Artificial Neural Networks

Oliver Schulte - CMPT 726

Neural Networks

- Neural networks arise from attempts to model human/animal brains
  - Many models, many claims of biological plausibility
  - We will focus on statistical and computational properties rather than plausibility
- An artificial neural network is a general *function approximator*
- The inner or hidden layers compute *learned basis functions*

Uses of Neural Networks

- Pros
  - Good for continuous input variables.
  - General continuous function approximators.
  - Highly non-linear.
  - Trainable basis functions.
  - 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/

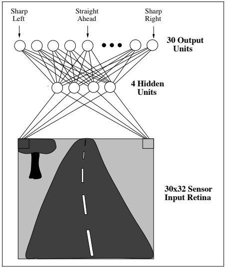
Outline

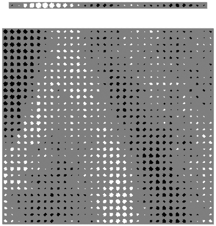
Outline

Feed-forward Networks

See powerpoint for network pictures

No Hands Across America





Non-linear Activation Functions

- Bad news: stacking linear regressions is equivalent to a single linear regression.
- Simple approach: apply a *non-linear* function g to the linear combination.
- Pass input in_j through a non-linear activation function $g(\cdot)$ to get output $a_j = g(in_j)$
  - Model of an individual neuron

from Russell and Norvig, AIMA3e

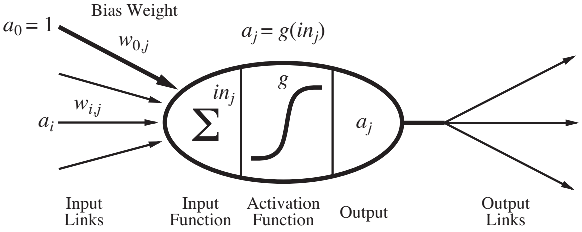
Non-linear Activation Functions

- Bad news: stacking linear regressions is equivalent to a single linear regression.
- Simple approach: apply a *non-linear* function g to the linear combination.
- Pass input in_j through a non-linear activation function $g(\cdot)$ to get output $a_j = g(in_j)$
    - Model of an individual neuron

from Russell and Norvig, AIMA3e

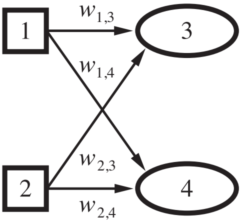
Non-linear Activation Functions

- Bad news: stacking linear regressions is equivalent to a single linear regression.
- Simple approach: apply a *non-linear* function g to the linear combination.
- Pass input in_j through a non-linear activation function $g(\cdot)$ to get output $a_j = g(in_j)$
    - Model of an individual neuron

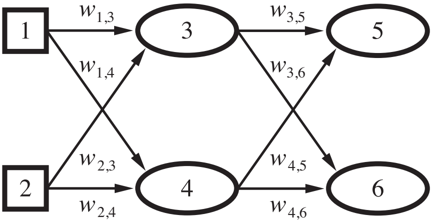


from Russell and Norvig, AIMA3e

Network of Neurons



(a)



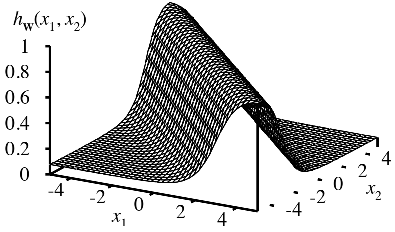
(b)

Activation Functions

- Can use a variety of activation functions
  - Sigmoidal (S-shaped)
    - Logistic sigmoid $1/(1+\exp(-a))$ (useful for binary classification)
    - Hyperbolic tangent $\tanh$
  - Softmax
    - Useful for multi-class classification
  - Rectified Linear Unit (RLU) $max(0, x)$
  - ...
- Should be differentiable for gradient-based learning (later)
- Can use different activation functions in each unit
- See `http://aispace.org/neural/`.

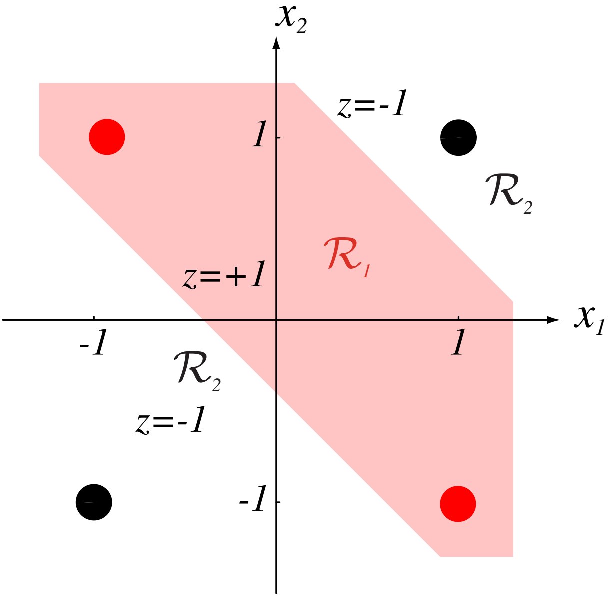
Function Composition

Think logic circuits

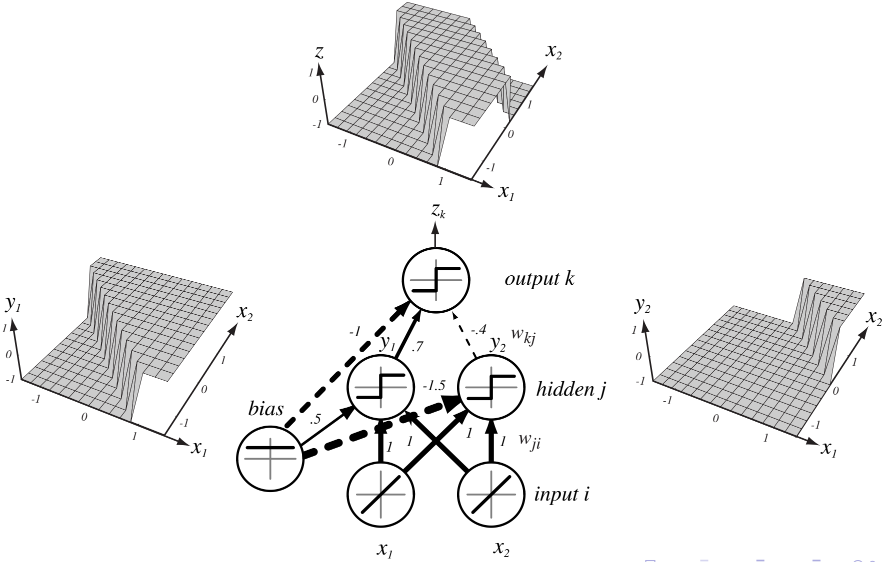


Two opposite-facing sigmoids = ridge. Two ridges = bump.

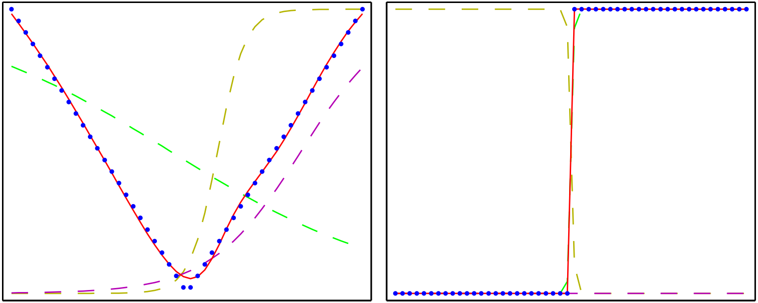
The XOR Problem Revisited



The XOR Problem Solved



Hidden Units Compute Basis Functions

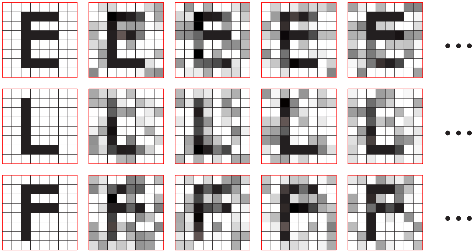


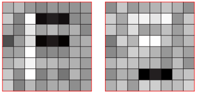
- red dots = network function
- dashed line = hidden unit activation function.
- blue dots = data points

Network function is roughly the sum of activation functions.

Hidden Units As Feature Extractors

sample training patterns





learned input-to-hidden weights

- 64 input nodes
- 2 hidden units
- learned weight matrix at hidden units

Outline

Network Training

- 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

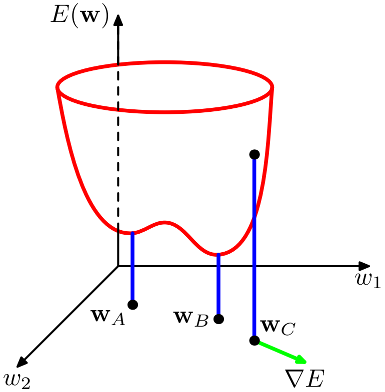
Network Training

- 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 $h_w(x)$ 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
- We will present the algorithm first, then prove that it implements gradient descent

Outline

Error Backpropagation

- Backprop is an efficient method for computing error derivatives $\frac{\partial E}{\partial w_{ij}}$ for *all* weights in the network. Intuition:
  1. Calculating derivatives for weights connected to output nodes is easy.
  2. Treat the derivatives as virtual "error", compute derivative of error for nodes in previous layer.
  3. Repeat until you reach input nodes.
- This procedure propagates backwards the output error signal through the network.
- Stochastic Gradient Descent: Fix input $\boldsymbol{x} \equiv \boldsymbol{x}_n$ and target output $\boldsymbol{y} \equiv \boldsymbol{y}_n$, resulting in error E_n.

Error at the output nodes

- First, feed training example $\boldsymbol{x}_n$ forward through the network, storing all node activations a_j
- Calculating derivatives for weights connected to output nodes is easy.
    - like logistic regression with input "features" a_j
- For output node k with activation $a_k = g(in_k) = g(\sum_j w_{jk} a_j)$:

$$\frac{\partial E_n}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} \frac{1}{2}(y_k - a_k)^2 = -a_j \times g'(in_k) \times (y_k - a_k)$$

- 0 if no error, or if input a_j from node j is 0.
- **Modified Error**: $\delta_k \equiv g'(in_k)(y_k - a_k)$.
- Gradient Descent Weight Update:

$$w_{jk} \leftarrow w_{jk} + \alpha \times a_j \times \delta_k$$

Error at the output nodes

- First, feed training example $\boldsymbol{x}_n$ forward through the network, storing all node activations a_j
- Calculating derivatives for weights connected to output nodes is easy.
  - like logistic regression with input "features" a_j
- For output node k with activation $a_k = g(in_k) = g(\sum_j w_{jk} a_j)$:

$$\frac{\partial E_n}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} \frac{1}{2}(y_k - a_k)^2 = -a_j \times g'(in_k) \times (y_k - a_k)$$

- 0 if no error, or if input a_j from node j is 0.
- **Modified Error**: $\delta_k \equiv g'(in_k)(y_k - a_k)$.
- Gradient Descent Weight Update:

$$w_{jk} \leftarrow w_{jk} + \alpha \times a_j \times \delta_k$$

Error at the hidden nodes

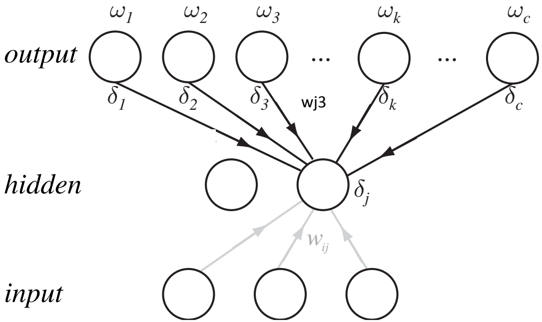
- Consider a hidden node j connected to output nodes.
- The **modified error** signal δ_j is node activation derivative, times the *weighted sum of contributions to the output errors.*
- In symbols,

$$\delta_j = g'(in_j) \sum_k w_{jk} \delta_k .$$

- Weight Update:

$$w_{ij} \leftarrow w_{ij} + \alpha \times a_i \times \delta_j$$

Backpropagation Picture



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

$$\delta_j = g'(in_j) \sum_k w_{jk} \delta_k.$$

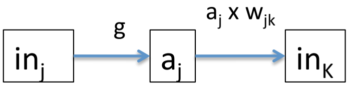
The Backpropagation Algorithm

1. Apply input vector $\boldsymbol{x}_n$ and forward propagate to find all inputs in_i and activation levels 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. Update each weight vector w_{ij}.

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

Outline

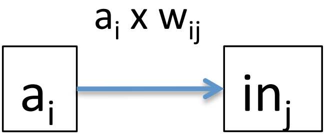
Correctness Proof for Backpropagation Algorithm I.



Exercise: From this functional diagram find expressions for the following quantities:

- $\frac{\partial in_k}{\partial w_{jk}}$
- $\frac{\partial in_k}{\partial a_j}$
- $\frac{\partial in_k}{\partial in_j}$

Correctness Proof for Backpropagation Algorithm II.



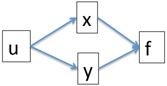
- We need to show that $-\frac{\partial E_n}{w_{ij}} = a_i \times \delta_j$.
- This follows easily given the following result

Theorem

For each node j, we have $\delta_j = -\frac{\partial E_n}{in_j}$.

- Proof given theorem: $-\frac{\partial E_n}{w_{ij}} = -\frac{\partial E_n}{in_j} \cdot \frac{\partial in_j}{\partial w_{ij}} = \delta_j \cdot a_i$.
- Next we prove the theorem.

Multi-variate Chain Rule



- For $f(x, y)$, with f differentiable wrt x and y, and x and y differentiable wrt u and v:

$$\frac{\partial f}{\partial u} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial u} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial u}$$

and

$$\frac{\partial f}{\partial v} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial v} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial v}$$

Proof of Theorem, I

- We want to show that $\delta_j = -\frac{\partial E_n}{a_j}$.
- Think of the error as a function of the activation levels of the nodes *after* node j.
- Formally, we can write $\frac{\partial E_n}{\partial in_j} = \frac{\partial}{\partial in_j} E_n(in_{j_1}, in_{j_2}, \ldots, in_{j_m})$ where $\{j_i\}$ are the indices of the nodes that receive input from j.
- Now using the multi-variate chain rule, we have

$$\frac{\partial E_n}{\partial in_j} = \sum_{k=1}^{m} \frac{\partial E_n}{\partial in_k} \frac{\partial in_k}{\partial in_j}$$

- We saw before that $\frac{\partial in_k}{\partial in_j} = w_{jk} \times g'(in_j)$.

$$\boxed{in_j} \xrightarrow{g} \boxed{a_j} \xrightarrow{a_j \times w_{jk}} \boxed{in_K}$$

Proof of Theorem, I

- We want to show that $\delta_j = -\frac{\partial E_n}{a_j}$.
- Think of the error as a function of the activation levels of the nodes *after* node j.
- Formally, we can write $\frac{\partial E_n}{\partial in_j} = \frac{\partial}{\partial in_j} E_n(in_{j_1}, in_{j_2}, \ldots, in_{j_m})$ where $\{j_i\}$ are the indices of the nodes that receive input from j.
- Now using the multi-variate chain rule, we have

$$\frac{\partial E_n}{\partial in_j} = \sum_{k=1}^{m} \frac{\partial E_n}{\partial in_k} \frac{\partial in_k}{\partial in_j}$$

- We saw before that $\frac{\partial in_k}{\partial in_j} = w_{jk} \times g'(in_j)$.

$$\boxed{in_j} \xrightarrow{g} \boxed{a_j} \xrightarrow{a_j \times w_{jk}} \boxed{in_K}$$

Proof of Theorem, II

- We want to show that $\delta_j = -\frac{\partial E_n}{in_j}$.
- Proof by backward induction. Easy to see that the claim is true for output nodes. (Exercise).
- Inductive step: Consider node j and suppose that $\delta_k = -\frac{\partial E_n}{in_k}$ for all nodes k that receive input from j.
- Using the multivariate chain rule, we have

$$
\begin{aligned}
-\frac{\partial E_n}{\partial in_j} &= \sum_{k=1}^{m} -\frac{\partial E_n}{\partial in_k}\frac{\partial in_k}{\partial in_j} \\
&= \sum_{k=1}^{m} \delta_k \frac{\partial in_k}{\partial in_j} = \sum_{k=1}^{m} \delta_k w_{jk} g'(in_j) = \delta_j.
\end{aligned}
$$

where step 1 applies the inductive hypothesis, step 2 the result from the previous slide, and step 3 the definition of δ_j.

Other Learning Topics

- Regularization: L2-regularizer (weight decay).
- Prune Weights: the Optimal Brain Method.
- Experimenting with Network Architectures is often key.

Outline

Feed-forward Networks

Network Training

Error Backpropagation

Theory: Backpropagation implements Gradient Descent

Applications

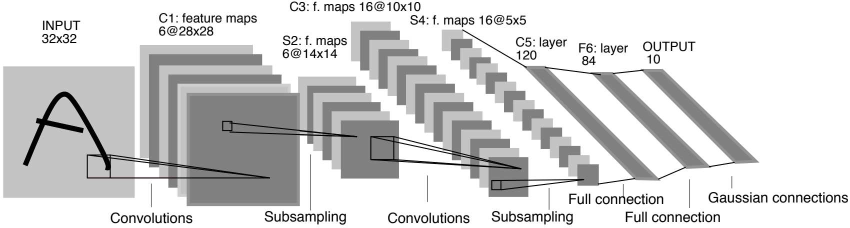
Applications of Neural Networks

- 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 regression or classification
    - Similar to linear models, except with adaptive non-linear basis functions
    - These allow us to do more than e.g. linear decision boundaries
- Different error functions
- Learning is more difficult, error function not convex
    - Use stochastic gradient descent, obtain (good?) local minimum
- Backpropagation for efficient gradient computation