Course feedback

Overall, how is the class going?

Course feedback

How is the difficulty of the class?

Course feedback

About how many hours a week do you spend on this class?
Course feedback

I like thinking in a way that is opposite from what I normally do in humanities courses. I enjoy the creativity of getting to the end goal by means that may be the same or different from the other people in the class.

Course feedback

we create something that is somewhat useful! somewhat understand how things / programs on our computers work

Course feedback

I think I would benefit more if you could break down the assignments. Rather than having one large weekly assignment, having it due by parts (like assignment 5 is) can help me stay on track and keep it more manageable.

Course feedback

Somehow increase the use of Piazza. The mentor sessions don't work well with my schedule, and I wish I could see what people are asking about.
Course feedback

It seems like it would be helpful for the mentors to have the solutions before lab because often they aren't sure what the correct answer is and spend a significant amount of time trying to figure it out themselves.

Course feedback

More info of how coding is done in the real world and it's applications and purposes...however I'm guessing that that will be covered in the weeks to come.

Optional parameters

- Look at optional_parameters.py

Artificial Neural Networks

Node (Neuron)

Edge (synapses)
W is the strength of signal sent between A and B.

If A fires and \( w \) is positive, then A stimulates B.

If A fires and \( w \) is negative, then A inhibits B.

### Training neural networks

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( x_3 )</th>
<th>( x_4 ) and ( x_5 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1 ( w_1 = ? )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0 ( w_2 = ? )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1 ( w_3 = ? )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1 ( w_4 = ? )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1 ( w_5 = ? )</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>

### Perceptron learning algorithm

repeat until you get all examples right:

- for each “training” example:
  - calculate current prediction on example
  - if wrong:
    - update weights and threshold towards getting this example correct
Perceptron learning

What could we adjust to make it right?

Weighted sum is 0.5, which is not equal or larger than the threshold

This weight doesn’t matter, so don’t change

Could increase any of these weights
Perceptron learning

Perceptron update rule

- if \textit{wrong}:
  - update weights and threshold towards getting this example correct

- if \textit{wrong}:

\[ w_i = w_i + \Delta w_i \]

\[ \Delta w_i = \lambda \cdot (\text{actual} - \text{predicted}) \cdot x_i \]
Perceptron learning

\[ w_i = w_i + \Delta w_i \]
\[ \Delta w_i = \lambda \times (\text{actual} - \text{predicted}) \times x_i \]

What if predicted = 1 and actual = 0?

We're over the threshold, so want to decrease weights: actual - predicted = -1

Perceptron learning

\[ w_i = w_i + \Delta w_i \]
\[ \Delta w_i = \lambda \times (\text{actual} - \text{predicted}) \times x_i \]

What does this do?

Only adjust those weights that actually contributed!
Perceptron learning

\[ w_i = w_i + \Delta w_i \]
\[ \Delta w_i = \lambda \times (\text{actual} - \text{predicted}) \times x_i \]

What does this do?

"learning rate": value between 0 and 1 (e.g. 0.1) adjusts how abrupt the changes are to the model

What about the threshold?

\[ w_i = w_i + \Delta w_i \]
\[ \Delta w_i = \lambda \times (\text{actual} - \text{predicted}) \times x_i \]
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:
    - \[ w_i = w_i + \lambda \times (\text{actual} - \text{predicted}) \times x_i \]
\[ w_i = w_i + \lambda \cdot (\text{actual} - \text{predicted}) \cdot x_i \]

If wrong:

- Decrease (0-1=-1) all non-zero \( x_i \) by 0.1

\[ \sum = 0.3; \text{predicted 1} \]

**Right or wrong?**

\[ \sum = 0.3; \text{predicted 1} \]

**Wrong**

\[ \sum = 0.3; \text{predicted 1} \]

**New weights?**

\[ \sum = 0.3; \text{predicted 1} \]
Right or wrong?

v = w + \lambda \cdot (actual - predicted) \cdot x_i

Wrong

Right. No update!
If wrong: $w_i = w_i + \lambda \ast (actual - predicted) \ast x_i$

new weights?

Right or wrong?

Right. No update!
### Example of Logistic Regression with Regularization

#### Data Points and Weights

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_1$ and $x_2$</th>
<th>$\lambda = 0.1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

#### Initial Weights

- $W_1 = 0.1$
- $W_2 = 0.4$
- $W_3 = -0.1$

#### Output Prediction

- Predicted $y = 2$ (1)

#### Weight Update

- $w_i = w_i + \lambda \times (\text{actual} - \text{predicted}) \times x_i$

#### Error Check

- If wrong:

  - $\sum = 0.3$: predicted 1

#### Adjustment

- Decrease $0-1=-1$ all non-zero $x_i$ by 0.1

---

### Example of Logistic Regression with Regularization

#### Data Points and Weights

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_1$ and $x_2$</th>
<th>$\lambda = 0.1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

#### Initial Weights

- $W_1 = 0.1$
- $W_2 = 0.4$
- $W_3 = -0.1$

#### Output Prediction

- Predicted $y = 2$ (1)

#### Weight Update

- $w_i = w_i + \lambda \times (\text{actual} - \text{predicted}) \times x_i$

#### Error Check

- If wrong:

  - $\sum = 0.2$: predicted 1

#### Adjustment

- Right. No update!
\[ w_i = w_i + \lambda \cdot (\text{actual} - \text{predicted}) \cdot x_i \]

if wrong:

\[ \text{sum} = \text{predicted } y \]

**Right. No update!**

**Are they all right?**

**Wrong**
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 with a line other.

Which of these are linearly separable?
Learning in multilayer networks

Similar idea as perceptrons

Examples are presented to the network

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

If there is an error, then the weights are adjusted to balance the error

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

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

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

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

$$\text{error} = \text{(actual} - \text{predicted)}$$

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

Update these weights and continue the process back through the network

Backpropagation

calculate the error at the predicted 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.
Popular for digit recognition and many computer vision tasks
http://yann.lecun.com/exdb/mnist/

Cog sci people like NNs

Expression/emotion recognition
  □ Gary Cottrell et al

Language learning

Interpreting Satellite Imagery for Automated Weather Forecasting

What NNs learned from youtube

What NNs learned from youtube

trained on 10M snapshots from youtube videos

NN with 1 billion connections

16,000 processors

Summary

Perceptrons, one layer networks, are insufficiently expressive

Multi-layer networks are sufficiently expressive and

can be trained by error back-propogation

Many applications including speech, driving, hand

written 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>x_1 and x_2</th>
<th>x_4</th>
<th>x_5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<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>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<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>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Our python NN module

Data:

list of examples

<table>
<thead>
<tr>
<th>table =</th>
</tr>
</thead>
</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], [0.0]),
| ( [1.0, 0.0, 1.0], [1.0]),
| ( [1.0, 1.0, 1.0], [0.0]) |

Data format

Calculated data

example = tuple

input list

predicted list
Training on the data

Construct a new network:

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

constructor: constructs a new NN object

input nodes

hidden nodes

predicted nodes

Training on the data

Construct a new network:

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

3 input nodes

2 hidden nodes

1 predicted node

After training, can look at the weights

```python
>>> nn.train(table)
>>> nn.get_IH_weights()
```

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

```python
>>> nn.train(table)
```

```python
>>> nn.get_LH_weights()
[[3.3435628797862624, -0.272324373735495],
 [4.846203738642956, -4.601230952566068],
 [5.423833151146873, 0.673636456637573],
 [2.938428644152128, 1.850976127273543]]
```

After training, can look at the weights

```python
>>> nn.get_HO_weights()
[[8.116192424400454],
 [5.358094903107918],
 [-4.73829543609533]]
```

Many parameters to play with

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

- learningRate defaults to 0.1.
- momentumFactor defaults to 0.1. The idea of momentum is discussed in the next section. Set it to 0 to suppress the effect of the momentum in the calculation.
- iterations defaults to 100. It specifies the number of passes over the training data.
- printInterval defaults to 100. The value of the error is displayed after printInterval 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.

You may specify some, or all, of the optional arguments by name in the following format:

```python
nn.train(trainingData,
    learningRate=0.1,
    momentumFactor=0.1,
    iterations=100,
    printInterval=10)
```

Calling with optional parameters

```python
>>> nn.train(table, iterations = 5, printInterval = 1)
error 0.005033
error 0.005026
error 0.005019
error 0.005012
error 0.005005
```
Train vs. test

```
>>> nn.train(train_data)
>>> nn.test(test_data)
```

![Table](http://www.sciencebytes.org/2011/05/03/blueprint-for-the-brain/)