BACKPROPAGATION

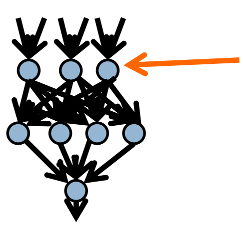
David Kauchak
CS159 – Fall 2019

Admin

Assignment 5

Neural network

inputs

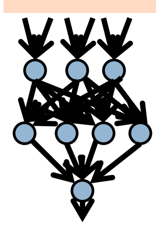


Individual perceptrons/neurons

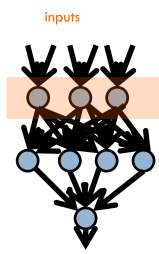
Neural network

inputs

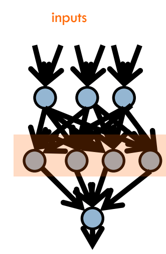
some inputs are provided/entered



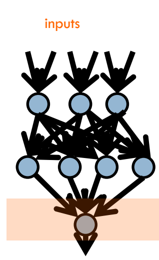
Neural network



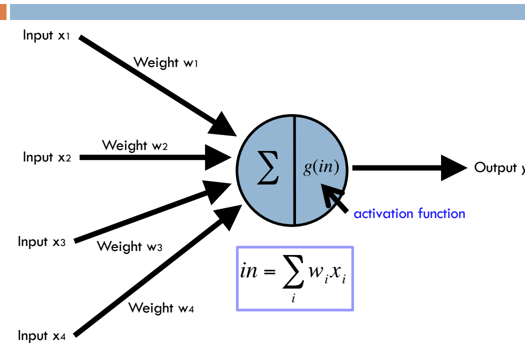
Neural network



Neural network



A neuron/perceptron

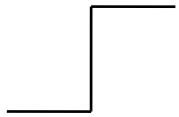


$$in = \sum_i w_i x_i$$

Activation functions

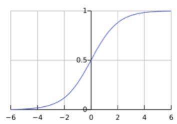
hard threshold:

$$g(in) = \begin{cases} 1 & if\ in \geq T \\ 0 & otherwise \end{cases}$$

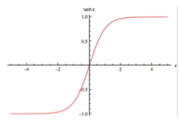


sigmoid

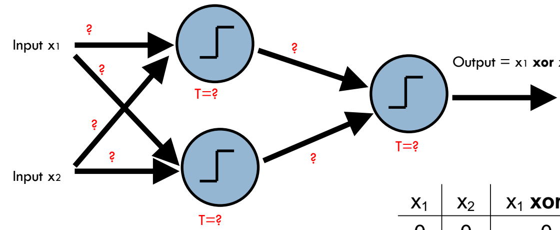
$$g(x) = \frac{1}{1+e^{-x}}$$



tanh x



Training

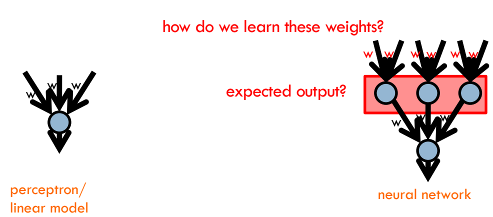


x_1	x_2	x_1 **xor** x_2
0	0	0
0	1	1
1	0	1
1	1	0

How do we learn the weights?

Learning in multilayer networks

Challenge: for multilayer networks, we don't know what the expected output/error is for the internal nodes!

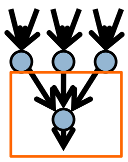


Backpropagation: intuition

Gradient descent method for learning weights by optimizing a loss function

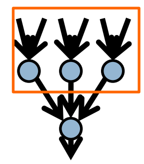
1. calculate output of all nodes
2. calculate the weights for the output layer based on the error
3. "backpropagate" errors through hidden layers

Backpropagation: intuition



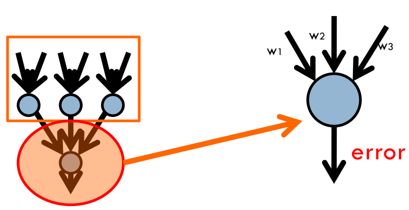
We can calculate the actual error here

Backpropagation: intuition



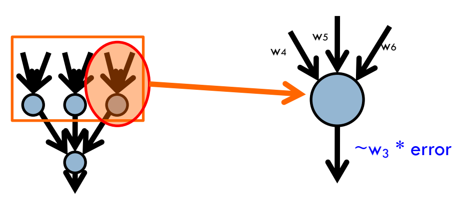
Key idea: propagate the error back to this layer

Backpropagation: intuition



error for node is ~ w_i * **error**

Backpropagation: intuition



Calculate as normal, but weight the error

Backpropagation: the details

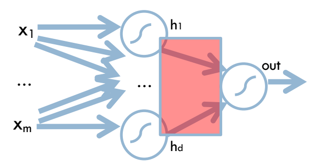
Gradient descent method for learning weights by optimizing a loss function

1. calculate output of all nodes
2. calculate the updates directly for the output layer
3. "backpropagate" errors through hidden layers

$$loss = \sum_x \frac{1}{2}(y - \hat{y})^2$$ squared error

Backpropagation: the details

Notation:



How many weights?

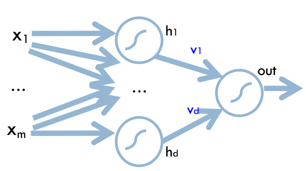
m: features/inputs

d: hidden nodes

hj: output from hidden nodes

Backpropagation: the details

Notation:



d weights: denote v_k

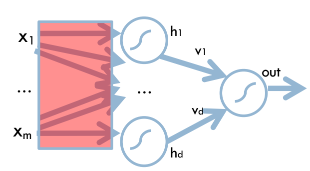
m: features/inputs

d: hidden nodes

hj: output from hidden nodes

Backpropagation: the details

Notation:



How many weights?

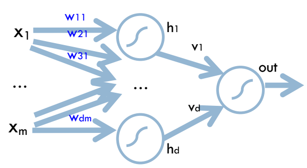
m: features/inputs

d: hidden nodes

hj: output from hidden nodes

Backpropagation: the details

Notation:



m: features/inputs

d: hidden nodes

h_k: output from hidden nodes

d * m: denote w_{kj}

first index = hidden node
second index = feature

- w_{23}: weight from input 3 to hidden node 2
- w_4: all the *m* weights associated with hidden node 4

Backpropagation: the details

Gradient descent method for learning weights by optimizing a loss function

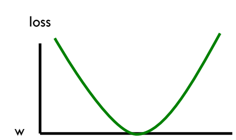
$$\text{argmin}_{w,v} \sum_x \frac{1}{2}(y-\hat{y})^2$$

1. calculate output of all nodes
2. calculate the updates directly for the output layer
3. “backpropagate” errors through hidden layers

Finding the minimum

You’re blindfolded, but you can see out of the bottom of the blindfold to the ground right by your feet. I drop you off somewhere and tell you that you’re in a convex shaped valley and escape is at the bottom/minimum. How do you get out?

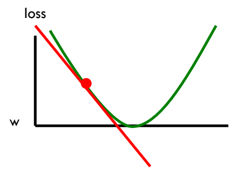
Finding the minimum



How do we do this for a function?

One approach: gradient descent

Partial derivatives give us the slope (i.e. direction to move) in that dimension

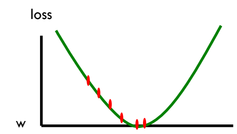


One approach: gradient descent

Partial derivatives give us the slope (i.e. direction to move) in that dimension

Approach:

- pick a starting point (w)
- repeat:
  - move a small amount towards decreasing loss (using the derivative)

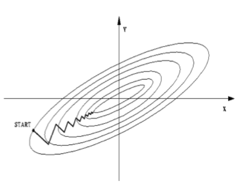


One approach: gradient descent

Partial derivatives give us the slope (i.e. direction to move) in that dimension

Approach:

- pick a starting point (w)
- repeat:
  - pick a dimension
  - move a small amount in that dimension towards decreasing loss (using the derivative)



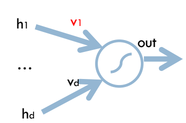
Output layer weights

$$v_k = v_k + (y - f(v \cdot h)) f'(v \cdot h) h_k$$

how far from correct and which direction

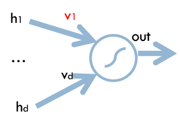
slope of the activation function where input is at

size and direction of the feature associated with this weight



Output layer weights

$$v_k = v_k + (y - f(v \cdot h)) f'(v \cdot h) h_k$$



how far from correct and which direction

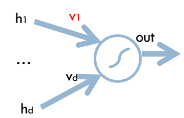
$(y - f(v \cdot h)) > 0$

?

$(y - f(v \cdot h)) < 0$

Output layer weights

$$v_k = v_k + (y - f(v \cdot h)) f'(v \cdot h) h_k$$



how far from correct and which direction

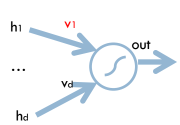
$(y - f(v \cdot h)) > 0$ prediction < label: increase the weight

$(y - f(v \cdot h)) < 0$ prediction > label: decrease the weight

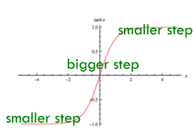
bigger difference = bigger change

Output layer weights

$$v_k = v_k + (y - f(v \cdot h)) f'(v \cdot h) h_k$$

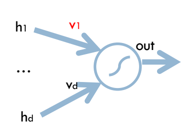


slope of the activation function where input is at



Output layer weights

$$v_k = v_k + (y - f(v \cdot h)) f'(v \cdot h) h_k$$



size and direction of the feature associated with this weight

Backpropagation: the details

Gradient descent method for learning weights by optimizing a loss function

$$\text{argmin}_{w,v} \sum_x \frac{1}{2}(y-\hat{y})^2$$

1. calculate output of all nodes
2. calculate the updates directly for the output layer
3. "backpropagate" errors through hidden layers

Backpropagation: hidden layer

output layer

$$= -(y - f(v \cdot h)) f'(v \cdot h) h_k$$

error | output activation slope | input

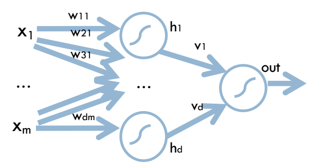
hidden layer

$$= -(y - f(v \cdot h)) f'(v \cdot h) v_k f'(w_k \cdot x) x_j$$

error | output activation slope | input

weight from hidden layer to output layer

slope of wx



$$\frac{dloss}{dv_k} = \frac{d}{dv_k}\left(\frac{1}{2}(y-\hat{y})^2\right)$$
$$= \frac{d}{dv_k}\left(\frac{1}{2}(y-f(v\cdot h)^2\right)$$
$$= (y-f(v\cdot h))\frac{d}{dv_k}\left(y-f(v\cdot h)\right)$$
$$= -(y-f(v\cdot h))\frac{d}{dv_k}f(v\cdot h)$$
$$= -(y-f(v\cdot h))f'(v\cdot h)\frac{d}{dv_k}v\cdot h$$
$$= -(y-f(v\cdot h))f'(v\cdot h)h_k$$

$$\frac{dloss}{dw_{kj}} = \frac{d}{dw_{kj}}\left(\frac{1}{2}(y-\hat{y})^2\right)$$
$$= \frac{d}{dw_{kj}}\left(\frac{1}{2}\left(y-f(v\cdot h)^2\right)\right)$$
$$= (y-f(v\cdot h))\frac{d}{dw_{kj}}\left(y-f(v\cdot h)\right)$$
$$= -(y-f(v\cdot h))\frac{d}{dw_{kj}}f(v\cdot h)$$
$$= -(y-f(v\cdot h))f'(v\cdot h)\frac{d}{dw_{kj}}v\cdot h$$
$$= -(y-f(v\cdot h))f'(v\cdot h)\frac{d}{dw_{kj}}v_k h_k$$
$$= -(y-f(v\cdot h))f'(v\cdot h)v_k\frac{d}{dw_{kj}}h_k$$
$$= -(y-f(v\cdot h))f'(v\cdot h)v_k\frac{d}{dw_{kj}}f(w_k\cdot x)$$
$$= -(y-f(v\cdot h))f'(v\cdot h)v_k f'(w_k\cdot x)\frac{d}{dw_{kj}}w_k\cdot x$$
$$= -(y-f(v\cdot h))f'(v\cdot h)v_k f'(w_k\cdot x)x_j$$

There's a bit of math to show this, but it's mostly just calculus…

Learning rate

Output layer update:

$$v_k = v_k + \eta h_k (y - f(v\cdot h)) f'(v\cdot h)$$

Hidden layer update:

$$w_{kj} = w_{kj} + \eta x_j f'(w_k\cdot x) v_k f'(v\cdot h)(y - f(v\cdot h))$$

- Adjust how large the updates we'll make (a parameter to the learning approach – like lambda for n-gram models)
- Often will start larger and then get smaller

Backpropagation implementation

for some number of iterations:

randomly shuffle training data

for each example:

- Compute all outputs going forward
- Calculate new weights and modified errors at output layer
- Recursively calculate new weights and modified errors on hidden layers based on recursive relationship
- Update model with new weights

Many variations

Momentum: include a factor in the weight update to keep moving in the direction of the previous update

Mini-batch:

- Compromise between online and batch
- Avoids noisiness of updates from online while making more educated weight updates

Simulated annealing:

- With some probability make a random weight update
- Reduce this probability over time

…

Challenges of neural networks?

Picking network configuration

Can be slow to train for large networks and large amounts of data

Loss functions (including squared error) are generally not convex *with respect to the parameter space*