# Kaggle: Project 2

<table>
<thead>
<tr>
<th>#</th>
<th>Δpub</th>
<th>Team Name</th>
<th>Kernel</th>
<th>Team Members</th>
<th>Score</th>
<th>Entries</th>
<th>Last</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>—</td>
<td>🦕 2045 🦕</td>
<td></td>
<td></td>
<td>0.81705</td>
<td>36</td>
<td>2d</td>
</tr>
<tr>
<td>2</td>
<td>▲ 1</td>
<td>Juicebox3.0 (Stay hydrated)</td>
<td></td>
<td></td>
<td>0.81082</td>
<td>25</td>
<td>1d</td>
</tr>
<tr>
<td>3</td>
<td>▼ 1</td>
<td>ZSV</td>
<td></td>
<td></td>
<td>0.81067</td>
<td>23</td>
<td>1d</td>
</tr>
<tr>
<td>4</td>
<td>—</td>
<td>Nothing but Nets</td>
<td></td>
<td></td>
<td>0.80600</td>
<td>24</td>
<td>4d</td>
</tr>
<tr>
<td>5</td>
<td>▲ 2</td>
<td>Jale Li</td>
<td></td>
<td></td>
<td>0.80577</td>
<td>10</td>
<td>1d</td>
</tr>
<tr>
<td>6</td>
<td>▼ 1</td>
<td>DMT</td>
<td></td>
<td></td>
<td>0.80414</td>
<td>29</td>
<td>1d</td>
</tr>
<tr>
<td>7</td>
<td>▲ 1</td>
<td>NAS</td>
<td></td>
<td></td>
<td>0.80290</td>
<td>24</td>
<td>1d</td>
</tr>
<tr>
<td>8</td>
<td>▼ 2</td>
<td>AJGARS</td>
<td></td>
<td></td>
<td>0.80271</td>
<td>25</td>
<td>1d</td>
</tr>
<tr>
<td>9</td>
<td>—</td>
<td>WIL</td>
<td></td>
<td></td>
<td>0.80071</td>
<td>28</td>
<td>2d</td>
</tr>
<tr>
<td>10</td>
<td>▲ 2</td>
<td>Cereal Killer</td>
<td></td>
<td></td>
<td>0.79736</td>
<td>22</td>
<td>1d</td>
</tr>
<tr>
<td>11</td>
<td>▼ 1</td>
<td>team-02-unmerged</td>
<td></td>
<td></td>
<td>0.79712</td>
<td>2</td>
<td>1d</td>
</tr>
<tr>
<td>12</td>
<td>▼ 1</td>
<td>team-02</td>
<td></td>
<td></td>
<td>0.79712</td>
<td>15</td>
<td>1d</td>
</tr>
<tr>
<td>13</td>
<td>—</td>
<td>LazyLearner</td>
<td></td>
<td></td>
<td>0.79319</td>
<td>26</td>
<td>1d</td>
</tr>
<tr>
<td>14</td>
<td>▲ 2</td>
<td>LR</td>
<td></td>
<td></td>
<td>0.79234</td>
<td>24</td>
<td>2d</td>
</tr>
<tr>
<td>15</td>
<td>▲ 5</td>
<td>Epte</td>
<td></td>
<td></td>
<td>0.79176</td>
<td>18</td>
<td>3d</td>
</tr>
</tbody>
</table>
Recall the perceptron

- We can take a linear combination and threshold it:

\[ h_w(x) = \text{sgn}(x \cdot w) = \begin{cases} +1 & \text{if } x \cdot w > 0 \\ -1 & \text{otherwise} \end{cases} \]

- The output is taken as the predicted class.
Decision surface of a perceptron

- Can represent many functions.
- To represent non-linearly separate functions (e.g. XOR), we could use a network of perceptron-like elements.
- If we connect perceptrons into networks, the error surface for the network is not differentiable (because of the hard threshold).
Example: A network representing XOR
Recall the sigmoid function

\[ \sigma(z) = \frac{1}{1 + e^{-z}} \]

- \( \sigma \) is the sigmoid function:
- It has the following nice property:

\[ \frac{d\sigma(z)}{dz} = \sigma(z)(1 - \sigma(z)) \]

We can derive a gradient descent rule to train:
  - One sigmoid unit; Multi-layer networks of sigmoid units.
Feed forward neural networks

- A collection of neurons with sigmoid activation, arranged in layers.
- Layer 0 is the **input layer**, its units just copy the input.
- Last layer (layer K) is the **output layer**, its units provide the output.
- Layers 1, .., K-1 are **hidden layers**, cannot be detected outside of network.
Why this name?

- In **feed-forward networks** the output of units in layer $k$ become input to the units in layers $k+1, k+2, \ldots, K$.

- No cross-connection between units in the same layer.

- No backward (“recurrent”) connections from layers downstream.

- Typically, units in layer $k$ provide input to units in layer $k+1$ only.

- In **fully-connected networks**, all units in layer $k$ provide input to all units in layer $k+1$. 
Feed-forward neural networks

Notation:

- $w_{ji}$ denotes weight on connection from unit $i$ to unit $j$.
- By convention, $x_{j0} = 1, \forall j$
- Output of unit $j$, denoted $o_j$ is computed using a sigmoid:
  \[ o_j = \sigma(w_j \cdot x_j) \]
  where $w_j$ is vector of weights entering unit $j$
  $x_j$ is vector of inputs to unit $j$
- By definition, $x_{ji} = o_i$.

Given an input, how do we compute the output? How do we train the weights?
Computing the output of the network

- Suppose we want network to make prediction about instance \( <x, y=?> \).

Run a **forward pass** through the network.

For layer \( k = 1 \ldots K \)

1. Compute the output of all neurons in layer \( k \):
   \[
   o_j = \sigma(w_j \cdot x_j), \forall j \in \text{Layer } k
   \]

2. Copy this output as the input to the next layer:
   \[
   x_{j, i} = o_i, \forall i \in \text{Layer } k, \forall j \in \text{Layer } k + 1
   \]

The output of the last layer is the predicted output \( y \).
Learning in feed-forward neural networks

- Assume the network structure (units+connections) is given.

- The learning problem is finding a good set of weights to minimize the error at the output of the network.

- Approach: gradient descent, because the form of the hypothesis formed by the network, $h_w$ is:
  - Differentiable! Because of the choice of sigmoid units.
  - Very complex! Hence direct computation of the optimal weights is not possible.
Gradient-descent preliminaries for NN

- Assume we have a fully connected network:
  - \( N \) input units (indexed 1, \ldots, \( N \))
  - \( H \) hidden units in a single layer (indexed \( N+1, \ldots, N+H \))
  - one output unit (indexed \( N+H+1 \))

- Suppose you want to compute the weight update after seeing instance \( <x, y> \).

- Let \( o_i, i = 1, \ldots, H+N+1 \) be the outputs of all units in the network for the given input \( x \).

- The sum-squared error function is:

\[
J(w) = \frac{1}{2}(y - h_w(x))^2 = \frac{1}{2}(y - o_{N+H+1})^2
\]
Gradient-descent update for output node

- Derivative with respects to the weights $w_{N+H+1,j}$ entering $o_{N+H+1}$:
  - Use the chain rule: $\frac{\partial J(w)}{\partial w} = (\frac{\partial J(w)}{\partial \sigma}) \cdot (\frac{\partial \sigma}{\partial w})$

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

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1 - \sigma(z))$$

$$\frac{\partial J(w)}{\partial \sigma} = -(y - o_{N+H+1})$$
Gradient-descent update for output node

- Derivative with respects to the weights $w_{N+H+1,j}$ entering $o_{N+H+1}$:
  - Use the chain rule: $\frac{\partial J(w)}{\partial w} = (\frac{\partial J(w)}{\partial \sigma}) \cdot (\frac{\partial \sigma}{\partial w})$

$$\frac{\partial J(w)}{\partial \sigma} = -(y - o_{N+H+1})$$

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

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1 - \sigma(z))$$

$$\frac{\partial J}{\partial w_{N+H+1,j}} = -(y - o_{N+H+1})o_{N+H+1}(1 - o_{N+H+1})x_{N+H+1,j}$$
Gradient-descent update for output node

- Derivative with respects to the weights $w_{N+H+1,j}$ entering $o_{N+H+1}$:
  - Use the chain rule: $\frac{\partial J(w)}{\partial w} = (\frac{\partial J(w)}{\partial \sigma}) \cdot (\frac{\partial \sigma}{\partial w})$

  $$\frac{\partial J}{\partial w_{N+H+1,j}} = -(y-o_{N+H+1})o_{N+H+1}(1-o_{N+H+1})x_{N+H+1,j}$$

- Hence, we can write:

  $$\frac{\partial J}{\partial w_{N+H+1,j}} = -\delta_{N+H+1}x_{N+H+1,j}$$

where:

$$\delta_{N+H+1} = (y - o_{N+H+1})o_{N+H+1}(1 - o_{N+H+1})$$
Gradient-descent update for hidden node

- The derivative wrt the other weights, $w_{l,j}$ where $j = 1, \ldots, N$ and $l = N+1, \ldots, N+H$ can be computed using the chain rule:

$$\frac{\partial J}{\partial w_{l,j}} = -(y - o_{N+H+1}) o_{N+H+1}(1 - o_{N+H+1})$$

$$\cdot \frac{\partial}{\partial w_{l,j}} (w_{N+H+1} \cdot x_{N+H+1})$$

$$= -\delta_{N+H+1} w_{N+H+1,l} \frac{\partial}{\partial w_{l,j}} x_{N+H+1,l}$$

- Recall that $x_{N+H+1,l} = o_l$. Hence we have:

$$\frac{\partial}{\partial w_{l,j}} x_{N+H+1,l} = o_l (1 - o_l) x_{l,j}$$

- Putting these together and using similar notation as before:

$$\frac{\partial J}{\partial w_{l,j}} = -o_l (1 - o_l) \delta_{N+H+1} w_{N+H+1,l} x_{l,j} = -\delta_l x_{l,j}$$
Gradient-descent update for hidden node

- The derivative wrt the other weights, $w_{k,j}$ where $j = 1, \ldots, N$ and $k = N+1, \ldots, N+H$ can be computed again using chain rule.

\[ \delta_h \leftarrow o_h (1 - o_h) w_{N+H+1,h} \delta_{N+H+1} \]

\[ \delta_{N+H+1} \leftarrow o (1 - o) (y - o) \]
Stochastic gradient descent

- Initialize all weights to small random numbers.

- Repeat until convergence:
  - Pick a training example.
  - Feed example through network to compute output $o = o_{N+H+1}$.
  - For the output unit, compute the correction:
    \[
    \delta_{N+H+1} \leftarrow o(1-o)(y-o)
    \]
  - For each hidden unit $h$, compute its share of the correction:
    \[
    \delta_h \leftarrow o_h(1-o_h)w_{N+H+1,h}\delta_{N+H+1}
    \]
  - Update each network weight:
    \[
    w_{h,i} \leftarrow w_{h,i} + \alpha_{h,i}\delta_h x_{h,i}
    \]
Organizing the training data

- **Stochastic gradient descent**: Compute error on a single example at a time (as in previous slide).

- **Batch gradient descent**: Compute error on all examples.
  - Loop through the training data, accumulating weight changes.
  - Update all weights and repeat.

- **Mini-batch gradient descent**: Compute error on small subset.
  - Randomly select a “mini-batch” (i.e. subset of training examples).
  - Calculate error on mini-batch, apply to update weights, and repeat.
Expressiveness of feed-forward NN

A single sigmoid neuron?
Expressiveness of feed-forward NN

A single sigmoid neuron?

• Same representational power as a perceptron: Boolean AND, OR, NOT, but not XOR.
Expressiveness of feed-forward NN

A single sigmoid neuron?

• Same representational power as a perceptron: Boolean AND, OR, NOT, but not XOR.

A neural network with a single hidden layer?
Learning the identity function

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000000</td>
<td>10000000</td>
</tr>
<tr>
<td>01000000</td>
<td>01000000</td>
</tr>
<tr>
<td>00100000</td>
<td>00100000</td>
</tr>
<tr>
<td>00010000</td>
<td>00010000</td>
</tr>
<tr>
<td>00001000</td>
<td>00001000</td>
</tr>
<tr>
<td>00000100</td>
<td>00000100</td>
</tr>
<tr>
<td>00000010</td>
<td>00000010</td>
</tr>
<tr>
<td>00000001</td>
<td>00000001</td>
</tr>
</tbody>
</table>
Learning the identity function

• Neural network structure:

• Learned hidden layer weights:
Expressiveness of feed-forward NN

A single sigmoid neuron?
- Same representational power as a perceptron: Boolean AND, OR, NOT, but not XOR.

A neural network with a single hidden layer?
- Can represent every boolean function, but might require a number of hidden units that is exponential in the number of inputs.
- Every bounded continuous function can be approximated with arbitrary precision by a boolean function.
Expressiveness of feed-forward NN

A single sigmoid neuron?

- Same representational power as a perceptron: Boolean AND, OR, NOT, but not XOR.

A neural network with a single hidden layer?

- Can represent every boolean function, but might require a number of hidden units that is exponential in the number of inputs.
- Every bounded continuous function can be approximated with arbitrary precision by a boolean function.

A neural network with two hidden layers?
Expressiveness of feed-forward NN

A single sigmoid neuron?
• Same representational power as a perceptron: Boolean AND, OR, NOT, but not XOR.

A neural network with a single hidden layer?
• Can represent every boolean function, but might require a number of hidden units that is exponential in the number of inputs.
• Every bounded continuous function can be approximated with arbitrary precision by a boolean function.

A neural network with two hidden layers?
• Any function can be approximated to arbitrary accuracy by a network with two hidden layers.
Project 3: Visual add / multiply (due Nov.13)

• For each image, give the result of the equation.
  – If it’s an “A”, add the 2 digits. If it’s an “M”, multiply the 2 digits.
Final notes

• What you should know:
  – Training by backpropagation.

• Additional information about neural networks:
  Video & slides from the Montreal Deep Learning Summer School:
  http://videolectures.net/deeplearning2017_larochelle_neural_networks/
  https://drive.google.com/file/d/0ByUKRdiCDK7-c2s2RjBiSms2UzA/view?usp=drive_web
  https://drive.google.com/file/d/0ByUKRdiCDK7-UXB1R1ZpX082MEk/view?usp=drive_web

• Tutorial 3 is today! TR3120, 6-7pm.

• Project #2 peer reviews will open next Monday on CMT.