

Artificial Neural Networks

Oliver Schulte

School of Computing Science

Simon Fraser University

Introduction to Deep Learning

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](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://research.microsoft.com/en-us/news/features/speechrecognition-082911.aspx)

- **Translation** <http://translate.google.com>

- <http://deeplearning.net/demos/>

Outline

Feed-forward Networks

Network Training

Error Backpropagation

Theory: Backpropagation implements Gradient Descent

Examples

Outline

Feed-forward Networks

Network Training

Error Backpropagation

Theory: Backpropagation implements Gradient Descent

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 $w_{0,j}$:

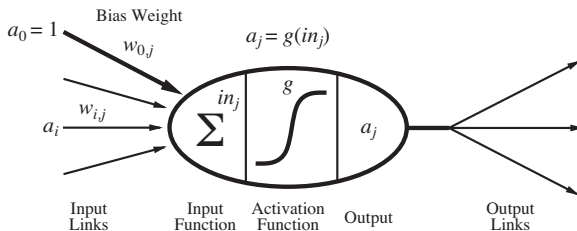
$$in_j = \sum_{i=0}^n w_{ij} a_i$$

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 $w_{0,j}$:

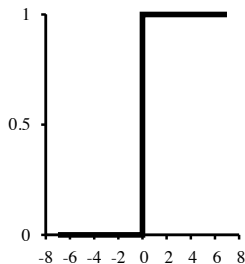
$$in_j = \sum_{i=0}^n w_{ij} a_i$$



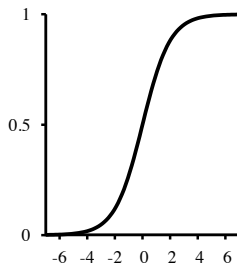
Activation Functions

- 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 $a_j = \sum_i (x_i - w_{ji})^2$
 - 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

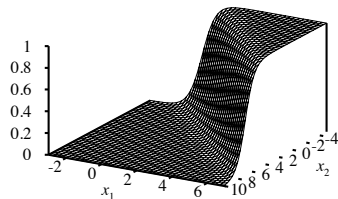
Activation Functions Visualized



Left Threshold

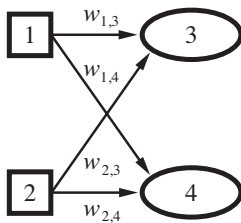


Middle Logistic sigmoid $Logistic(x) = \frac{1}{1+\exp(-x)}$
maps a real number to a probability

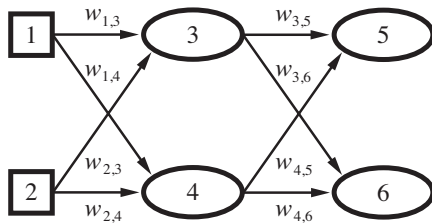


Right Logistic regression $Logistic(\mathbf{w} \bullet \mathbf{x})$

Network of Neurons



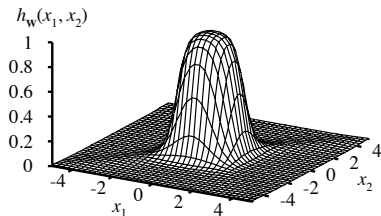
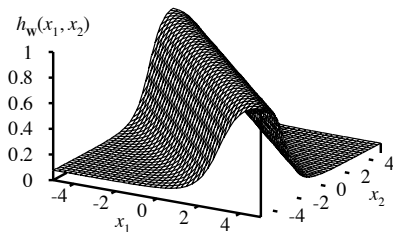
(a)



(b)

Function Composition

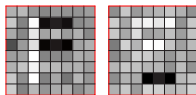
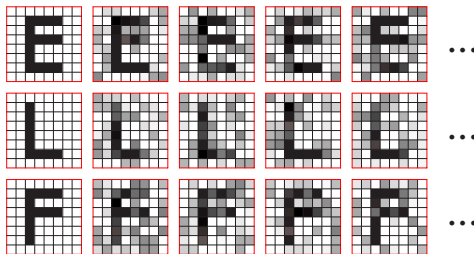
Think logic circuits



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

Hidden Units As Feature Extractors

sample training patterns

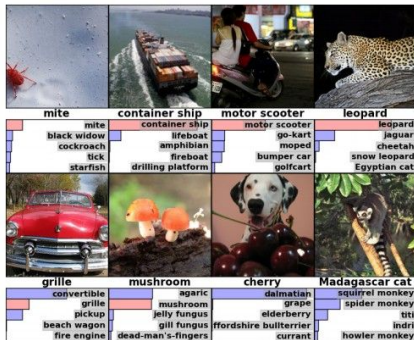


learned input-to-hidden weights

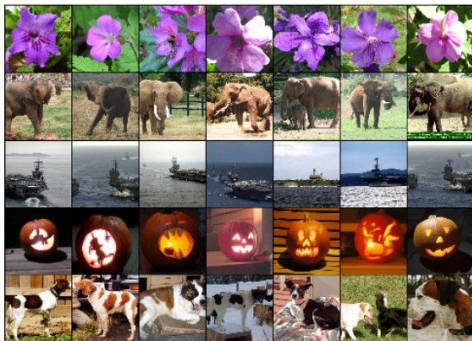
- 64 input nodes
- 2 hidden units
- 2x learned weight vector at hidden unit

Image Analysis Tasks

Classification

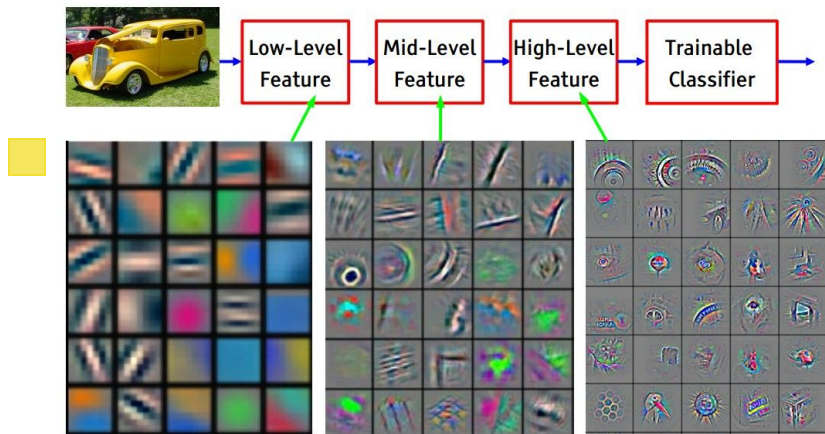


Retrieval



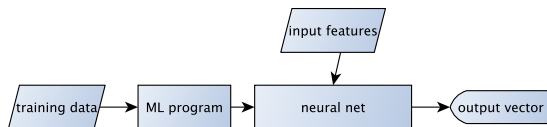
[Krizhevsky 2012]

Neural Net Learned Features



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

Expressive Power



- From the outside perspective, a neural net defines a function from input node values x to output node values.
- Let us write $a_w(x)$ for the output activation vector.
- How complex can a_w be?
- With 1 hidden layer, can represent
 - *exactly* Boolean functions
 - *approximately* continuous functions. e.g. non-convex classification. <http://playground.tensorflow.org>
- With 2 hidden layers, can represent *approximately* an arbitrary function.

Outline

Feed-forward Networks

Network Training

Error Backpropagation

Theory: Backpropagation implements Gradient Descent

Examples

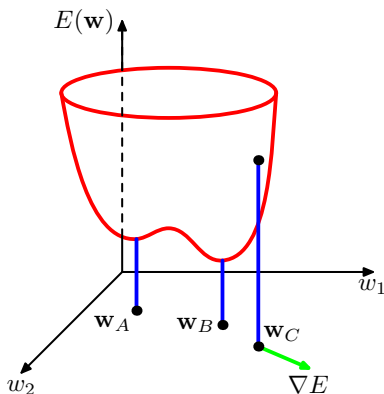
Measuring Training Error

- 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 $(\mathbf{x}_n, \mathbf{y}_n)$
- Corresponds to neural net with multiple output nodes
- Given a set of weight values $\mathbf{w}$, the network defines an output activation vector $\mathbf{a}_{\mathbf{w}}(\mathbf{x})$.
- Given training data $\mathbf{x}, \mathbf{y}$, define
 - output node error $E^k \equiv \ell(a_{\mathbf{w},k}(\mathbf{x}), y_k)$ measures the loss for a single output. Must be differentiable.
 - e.g. $\ell(a_{\mathbf{w},k}(\mathbf{x}), y_k) = 1/2(y_k - a_{\mathbf{w},k})^2$
- The **error function** is the sum of the individual output node errors $E(\mathbf{w}, \mathbf{x}) = \sum_k E^k$.

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 $(\mathbf{x}_n, \mathbf{y}_n)$
- Corresponds to neural net with multiple output nodes
- Given a set of weight values $\mathbf{w}$, the network defines an output activation vector $\mathbf{a}_{\mathbf{w}}(\mathbf{x})$.
- Given training data $\mathbf{x}, \mathbf{y}$, define
 - output node error $E^k \equiv \ell(a_{\mathbf{w},k}(\mathbf{x}), y_k)$ measures the loss for a single output. Must be differentiable.
 - e.g. $\ell(a_{\mathbf{w},k}(\mathbf{x}), y_k) = 1/2(y_k - a_{\mathbf{w},k})^2$
- The **error function** is the sum of the individual output node errors $E(\mathbf{w}, \mathbf{x}) = \sum_k E^k$.

Parameter Optimization



- For typical cases, the error function $E(\mathbf{w}) \equiv \sum_x E(\mathbf{w}, \mathbf{x})$ is nasty
 - Nasty = non-convex
 - Non-convex = has **local minima**

Gradient Descent

- The function $a_w(x)$ 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

Outline

Feed-forward Networks

Network Training

Error Backpropagation

Theory: Backpropagation implements Gradient Descent

Examples

Notice of Notation

- In the following we consider a backpropagation training step for a fixed example $(\mathbf{x}_n, \mathbf{y}_n)$ and fixed current weights $\mathbf{w}$.
- We adapt our notation by omitting these arguments unless needed.
- Examples:
 - E_n instead of $E(\mathbf{w}, \mathbf{x}_n)$
 - a_k instead of $a_{\mathbf{w},k}(\mathbf{x})$

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 $\mathbf{x}_n$ forward through the network, storing *all* node activations a_i
- Calculating derivatives for weights connected to output nodes is easy.
- For output node k with activation $a_k = g(in_k) = g(\sum_i w_{ik}a_i)$ and target value y_k the error signal is

$$\Delta[k] \equiv -g'(in_k) \frac{\partial \ell(a_k, y_k)}{\partial a_k}.$$

- E.g. for squared error loss $\Delta[k] = g'(in_k)(y_k - a_k)$
- Gradient Descent Weight Update:

$$w_{ik} \leftarrow w_{ik} + \alpha \times a_i \times \Delta[k]$$

Error at the output nodes

- First, feed training example $\mathbf{x}_n$ forward through the network, storing *all* node activations a_i
- Calculating derivatives for weights connected to output nodes is easy.
- For output node k with activation $a_k = g(in_k) = g(\sum_i w_{ik} a_i)$ and target value y_k the error signal is

$$\Delta[k] \equiv -g'(in_k) \frac{\partial \ell(a_k, y_k)}{\partial a_k}.$$

- E.g. for squared error loss $\Delta[k] = g'(in_k)(y_k - a_k)$
- Gradient Descent Weight Update:

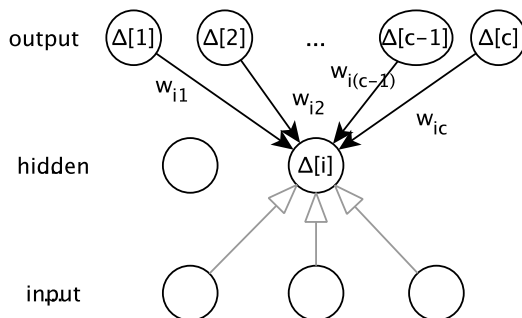
$$w_{ik} \leftarrow w_{ik} + \alpha \times a_i \times \Delta[k]$$

Error at the hidden nodes

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

$$\Delta[i] = g'(in_i) \sum_{j=1}^c w_{ij} \Delta[j].$$

The Backpropagation Algorithm

1. Apply input vector 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 Alspace <http://aispace.org/neural/>.

Outline

Feed-forward Networks

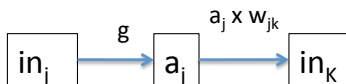
Network Training

Error Backpropagation

Theory: Backpropagation implements Gradient Descent

Examples

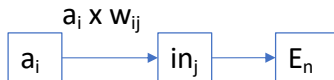
Correctness Proof for Backpropagation Algorithm I.



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

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

Correctness Proof for Backpropagation Algorithm II.



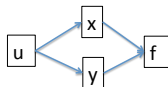
- We need to show that $-\frac{\partial E_n}{\partial w_{ij}} = \Delta[j] \cdot a_i$.
- This follows easily given the following result

Theorem

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

- Proof given theorem: $-\frac{\partial E_n}{\partial w_{ij}} = -\frac{\partial E_n}{\partial in_j} \cdot \frac{\partial in_j}{\partial w_{ij}} = \Delta[j] \cdot a_i$.
- 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 $\Delta[j] = -\frac{\partial E_n}{\partial in_j}$.
- Think of the error as a function of the activation levels of the nodes *after* node j .
- Formally, we can write $\frac{\partial E_n}{\partial in_j} = \frac{\partial}{\partial in_j} E_n(in_{k_1}, in_{k_2}, \dots, in_{k_m})$ where $\{k_i\}$ are the indices of the nodes that receive input from j .
- Now using the multi-variate chain rule, we have

$$\frac{\partial E_n}{\partial in_j} = \sum_k \frac{\partial E_n}{\partial in_k} \frac{\partial in_k}{\partial in_j}$$

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

Proof of Theorem, I

- We want to show that $\Delta[j] = -\frac{\partial E_n}{\partial in_j}$.
- Think of the error as a function of the activation levels of the nodes *after* node j .
- Formally, we can write $\frac{\partial E_n}{\partial in_j} = \frac{\partial}{\partial in_j} E_n(in_{k_1}, in_{k_2}, \dots, in_{k_m})$ where $\{k_i\}$ are the indices of the nodes that receive input from j .
- Now using the multi-variate chain rule, we have

$$\frac{\partial E_n}{\partial in_j} = \sum_k \frac{\partial E_n}{\partial in_k} \frac{\partial in_k}{\partial in_j}$$

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

Proof of Theorem, II

- We want to show that $\Delta[j] = -\frac{\partial E_n}{\partial in_j}$.
- Proof by backward induction. Easy to see that the claim is true for output nodes. (Exercise).
- Inductive step: Consider node j and suppose that $\Delta[k] = -\frac{\partial E_n}{\partial in_k}$ 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 2 applies the inductive hypothesis, step 3 the result from the previous slide, and step 4 the definition of $\Delta[j]$.

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

Feed-forward Networks

Network Training

Error Backpropagation

Theory: Backpropagation implements Gradient Descent

Examples

Applications of Neural Networks

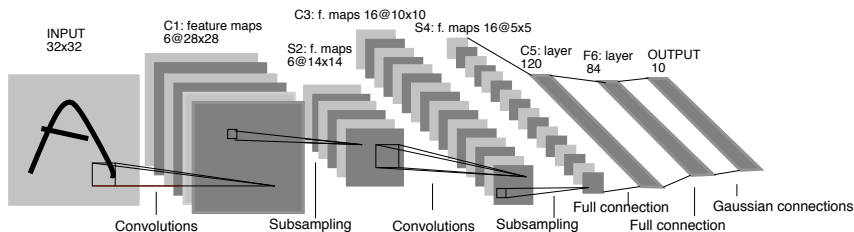
- Many success stories for neural networks
 - Credit card fraud detection
 - Hand-written digit recognition
 - Face detection
 - Autonomous driving (CMU ALVINN)

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.