Artificial Neural Networks

Oliver Schulte
School of Computing Science
Simon Fraser University

Neural Networks

- Neural networks arise from attempts to model human/animal brains
  - Many models, many claims of biological plausibility
- We will focus on multi-layer perceptrons
  - Mathematical properties rather than biological plausibility

Uses of Neural Networks

- Pros
  - Good for continuous input variables.
  - General continuous function approximators.
  - Highly non-linear.
  - Learn features.
  - Good to use in continuous domains with little knowledge:
    - When you don't know good features.
    - You don't know the form of a good functional model.
- Cons
  - Not interpretable, "black box".
  - Learning is slow.
  - Good generalization can require many datapoints.

Function Approximation Demos

- Home Value of Hockey State https://user-images.githubusercontent.com/22108101/28182140-eb64b49a-67bf-11e7-97aa-046298f721e5.jpg
- Function Learning Examples (open in Safari) http://neuron.eng.wayne.edu/bpFunctionApprox/bpFunctionApprox.html

Applications

There are many, many applications.

- World-Champion Backgammon Player.
  http://en.wikipedia.org/wiki/TD-Gammon
  http://en.wikipedia.org/wiki/Backgammon
- No Hands Across America Tour.
  http://www.cs.cmu.edu/afs/cs/usr/tjochem/www/nhaa/nhaa_home_page.html
- Digit Recognition with 99.26% accuracy.
- Speech Recognition
  http://research.microsoft.com/en-us/news/features/speechrecognition-082911.aspx
- http://deeplearning.net/demos/

Outline

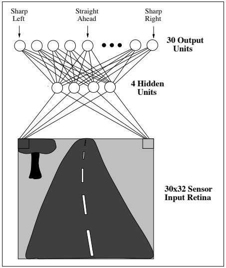
Feed-forward Networks

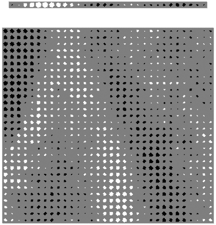
Network Training

Error Backpropagation

Examples

No Hands Across America





Outline

Feed-forward Networks

Network Training

Error Backpropagation

Examples

Neurons

Model of an individual neuron j

- Pass input in_j through a non-linear activation function to get output $a_j = g(in_j)$
- For non-input nodes, the input is the weighted linear sum of connected node activations + bias $w_{0,j}$:

$$in_j = \sum_{i=0}^{n} w_{ij} a_i$$

Neurons

Model of an individual neuron j

- Pass input in_j through a non-linear activation function to get output $a_j = g(in_j)$
- For non-input nodes, the input is the weighted linear sum of connected node activations + bias $w_{0,j}$:

$$in_j = \sum_{i=0}^{n} w_{ij} a_i$$

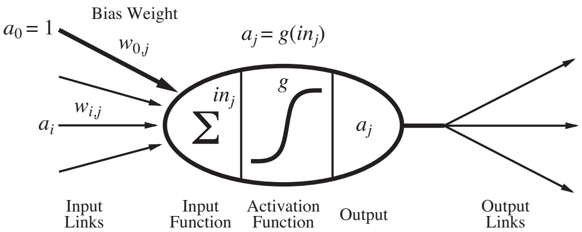
from Russell and Norvig, AIMA2e

Neurons

Model of an individual neuron j

- Pass input in_j through a non-linear activation function to get output $a_j = g(in_j)$
- For non-input nodes, the input is the weighted linear sum of connected node activations + bias $w_{0,j}$:

$$in_j = \sum_{i=0}^{n} w_{ij} a_i$$

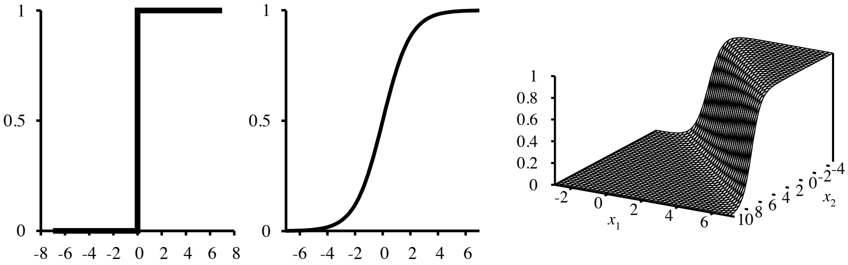


from Russell and Norvig, AIMA2e

Activation Functions

- Can use a variety of activation functions
  - Sigmoidal (S-shaped)
    - Logistic sigmoid $1/(1+\exp(-a))$ (useful for binary classification)
    - Hyperbolic tangent $\tanh$
  - Radial basis function $z_j = \sum_i (x_i - w_{ji})^2$
  - Softmax
    - Useful for multi-class classification
  - Hard Threshold
  - Rectified Linear Unit (deep learning)
  - ...
- Should be differentiable for gradient-based learning (later)
- Can use different activation functions in each unit

Activation Functions Visualized

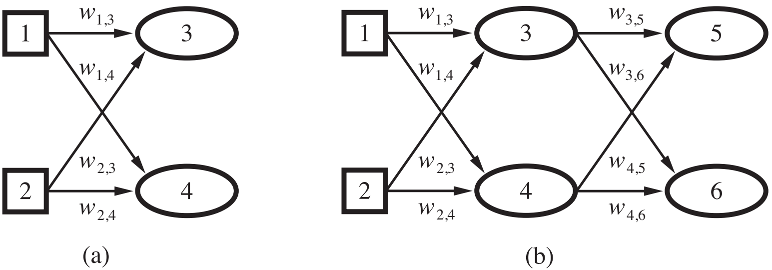


Left Threshold

Middle Logistic sigmoid $Logistic(x) = \frac{1}{1+\exp(-x)}$ maps a real number to a probability

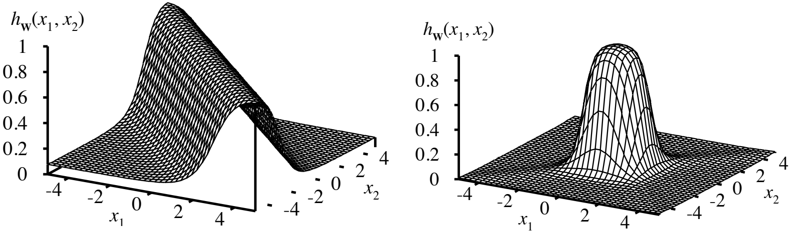
Right Logistic regression $Logistic(\boldsymbol{w} \bullet \boldsymbol{x})$

Network of Neurons



Function Composition

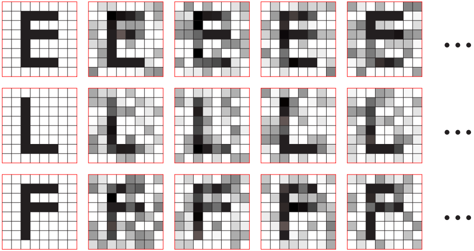
Think logic circuits

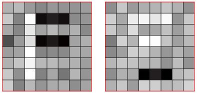


Two opposite-facing sigmoids = ridge. Two ridges = bump.

Hidden Units As Feature Extractors

sample training patterns



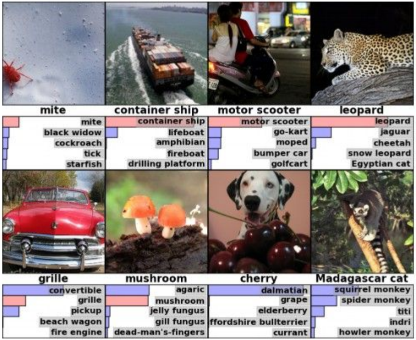


learned input-to-hidden weights

- 64 input nodes
- 2 hidden units
- 2x learned weight vector at hidden unit

Image Analysis Tasks

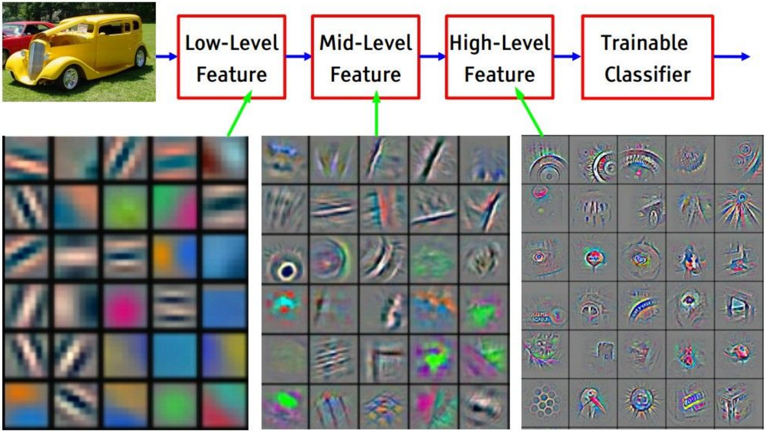
Classification



Retrieval

[Krizhevsky 2012]

Neural Net Learned Features



Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

Outline

Feed-forward Networks

Network Training

Error Backpropagation

Examples

Measuring Training Error

- Given a specified network structure, how do we set its parameters (weights)?
  - As usual, we define a criterion to measure how well our network performs, optimize against it
- Training data are $(\boldsymbol{x}_n, \boldsymbol{y}_n)$
- Corresponds to neural net with multiple output nodes
- Given a set of weight values $\boldsymbol{w}$, the network defines a function $\boldsymbol{h}_{\boldsymbol{w}}(\boldsymbol{x})$.
- Can train by minimizing L2 loss:

$$E(\boldsymbol{w}) = \sum_{n=1}^{N} |\boldsymbol{h}_{\boldsymbol{w}}(\boldsymbol{x}_n) - \boldsymbol{y}_n)|^2 = \sum_{n=1}^{N} \sum_{k} (y_k - a_k)^2$$

where k indexes the output nodes

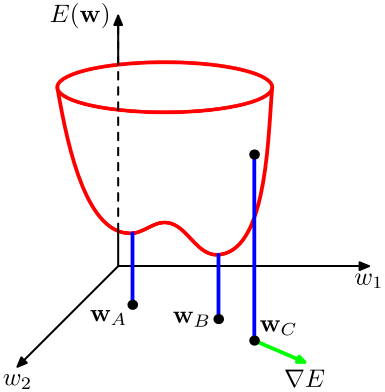
Measuring Training Error

- Given a specified network structure, how do we set its parameters (weights)?
  - As usual, we define a criterion to measure how well our network performs, optimize against it
- Training data are $(\boldsymbol{x}_n, \boldsymbol{y}_n)$
- Corresponds to neural net with multiple output nodes
- Given a set of weight values $\boldsymbol{w}$, the network defines a function $\boldsymbol{h}_{\boldsymbol{w}}(\boldsymbol{x})$.
- Can train by minimizing L2 loss:

$$E(\boldsymbol{w}) = \sum_{n=1}^{N} |\boldsymbol{h}_{\boldsymbol{w}}(\boldsymbol{x}_n) - \boldsymbol{y}_n)|^2 = \sum_{n=1}^{N} \sum_{k} (y_k - a_k)^2$$

where k indexes the output nodes

Parameter Optimization



- For either of these problems, the error function $E(\boldsymbol{w})$ is nasty
  - Nasty = non-convex
  - Non-convex = has local minima

Gradient Descent

- The function $h_{\mathbf{w}}(\mathbf{x})$ implemented by a network is complicated.
- No closed-form: Use gradient descent.
- It isn't obvious how to compute error function derivatives with respect to *hidden* weights.
  - The credit assignment problem.
- Backpropagation solves the credit assignment problem

Outline

Error Backpropagation

- Backprop is an efficient method for computing error derivatives $\frac{\partial E_n}{\partial w_{ji}}$ for *all* nodes in the network. Intuition:
  1. Calculating derivatives for weights connected to output nodes is easy.
  2. Treat the derivatives as virtual "error"—how far is each node activation "off". Compute derivative of error for nodes in previous layer.
  3. Repeat until you reach input nodes.
- Propagates backwards the output error signal through the network.

Error at the output nodes

- First, feed training example $\boldsymbol{x}_n$ forward through the network, storing all node activations a_i
- Calculating derivatives for weights connected to output nodes is easy.
- For output node j with activation $a_j = g(in_j) = g(\sum_i w_{ij} a_i)$:

$$\frac{\partial E_n}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \frac{1}{2}(y_j - a_j)^2 = -a_j \times g'(in_j) \times (y_j - a_j)$$

- 0 if no error, or if input a_i from node i is 0.
- **Modified Error**: $\Delta[j] \equiv g'(in_j)(y_j - a_j)$.
- Gradient Descent Weight Update:

$$w_{ij} \leftarrow w_{ij} + \alpha \times a_i \times \Delta[j]$$

Error at the output nodes

- First, feed training example $\boldsymbol{x}_n$ forward through the network, storing all node activations a_i
- Calculating derivatives for weights connected to output nodes is easy.
- For output node j with activation $a_j = g(in_j) = g(\sum_i w_{ij} a_i)$:

$$\frac{\partial E_n}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \frac{1}{2}(y_j - a_j)^2 = -a_j \times g'(in_j) \times (y_j - a_j)$$

ai

- 0 if no error, or if input a_i from node i is 0.
- **Modified Error**: $\Delta[j] \equiv g'(in_j)(y_j - a_j)$.
- Gradient Descent Weight Update:

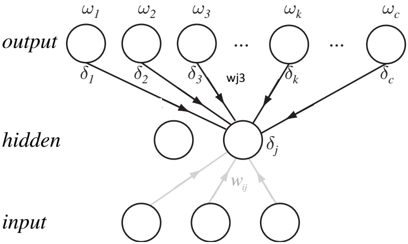
$$w_{ij} \leftarrow w_{ij} + \alpha \times a_i \times \Delta[j]$$

Error at the hidden nodes

- Consider a hidden node i connected to downstream nodes in the next layer.
- The **modified error** signal $\Delta[i]$ is node activation derivative, times the *weighted sum of contributions to the connected errors.*
- In symbols,

$$\Delta[i] = g'(in_i) \sum_j w_{ij} \Delta[j].$$

Backpropagation Picture



The error signal at a hidden unit is proportional to the error signals at the units it influences:

$$\Delta[j] = g'(in_j) \sum_k w_{jk} \Delta[k].$$

The Backpropagation Algorithm

1. Apply input vector $\boldsymbol{x}_n$ and forward propagate to find all inputs in_i and outputs a_i.
2. Evaluate the error signals Δ_k for all output nodes.
3. Backpropagate the Δ_k to obtain error signals Δ_j for each hidden node.
4. Perform the gradient descent updates for each weight vector w_{ij}:

$$w_{ij} \leftarrow w_{ij} + \alpha \times a_i \times \Delta[j]$$

Demo AIspace `http://aispace.org/neural/`.

Other Learning Topics

- Regularization: L2-regularizer (weight decay).
- Experimenting with Network Architectures is often key.
- Learn Architecture
  - Prune Weights: the Optimal Brain Damage Method.
  - Grow Network: Tiling, Cascade-Correlation Algorithm.

Outline

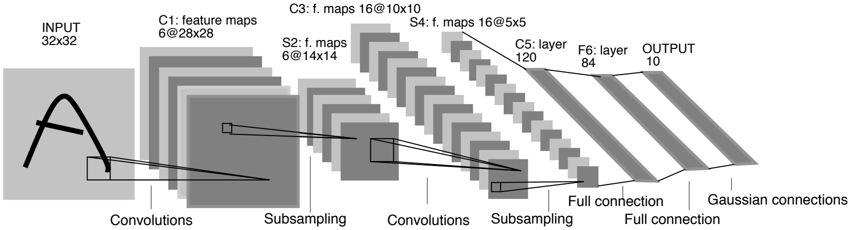
Applications of Neural Networks

- Many success stories for neural networks
  - Credit card fraud detection
  - Hand-written digit recognition
  - Face detection
  - Autonomous driving (CMU ALVINN)

Hand-written Digit Recognition

- MNIST - standard dataset for hand-written digit recognition
  - 60000 training, 10000 test images

LeNet-5



- LeNet developed by Yann LeCun et al.
  - Convolutional neural network
    - Local receptive fields (5x5 connectivity)
    - Subsampling (2x2)
    - Shared weights (reuse same 5x5 “filter”)
    - Breaking symmetry
- See
  http://www.codeproject.com/KB/library/NeuralNetRecognition.aspx

- The 82 errors made by LeNet5 (0.82% test error rate)

Conclusion

- Feed-forward networks can be used for predicting discrete or continuous target variables
- Very expressive, can approximate arbitrary continuous functions.
- Different activation functions possible.
- Learning is more difficult, error function has many local minima
  - Use stochastic gradient descent, obtain (good?) local minimum
- Backpropagation for efficient gradient computation.