Backpropagation

Dave Kauchak
Alexandra Papoutsaki
Zilong Ye
CSCI 051A
Spring 2022
Perceptron learning algorithm

initialize weights of the model randomly

repeat until you get all examples right:

for each “training” example (in a random order):
  calculate current prediction on the example
  if wrong:
    \[ w_i = w_i + \lambda \times (\text{actual} - \text{predicted}) \times x_i \]
Perceptron learning

A few missing details, but not much more than this

Keeps adjusting weights as long as it makes mistakes

If the training data is linearly separable, the perceptron learning algorithm is guaranteed to converge to the “correct” solution (where it gets all examples right)
A data set is **linearly separable** if you can separate one example type from the other with a line.

Which of these are linearly separable?
Which of these are linearly separable?

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_1 \text{ and } x_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_1 \text{ or } x_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_1 \text{ xor } x_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Which of these are linearly separable?

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_1 \text{ and } x_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_1 \text{ or } x_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_1 \text{ xor } x_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
XOR

\[
\begin{align*}
T &= ? \\
T &= ? \\
T &= ? \\
\text{Output} &= x_1 \text{ xor } x_2
\end{align*}
\]

<table>
<thead>
<tr>
<th>(x_1)</th>
<th>(x_2)</th>
<th>(x_1 \text{ xor } x_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
XOR

Input $x_1$

$T = 1$

$T = 1$

Input $x_2$

Output = $x_1 \text{ xor } x_2$

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_1 \text{ xor } x_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Learning in multilayer networks

Similar idea as perceptrons

Examples are presented to the network

If the network computes an output that matches the desired, nothing is done

If there is an error, then the weights are adjusted to balance the error
Learning in multilayer networks

Key idea for perceptron learning: if the perceptron’s output is different than the expected output, update the weights

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

perceptron

multi-layer network
Backpropagation

Say we get it wrong, and we now want to update the weights

We can update this layer just as if it were a perceptron
Backpropagation

Say we get it wrong, and we now want to update the weights

“back-propagate” the error (actual – predicted):

Assume all of these nodes were responsible for some of the error

How can we figure out how much they were responsible for?
Backpropagation

Say we get it wrong, and we now want to update the weights

Error for node $i$ is: $w_i \text{ error}$
Backpropagation

Say we get it wrong, and we now want to update the weights.

Update these weights and continue the process back through the network.
Backpropagation

calculate the error at the output layer

backpropagate the error up the network

Update the weights based on these errors

Can be shown that this is the appropriate thing to do based on our assumptions

That said, many neuroscientists don’t think the brain does backpropagation of errors
Neural network regression

Given enough hidden nodes, you can learn *any* function with a neural network

Challenges:

- **overfitting** – learning only the training data and not learning to generalize
- **picking a network structure**
- **can require a lot of tweaking of parameters, preprocessing, etc.**
Handwritten digits (MNIST)
Summary

Perceptrons, one-layer networks, are insufficiently expressive

Multi-layer networks are sufficiently expressive and can be trained by error back-propagation

Many applications including speech, driving, handwritten character recognition, fraud detection, driving, etc.
Our python NN module

Data:

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

```python
table = [
    ([0.0, 0.0, 0.0], [1.0]),
    ([0.0, 1.0, 0.0], [0.0]),
    ([1.0, 0.0, 0.0], [1.0]),
    ([1.0, 1.0, 0.0], [0.0]),
    ([0.0, 0.0, 1.0], [1.0]),
    ([0.0, 1.0, 1.0], [1.0]),
    ([1.0, 0.0, 1.0], [1.0]),
    ([1.0, 1.0, 1.0], [0.0])
]```
Data format

list of examples

table = \
[ ([0.0, 0.0, 0.0], [1.0]),
  ([0.0, 1.0, 0.0], [0.0]),
  ([1.0, 0.0, 0.0], [1.0]),
  ([1.0, 1.0, 0.0], [0.0]),
  ([0.0, 0.0, 1.0], [1.0]),
  ([0.0, 1.0, 1.0], [1.0]),
  ([1.0, 0.0, 1.0], [1.0]),
  ([1.0, 1.0, 1.0], [0.0]) ]

input list  output list

example = tuple
Training on the data

Construct a new network:

```python
>>> nn = NeuralNet(3, 2, 1)
```

constructor: constructs a new NN object

input nodes

hidden nodes

output nodes
Training on the data

Construct a new network:

```python
>>> nn = NeuralNet(3, 2, 1)
```

![Diagram of a neural network with 3 input nodes, 2 hidden nodes, and 1 output node.]
Training on the data

>>> nn.train(table)
error 0.195200
error 0.062292
error 0.031077
error 0.019437
error 0.013728
error 0.010437
error 0.008332
error 0.006885
error 0.005837
error 0.005047

by default, trains 1000 iterations and prints out error values every 100 iterations
After training, can look at the weights

```python
>>> nn.train(table)
>>> nn.get_IH_weights()
[[ [w1a, w1b, w1c],
  [w2a, w2b, w2c] ],
  [b1, b2]]
```
After training, can look at the weights

```python
>>> nn.get_HO_weights()
[[[w1a, w1b]],
 [b1]]
```
Many parameters to play with

```python
def train(data,
    learning_rate=0.01,
    iterations=1000, print_interval=100)
```

`nn.train(training_data)` carries out a training cycle. As specified earlier, the training data is a list of input-output pairs. There are three optional arguments to the train function.

- `learning_rate` defaults to 0.01
- `iterations` defaults to 1000. It specifies the number of passes over the training data
- `print_interval` defaults to 100. The value of the error is displayed after `print_interval` passes over the data; we hope to see the value decreasing. Set the value to 0 if you do not want to see the error values.
Calling with optional parameters

>>> nn.train(table, iterations = 5, print_interval = 1)
error 0.005033
error 0.005026
error 0.005019
error 0.005012
error 0.005005
Optional parameters

See:

https://cs.pomona.edu/classes/cs51a/examples(optional_parameters.txt)
Optional parameters

Check out the constructor (\_\_init\_\_ function) of NeuralNet for another interesting optional parameter: activation function!

It may be worth experimenting with different activation functions to see what happens to accuracy and run time...
Train vs. test

TrainData

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.00</td>
</tr>
<tr>
<td>0.2</td>
<td>0.04</td>
</tr>
<tr>
<td>0.4</td>
<td>0.16</td>
</tr>
<tr>
<td>0.6</td>
<td>0.36</td>
</tr>
<tr>
<td>0.8</td>
<td>0.64</td>
</tr>
<tr>
<td>1.0</td>
<td>1.00</td>
</tr>
</tbody>
</table>

TestData

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.3</td>
<td>0.09</td>
</tr>
<tr>
<td>0.5</td>
<td>0.25</td>
</tr>
<tr>
<td>0.7</td>
<td>0.49</td>
</tr>
<tr>
<td>0.8</td>
<td>0.64</td>
</tr>
<tr>
<td>0.9</td>
<td>0.81</td>
</tr>
</tbody>
</table>

```python
>>> nn.train(trainData)
>>> nn.test(testData)
```