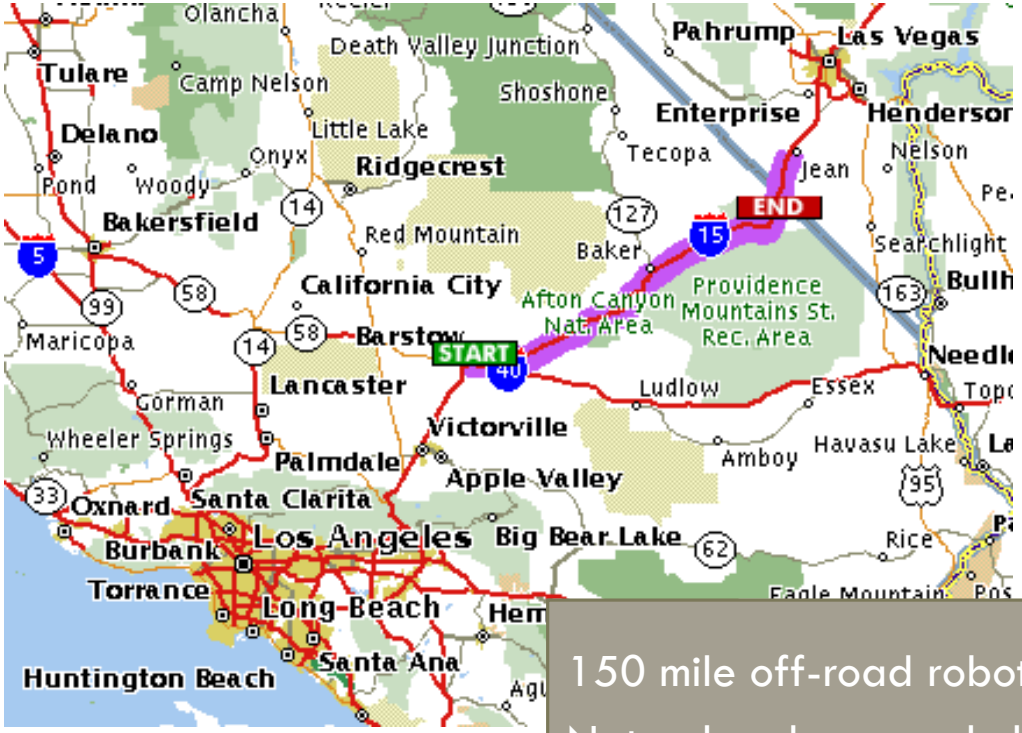
low/zero weight

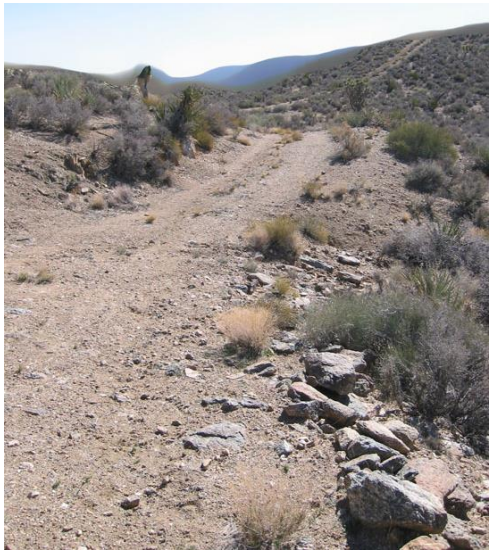
AUTONOMOUS VEHICLES



Slides courtesy of UC Berkeley

GRAND CHALLENGE 2005: BARSTOW, CA, TO PRIMM, NV





- 150 mile off-road robot race across the Mojave desert
- Natural and manmade hazards
- No driver, no remote control
- No dynamic passing

GRAND CHALLENGE 2005

[VIDEO: nova-race-supershort.mp4]