Neural Networks
CIML Chapter 10
Daniel Lowd
Partially based on slides by Pedro Domingos

Properties of neural nets:
• Many neuron-like threshold switching units
• Many weighted interconnections among units
• Highly parallel, distributed process
• Emphasis on tuning weights automatically

Connectionist Models
Consider humans:
• Neuron switching time ~ .001 second
• Number of neurons ~ 10^{10}
• Connections per neuron ~ 10^{4–5}
• Scene recognition time ~ .1 second
• 100 inference steps doesn’t seem like enough
  ⇒ Much parallel computation

How can we combine linear models?
• Idea 1: Linear model of linear models
  \[ a_i = w_i^T x + b_i \] for \( i = 1 \ldots N \)
  \[ g(x) = w^T a + b \]

• Idea 2: Linear model of non-linear functions of linear models
  \[ a_i = f(w_i^T x + b) \] for \( i = 1 \ldots N \)
  \[ g(x) = w^T a + b \]
Activation Functions

Also called “link functions”

\[ \text{sign}(a) = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{if } a = 0 \\ -1 & \text{if } a < 0 \end{cases} \]

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

\[ \tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}} \]

\[ \text{ReLU}(a) = \max(a, 0) \]

\[ \text{SoftPlus}(a) = \log(1 + e^a) \]

Expressiveness of Neural Nets

Boolean functions:
- Every Boolean function can be represented by network with single hidden layer
- But might require exponential (in number of inputs) hidden units

Continuous functions:
- Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers

Multilayer Networks of Sigmoid Units

We can derive gradient descent rules to train
- One sigmoid unit
- Multilayer networks of sigmoid units → Backpropagation

Backpropagation

- Gradient descent + chain rule
- Want to minimize overall loss (e.g., squared loss):
  \[ \min_w \frac{1}{2} \sum_y \left( y - \sum_i w_i f_i(x) \right)^2 \]

- Gradient for outer weights \( w \), where \( h_i \) is hidden units:
  \[ \nabla_w = - \sum_i \varepsilon_i h_i \]

Backpropagation, continued:

\[ \mathcal{L}(W) = \frac{1}{2} \left( y - \sum_i w_i f_i(x) \right)^2 \]

\[ \frac{\partial \mathcal{L}}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial f_i} \frac{\partial f_i}{\partial w_i} \]

\[ \frac{\partial \mathcal{L}}{\partial f_i} = - \left( y - \sum_j w_j f_j(x) \right) v_i = -\varepsilon_i \]

\[ \frac{\partial f_i}{\partial w_i} = f'(w_i \cdot x) x \]

\[ \nabla_w = -\varepsilon_i f'(w_i \cdot x) x \]
Two Layer Network Training

\[ \text{Algorithm: } \text{TwoLayerNetworkTrain}(D, K, \text{MaxIter}) \]

1. \( W \leftarrow \text{KxK matrix of small random values} \) // initialize input layer weights
2. \( v \leftarrow \text{K vector of small random values} \) // initialize output layer weights
3. for all \((x_i, y_i) \in D\) do
   4. \( G \leftarrow \text{K matrix of zeros} \) // initialize input layer gradient
   5. \( g \leftarrow \text{K vector of zeros} \) // initialize output layer gradient
   6. for \( i = 1 \) to \( K \) do
      7. \( a_i \leftarrow w_i \cdot \hat{x} \) // compute activation of hidden unit \( i \)
     8. \( h_i \leftarrow \tanh(a_i) \) // compute output unit
   9. \( \hat{y} \leftarrow v \cdot h \) // compute output \( \hat{y} \)
10. \( e \leftarrow y - \hat{y} \) // compute error
11. \( g \leftarrow g + e \cdot h \) // update gradient for output layer
   12. for \( i = 1 \) to \( K \) do
      13. \( G_i \leftarrow G_i + (1 - \tanh(2)(a_i)) \cdot x \) // update gradient for input layer
     14. end for
15. \( W \leftarrow W - \eta G \) // update input layer weights
16. \( v \leftarrow v - \eta g \) // update output layer weights
   17. end for
18. return \( W, v \)

Putting this together, we get that the gradient with respect to \( w_i \) is:

\[ r_{w_i} = \sum_{j} v_j f(0) \]

Intuitively you can make sense of this. If the overall error of the predictor \( e \) is small, you want to make small steps. If \( v_i \) is small for hidden unit \( i \), then this means that the output is not particularly sensitive to the activation of the \( i \)th hidden unit. Thus, its gradient should be small. If \( v_i \) flips sign, the gradient at \( w_i \) should also flip signs. The name back-propagation comes from the fact that you propagate gradients backward through the network, starting at the end.

The complete instantiation of gradient descent for a two layer network with \( K \) hidden units is sketched in Algorithm 10. Note that this really is exactly a gradient descent algorithm; the only different is that the computation of the gradients of the input layer is moderately complicated.

What would happen to this algorithm if you wanted to optimize exponential loss instead of squared error? What if you wanted to add in weight regularization?

As a bit of practical advice, implementing the back-propagation algorithm can be a bit tricky. Sign errors often abound. A useful trick is first to keep \( W \) fixed and work on just training \( v \). Then keep \( v \) fixed and work on training \( W \). Then put them together.

If you like matrix calculus, derive the same algorithm starting from Eq (10.3).

More on Backpropagation
- Gradient descent over entire network weight vector
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum
  - In practice, often works well
  - (can run multiple times)
- Often include weight momentum \( \alpha \)
  \[ \Delta w_{ij}(n) = \eta \delta_{ij} + \alpha \Delta w_{ij}(n - 1) \]
- Minimizes error over training examples
  - Will it generalize well to subsequent examples?
- Training can take thousands of iterations → slow!
- Using network after training is very fast

Learning Hidden Layer Representations

A target function:

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000000</td>
<td>1000000</td>
</tr>
<tr>
<td>0100000</td>
<td>0100000</td>
</tr>
<tr>
<td>0010000</td>
<td>0100000</td>
</tr>
<tr>
<td>0001000</td>
<td>0001000</td>
</tr>
<tr>
<td>0000100</td>
<td>0000100</td>
</tr>
<tr>
<td>0000010</td>
<td>0000010</td>
</tr>
<tr>
<td>0000001</td>
<td>0000001</td>
</tr>
</tbody>
</table>

Can this be learned?

Learned hidden layer representation:

<table>
<thead>
<tr>
<th>Input</th>
<th>Hidden Values</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000000</td>
<td>.89 .04 .08</td>
<td>1000000</td>
</tr>
<tr>
<td>0100000</td>
<td>.01 .11 .88</td>
<td>0100000</td>
</tr>
<tr>
<td>0010000</td>
<td>.01 .97 .27</td>
<td>0010000</td>
</tr>
<tr>
<td>0001000</td>
<td>.99 .97 .71</td>
<td>0001000</td>
</tr>
<tr>
<td>0000100</td>
<td>.03 .05 .02</td>
<td>0000100</td>
</tr>
<tr>
<td>0000010</td>
<td>.22 .99 .99</td>
<td>0000010</td>
</tr>
<tr>
<td>0000001</td>
<td>.80 .01 .98</td>
<td>0000001</td>
</tr>
<tr>
<td>0000000</td>
<td>.01 .94 .01</td>
<td>0000000</td>
</tr>
</tbody>
</table>
Convergence of Backpropagation

Gradient descent to some local minimum
- Perhaps not global minimum...
- Add momentum
- Stochastic gradient descent
- Train multiple nets with different initial weights

Nature of convergence
- Initialize weights near zero
- Therefore, initial networks near-linear
- Increasingly non-linear functions possible as training progresses

Overfitting in Neural Nets

Overfitting Avoidance

Penalize large weights:
\[
E(\theta) = \frac{1}{2} \sum_{i,j} (t_{i,j} - o_{i,j})^2 + \eta \sum_{j} w_j^2
\]

Train on target slopes as well as values:
\[
E(\theta) = \frac{1}{2} \sum_{i,j} (t_{i,j} - o_{i,j})^2 + \mu \sum_{j} \left( \frac{\partial_{\theta_{i,j}}}{\theta_{i,j}} - \frac{\partial_{w_j}}{w_j} \right)^2
\]

Weight sharing
- Early stopping
Neural Networks: Summary

- Perceptrons
- Gradient descent
- Multilayer networks
- Backpropagation

ImageNet

- 1.2 million images; 1000 categories
- Goal: Predict the correct label in the top 5.
- State-of-the-art: < 5% error!
  (Better than humans.)

Neural Network Architecture

Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the base parts of the top of the figure while the other runs the base parts at the bottom. The GPUs communicate only at certain layers. The network’s input is 150,528-dimensional, and the number of neurons in the network’s remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

Key Techniques: Convolution

- Apply the same set of weights to each patch of the image
- Leads to a set of local attributes, e.g., which textures or lines are present in each position

Key Techniques: Max Pooling

- In images, you may not care where a tire appears — as long as there’s a tire in the image, you may guess that it’s a vehicle.
- Solution: Take the maximum of a set of outputs, each representing a similar feature but in different locations. Works well with convolution.

Key Techniques: Dropout

- How do you avoid overfitting?
- Train the network to work well when data or hidden units are missing.
Key Techniques: Data + GPU

- Training massive networks requires massive amounts of data (1+ million images)
- GPUs help scalability a lot.
  - Repetitive structure of neural network works well with vectorization

Key Techniques: ReLU

- Sigmoids are ok, but ReLUs are often better!
- Rectified Linear Unit:
  \[ \text{ReLU}(a) = \max(0, a) \]
- Linear gradient! Fast, and provides useful response at a wide range of values.