Computing Science (CMPUT) 455

Search, Knowledge, and Simulations

James Wright

Department of Computing Science
University of Alberta
james.wright@ualberta.ca

Fall 2021

455 Today - Lecture 19

- Using learned models in UCT
- Introduction to Neural Networks (NN)
- Examples
- Learning with NN - Backprop
- Types of (artificial) neural networks
- NN as universal function approximators

Coursework

- Assignment 3: late submission deadline was last night
  - Grades available by the end of the weekend
- Lecture 19 activities:
  - Videos and demos for neural nets
- Quiz 10: Machine learning with simple features (Double-header)

Recap

- Learning with simple features
- Coulom's approach:
  - Generalized Bradley-Terry model for strength of moves
  - MM algorithm for learning weights

Using Knowledge in UCT

- Regular UCT: select best child by UCT formula
- UCT value of move *i* from parent *p*:

$$UCT(i) = \hat{\mu}_i + C\sqrt{\frac{\log n_p}{n_i}}$$

- This uses only information from simulations
  - Empirical winrate $\hat{\mu}_i$, number of simulations n_i, number of simulations for parent n_p
- We can improve move selection by using learned knowledge
  - Examples: simple features, neural networks
- Idea: give good moves a bonus before simulations start

How to Use Knowledge

Three ways:

1. Initialization of node statistics
2. Additive knowledge term
3. Multiplicative knowledge term

Decay Knowledge over Time

- At the beginning, we have only few simulations
  - Win rate $\hat{\mu}_i$ is very noisy
  - Knowledge may be more reliable, can help to guide search
- Later, we may have many simulations for a node
  - We should trust them more now
  - All knowledge is heuristic, may be wrong
  - Slowly phase out knowledge as more simulations accumulate

1. Initialization of Node Statistics

- Normal UCT: count number of simulations and wins
- Initialize to 0
  - For all children i
  - Wins $w_i = 0$
  - Simulations $n_i = 0$
- We can initialize with other values to encode knowledge about moves
  - Give good moves some imaginary initial "wins"
  - Give bad moves some imaginary initial "losses"

1. Initialization of Node Statistics (2)

- How to initialize n_i and w_i ?
- Size of n_i expresses how reliable the knowledge is
- Winrate w_i/n_i expresses how good or bad the move is, according to the knowledge
- Original work by Gelly and Silver (2007): knowledge worth up to 50 simulations
- Fuego program: simple feature knowledge converted into winrate/simulations
- Decay over time: yes
  - Over time, real simulation statistics dominate over initialization

2. Additive Knowledge

- Idea: add a term to UCT formula

$$UCT(i) = \hat{\mu}_i + \textbf{knowledgeValue}(i) + C\sqrt{\frac{\log n_p}{n_i}}$$

- `knowledgeValue(i)` computed e.g. from simple features or neural network
- Must scale it relative to other terms by tuning
  - Too small: little influence on search
  - Too big: too greedy, ignores winrate
- Decay over time: must be explicitly programmed
- Multiply knowledge term by some *decay factor*
  - Examples: $1/(n_i + 1)$, $\sqrt{1/(n_i + 1)}$,...

3. Multiplicative Knowledge, Probabilistic UCT (PUCT)

- Idea: explore promising moves more
- Knowledge used:
  - Probability p_i that move i is best
- Multiply exploration term by p_i

$$PUCT(i) = \hat{\mu}_i + \mathbf{p_i} \times C\sqrt{\frac{\log n_p}{n_i}}$$

- Decay over time: yes
  - Divide by n_i in the exploration term
- Exploration term smaller than before, because $p_i \leq 1$
  - May need to balance by increasing C
- AlphaGo: exploration term $p_i \times C/(n_i + 1)$

Summary of Knowledge in UCT

- Knowledge can be used in an in-tree selection formula
- Independent from using knowledge during the simulation phase
- Can be (much) slower, used only in tree nodes, not in each simulation step
- Different approaches have been tried successfully
    1. Initialization of node statistics by knowledge
    2. Additive term
    3. Multiplicative term, PUCT

Outline

- Introduction to Neural Networks (NN)
- Artificial neural networks in computing science
- Neural networks as function approximators
- Learning weights for NN - Backpropagation

Neural Networks

- A neural network in Computing Science is a *function*

$$y = f(x; w)$$

- It takes input (x) and produces outputs (y)
- It has many parameters (weights w) which are determined by learning (training)
- Deep neural networks can approximate (almost) any function in practice
- Training NN:
  - Supervised learning
  - Reinforcement learning

Neural networks in Biology - Neurons

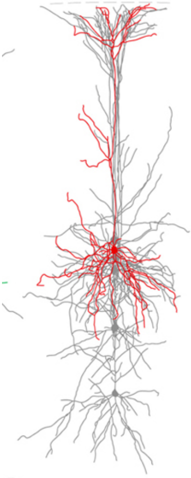


Image source:

`http://www.frontiersin.org/files/Articles/62984/fncel-07-00174-r2/`

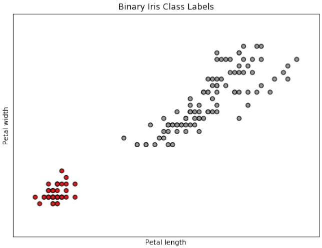
- Neuron = nerve cell
- Found in:
  - Central nervous system (brain and spinal cord)
  - Peripheral nervous system (nerves connecting to limbs and organs)
- Involved in all sensing, movement, and information processing (thinking, reflexes)
- Very complex systems, function is still only partially understood

Neural Networks (NN) in Computing Science

- Massively simplified, abstract model
- Used as a powerful function approximator for (almost) arbitrary functions
- We now have effective learning algorithms even for very large and deep networks
- Single (artificial) neuron: implements a simple mathematical function from its inputs to its output
- Connections between neurons:
  - Each connection has a *weight*
  - Expresses the strength of the connection

Binary Classification Example

- Consider the binary classification problem
- We want to draw a line between the classes
- For a problem with two features, the equation becomes
  $z = \operatorname{sgn}(w_1 x_1 + w_2 x_2 + b)$
    - x_1, x_2 are the input features
    - z is the output (class value)
    - sgn is the sign operator
    - w_1, w_2 are the feature weights
    - b is the bias term
- Find w_1, w_2 and b such that the line can separate the classes clearly



The Perceptron: A Single Neuron

- Inputs $x_1 \ldots x_m$ (from m neurons on previous layer)
- Extra constant input $x_0 = 1$
- Each input x_i has a weight w_i
- Weighted sum of inputs $\sum_{i=0}^{m} w_i x_i$
- Nonlinear activation function (or transfer function) ϕ
- Output $y = \phi(\sum_{i=0}^{m} w_i x_i)$
- Output used as input for neurons on next layer

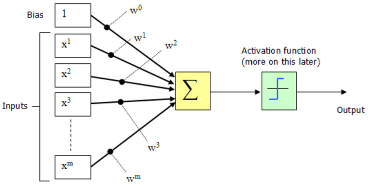


Image source: `https://www.codeproject.com/KB/AI/NeuralNetwork_1/nn2.png`

Components of a NN - Input, Output and Hidden Layers

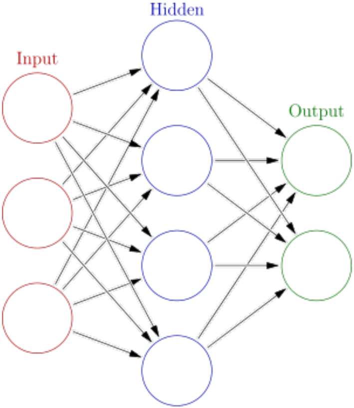


Image source: `https://en.wikipedia.org/wiki/Artificial_neural_network`

- Organized in layers of neurons
- Each layer is connected to the next
- Input layer
- One or more hidden layers
- Output layer
- Shallow vs Deep NN
  Main difference:
  Number of hidden layers

Supervised Training of a Network - Overview

- View the whole network as a function $y = f(x)$
- Both x and y are vectors of numbers
- Train by supervised learning from set of data (x_j, y_j)
- Compute errors - differences between y_j and $f(x_j)$
- Compute how error depends on each weight w_i in network
- Gradient descent - adjust weights w_i in network to reduce these errors
- Example now, details later

Software: NN Toy Examples in Python

- First example: `nn.py` in `python/code`
- Adapted from article at `http://iamtrask.github.io/2015/07/12/basic-python-network`
- 1 input layer, 1 hidden layer, 1 output node
- 3 input nodes - Each input x_i consists of three values
- Training data: 4 examples
- Input: 4 rows, 1 for each x_i, $i = 0, 1, 2, 3$
- Sigmoid activation function (see next slide)
- Output vector with 4 numbers y_i

Sigmoid Function

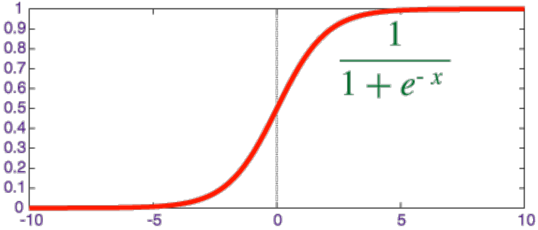


Image source: `https://qph.ec.quoracdn.net`

- Nonlinear function, popular for activation function
- Smoothly grows from 0 to 1
- Definition:

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

Properties of Sigmoid Function

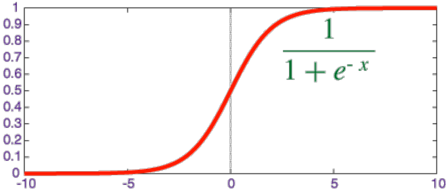


Image source: `https://qph.ec.quoracdn.net`

- x large negative number:
  e^{-x} very large, $\sigma(x)$ close to 0
- x large positive number:
  e^{-x} very small, $\sigma(x)$ close to 1
- $x = 0$: $\sigma(x) = 1/2$
- Nice property of $\sigma(x)$: derivative

$$\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$$

Backpropagation and Training - Error

- Same basic ideas as learning with simple features
- Let f be the function computed by the net
- Result of f depends on
  - input vector x
  - all weights w_j
- Output $y = f(x, w_0, ...w_n)$
- Error on data point (x_i, y_i):
  - Difference between $f(x_i)$ and y_i
  - Usual measure - squared error $(y_i - f(x_i))^2$
- Goal: minimize sum of square errors over training data
- Error $E = \sum_i (y_i - f(x_i))^2$

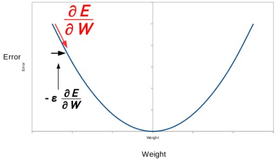
Backpropagation Concepts

- How to reduce error?
- The only thing we can change are the weights w_i
- How does error E depend on all the weights?
- Simpler question: how does error E depend on a single weight w_i?
- Should we increase w_i, decrease it, or leave it the same?
- The *partial derivative* of E with respect to w_i gives the answer

$$\frac{\partial E}{\partial w_i}$$

Partial Derivative - Intuition

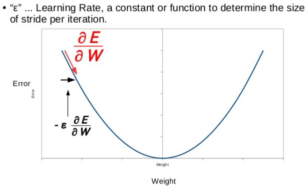
- "ε" ... Learning Rate, a constant or function to determine the size of stride per iteration.



- Meaning of $\frac{\partial E}{\partial w_i}$
- Make a small change of w_i
- How does it affect the error E?
- Which change will *reduce* the error?
- Look at sign of derivative

- $\frac{\partial E}{\partial w_i} > 0$ - Small **decrease** in w_i will decrease E
- $\frac{\partial E}{\partial w_i} = 0$ - Small change in w_i will have no effect on E
- $\frac{\partial E}{\partial w_i} < 0$ - Small **increase** in w_i will decrease E

Partial Derivative and Rate of Change



- Error E is a function of
  all inputs x, all outputs y and all weights w
- Partial derivative quantifies the effect of
  leaving everything else constant
  and making a small change ϵ to w_i
- $E(\cdots, w_i + \epsilon, \cdots) \approx E(\cdots, w_i, \cdots) + \frac{\partial E}{\partial w_i}\epsilon$

Derivative and Chain Rule

- How does the error E change if we change *any* single weight in the net?
- We can break down the computation layer by layer
- The error function is a simple function of the output
- The output is the result from the last layer in the net
- Each node implements a simple function of its inputs
- The inputs are again simple functions of the previous layer, etc.
- We can break down the computation of $\frac{\partial E}{\partial w_i}$ into a neuron-by-neuron computation using the chain rule

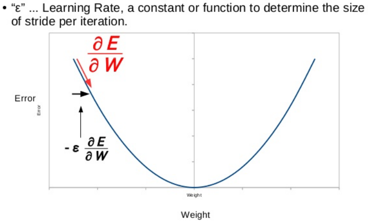
Chain Rule

- $z = f(x)$, $y = g(z) = g(f(x))$
- Then
$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial z} \times \frac{\partial z}{\partial x}$$
- Example:
- Neuron input
$z = \sum_{i=0}^{m} w_i x_i$
- Sigmoid activation function
$y = \sigma(z) = \sigma(\sum_{i=0}^{m} w_i x_i)$
- How does output y depend on some weight, say w_1?

Chain Rule Example Continued

- Example - compute derivative of y with respect to w_1, $\frac{\partial y}{\partial w_1}$
- By chain rule, $\frac{\partial y}{\partial w_1} = \frac{\partial y}{\partial z} \times \frac{\partial z}{\partial w_1}$
- First, derivative of z with respect to w_1, $\frac{\partial z}{\partial w_1}$
  - z is just a linear function of w_1
  - $z = w_1 x_1 +$ (terms that do not depend on w_1)
  - $\frac{\partial z}{\partial w_1} = x_1$
- Now, $\frac{\partial y}{\partial z} = \frac{\partial \sigma(z)}{\partial z}$
- Remember $\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$
- So $\frac{\partial y}{\partial z} = \sigma(z)(1 - \sigma(z))$
- Result: $\frac{\partial y}{\partial w_1} = \sigma(z)(1 - \sigma(z)) \times x_1 = y(1 - y)x_1$
- Final result is simple, easy to compute
- In practice, packages such as PyTorch, TensorFlow, etc. can do all of the math automatically

Backpropagation (Backprop) Step



- Apply chain rule to compute how changes to weights reduce error
- Go some distance ϵ along the *gradient* of E with respect to weights
- $w_i = w_i - \epsilon \frac{\partial E}{\partial w_i}$
- Choice of *step size* ϵ is important
- Go too far - overshoot the minimum
- Go too little - very slow improvement of E

Backprop Algorithms

- Developed starting in the 1960's
- Main ideas
- Define step size ϵ
- Compute backprop step for *all* weights
- Repeat until error on test set does not improve
- Huge number of variations of backprop algorithms
  - Momentum, adaptive step size, stochastic vs batch data, ...

Network Types

- Feed-forward NN (all our examples)
  - Information flows in one direction from input to output
- Recurrent NN (RNN)
  - Directed cycles in the network
  - Popular in natural language processing, speech and handwriting recognition
  - Example of very successful deep RNN architecture: LSTM, “Long short-term memory”
    - Can be trained by backprop, like our feed-forward nets
- Autoencoder - learn representation for data with unsupervised learning
- Hundreds of other NN types, new ones each month

Building a Neural Network

Important Questions:

- How many layers?
- How to connect the layers
- How many neurons in each layer?
- What kind of functions can we represent in principle?
- What kind of functions can we learn efficiently?

Neural Networks as Universal Approximators

- NN with at least one hidden layer can *approximate* any *continuous* function arbitrarily well, given enough neurons in the hidden layer
- Given a continuous function $f(x)$
- Consider $f(x)$ in the range $0 \leq x \leq 1$
- Given an arbitrarily small $\epsilon > 0$
- Theorem (Cybenko 1989)
  There exists a 1-hidden-layer NN $g(x)$ such that

$$|f(x) - g(x)| < \epsilon \quad \text{for all} \quad 0 \leq x \leq 1$$

NN as Universal Approximators (2)

- How is that possible?
- Intuitively, it works by:
  - Having lots of neurons in the hidden layer
  - Two neurons together can approximate a *step function*
  - Their sum is very close to $f(x)$ in a tiny interval
  - Their sum is almost 0 everywhere else
- Demo from `http://neuralnetworksanddeeplearning.com/chap4.html`
- Note: constant b in demo is what we called w_0

NN as Universal Approximators (3)

Comments:

- The theorem does *not* mean that any network can approximate any function arbitrarily well
- The theorem says that by *adding* more and more hidden neurons, we can make the error smaller and smaller
- The theorem is only about *continuous* function. But we can also approximate functions with discontinuous jumps pretty well

NN as Universal Approximators (4)

More comments:

- Why are we using multilayer “deep” networks if 1 hidden layer is enough in theory?
- Short answers:
  - Efficiency of learning
  - Size of representation
- Details: `http://neuralnetworksanddeeplearning.com/chap5.html`

Network Architecture - fully connected

- Review - usually, connections are only from one layer to the next
- Some recent success with adding connections to layers "further up" (not discussed here)
- Simplest architecture: *fully connected*
  - Each neuron on layer n connected to each neuron on layer $n+1$

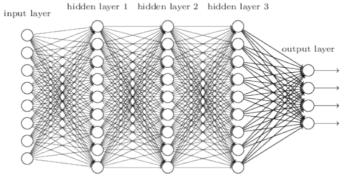


Image source: `http://neuralnetworksanddeeplearning.com/chap6.html`

Sparse Network Architectures

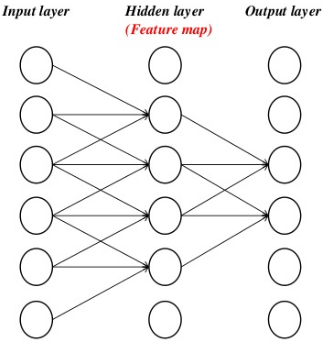


Image source: `https://www.slideshare.net/SeongwonHwang/presentations`

- Opposite of fully connected: *sparse*
- Neuron connected to only *some* neurons on next layer
- Important case for us: *Convolutional* NN (next lecture)

Summary

- Introduced neural networks
- Backprop algorithm
- Examples of networks
- Next time: convolutional networks, deep networks
- Move prediction in Go with deep convolutional networks