# Computing Science (CMPUT) 455

## Search, Knowledge, and Simulations

Martin Müller

Department of Computing Science
University of Alberta
mmueller@ualberta.ca

Fall 2022

# 455 Today - Lecture 19

- Introduction to Neural Networks (NN)
- Examples
- Learning with NN - Backprop
- Types of (artificial) neural networks
- NN as universal function approximators
- Demo - NN for TicTacToe

# Coursework

- Lecture 19 activities:
  - Videos and demos for neural nets
- Quiz 11: Neural Networks and Deep Neural Networks (double length)

# Recap

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

# 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

- Lecture 19 activities:
  - Videos and demos for neural nets
- Quiz 11: Neural Networks and Deep Neural Networks (double length)

# Recap

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

# Outline

- Introduction to Neural Networks (NN)
- Artificial neural networks in computing science
- Neural networks as function approximators
- Learning weights for NN - Backpropagation
- Example: training a neural net to play TicTacToe

# 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

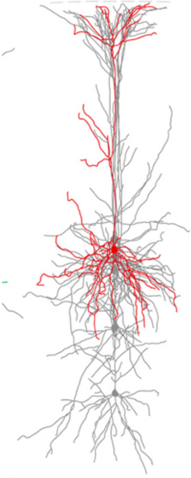


Image source:

`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 in Biology - Neurons

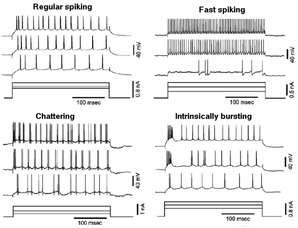



Image source:

`http://ecee.colorado.edu/`

`~ecen4831/cnsweb/cns0.html`

- Neurons transmit information through electrical and chemical signals
- Transmission through synapses - connections between two neurons
- Complex behaviors:
    - In time
    - In space

# Synapses

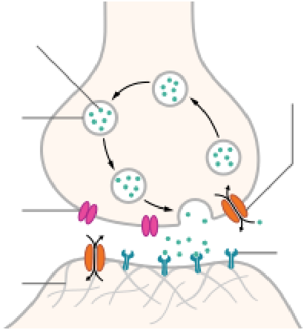


Image source:

`https://en.wikipedia.org/wiki/Synapse`

- Small (atom-scale) gap between neurons
- Information transmitted via
  - Chemicals (neurotransmitters, main mechanism)
  - Electric currents (faster)
- Human brain - about 150 trillion ($1.5 \times 10^{14}$) synapses
- Some neurons have up to 100000 synapses

# Neuron Count in Humans and Animals

- Elephant 251 billion
- **Human 86 billion**
- Gorilla 33 billion
- Baboon 14 billion
- Raven 2.2 billion
- Cat 760 million
- Rat 200 million
- Frog 16 million
- Cockroach 1 million
- Fruit fly 250,000
- Ant 250,000
- Jellyfish 5600
- Worm 300
- Sponge 0

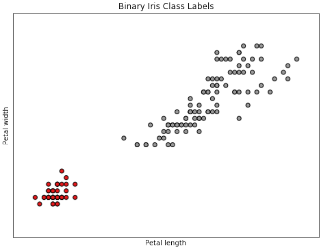
Source: `https://en.wikipedia.org/wiki/List_of_animals_by_number_of_neurons`

# 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

- Binary classification - separate red and grey points by a line
- Each item described by two features $x_1$ and $x_2$
- Compute classifier:
  $z = \text{sgn}\,(w_1 x_1 + w_2 x_2 + b)$
  - $z$ is the output (class value)
  - sgn is the sign operator - +1 or -1
  - $w_1, w_2$ are the feature weights
  - $w_0$ is the bias term
- Find $w_1, w_2$ and $w_0$ 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) $\phi$
- Output $y = \phi(\sum_{i=0}^{m} w_i x_i)$
- Output used as input for neurons on next layer

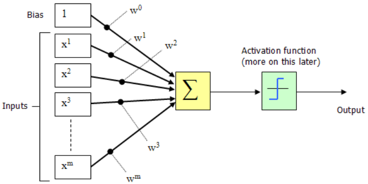



Image source: `https://www.codeproject.com/KB/AI/NeuralNetwork_1/nn2.png`

# Components of a NN - Input, Output and Hidden Layers

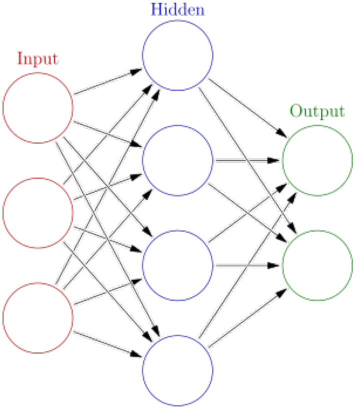



Image source: `https://en.wikipedia.org/wiki/Artificial_neural_network`

- 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

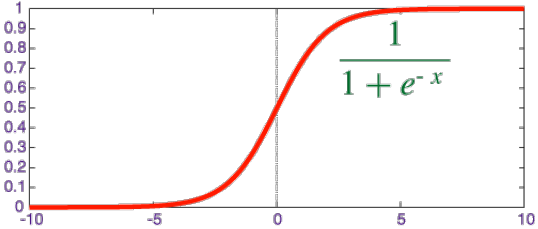



Image source: `https://qph.ec.quoracdn.net`

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

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

# Properties of Sigmoid Function

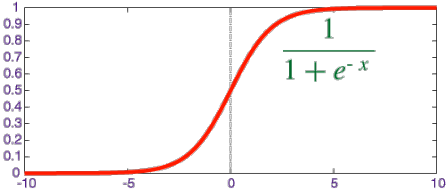



Image source: `https://qph.ec.quoracdn.net`

- $x$ 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 $E = \sum_i (y_i - f(x_i))^2$

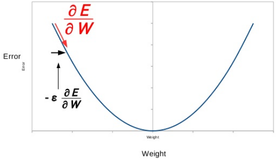
# Backpropagation Concepts

- 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

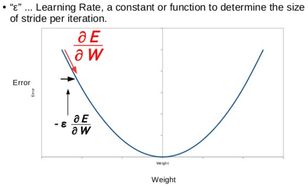
- "ε" ... Learning Rate, a constant or function to determine the size of stride per iteration.




- 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 $\epsilon$ 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

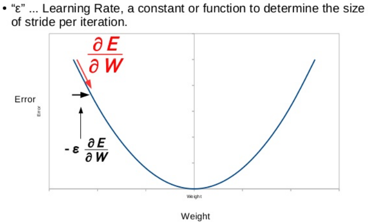
# Chain Rule

- $z = f(x)$, $y = g(z) = g(f(x))$
- 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 for you

# Backpropagation (Backprop) Step




- Apply chain rule to compute how changes to weights reduce error
- Go some distance $\epsilon$ along the *gradient* of *E* with respect to weights
- $w_i = w_i - \epsilon \frac{\partial E}{\partial w_i}$
- Choice of *step size* $\epsilon$ 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 $\epsilon$
- 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 $n+1$

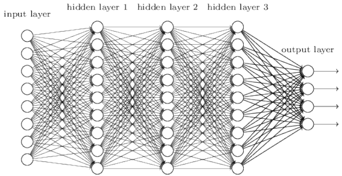



Image source: `http://neuralnetworksanddeeplearning.com/chap6.html`

# Sparse Network Architectures

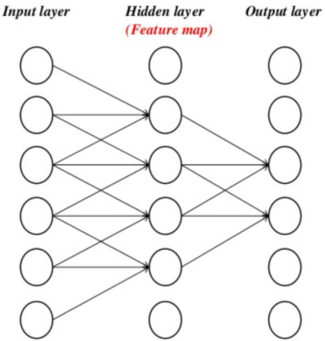



Image source: `https://www.slideshare.net/SeongwonHwang/presentations`

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

# Example: Training a Neural Net for TicTacToe

- See python code for Lecture 19
- Train a neural net for TicTacToe
- Learns from a database of all solved TicTacToe positions
- Achieves perfect or close to perfect play after training
- You can train your own net, or use a net already trained

# Software: multilayer_perceptron.py

- Multilayer Perceptron (MLP)
- A type of simple feed-forward neural network
- The whole network is a function $y = f(x)$
- $x$ is the input state - a TicTacToe position
- $y = f(x)$ predicts win/loss/draw probabilities for $x$
- $f$ predicts the correct (minimax optimal) winner of any given TicTacToe position

# Network Architecture

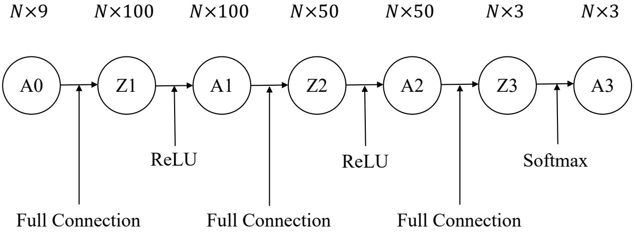



Image source: Henry Du

- Input layer A0
- First hidden layer Z1, A1
  - Z and A together implement the computation for one layer of neurons
  - Z computes the weighted sum $z = \sum_{i=0}^{m} w_i x_i$
  - A computes the activation function, $a = \phi(z)$
- 2nd hidden layer Z2, A2
- Output layer Z3, A3
- Batch size $N$ (see later slides)

# Input Layer

- 9 neurons
- Flatten the board into one vector of size 9
- One input for each point on the board
- Batch training:
  - Batch size $N$ - number of samples fed into the neural network together
  - Input: Stack $N$ vectors to form a $N \times 9$ matrix
  - $N$ rows
  - One row (of length 9) for each of $N$ input boards

# Two Hidden Layers

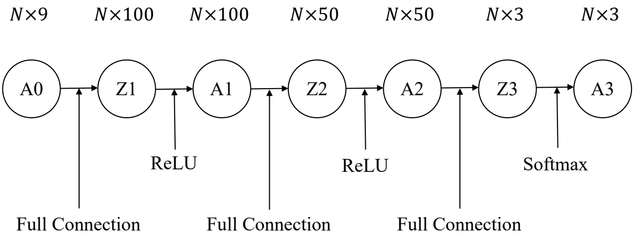



Image source: Henry Du

- First hidden layer: 100 neurons
- Second hidden layer: 50 neurons
- Fully connected to input, between hidden layers, and to output

# Output Layer

- Three neurons represent the probability of win/draw/loss
- The values of these 3 neurons add up to 1
- Output size $N \times 3$, depends on the batch size $N$ of the input layer
- 3 probabilities for each input board

# Software: tic_tac_toe_train_nnet.py

- Trains the MLP
- Three training methods/optimizers implemented:
  - Gradient Descent (called GD in experiments)
  - Stochastic Gradient Descent with mini-batch (mini_batch)
  - Modified Stochastic Gradient Descent (modified)

# Gradient Descent Algorithm

- At each iteration
- Perform forward and backward pass on the whole dataset
- Downsides:
  - Computationally very expensive on large dataset
  - More likely to be trapped in local minimum
  - Slow learning, poor accuracy here

# Stochastic Gradient Descent

- At each iteration, randomly select one sample
- Do forward and backward pass
- Much faster per iteration
- Randomness makes it less likely to be trapped in local minimum
- Downside: needs many more iterations to observe the whole dataset

# Stochastic Gradient Descent (SGD) with Mini-Batch

- Between Gradient Descent and SGD
- Perform forward and backward pass on $N$ samples
- In each *epoch:*
- Shuffle the whole dataset
- Divide the dataset into batches
- Train the neural net on each batch
- The model observes the whole dataset in one epoch

# Modified Stochastic Gradient Descent with Mini-Batch

- Often used in practice
- In each iteration:
- Instead of dividing the dataset into batches, sample N items randomly to form a batch
- A sample may appear in more than one batch, or never
- More randomness in training, often faster to learn, higher accuracy

# Network Training Results

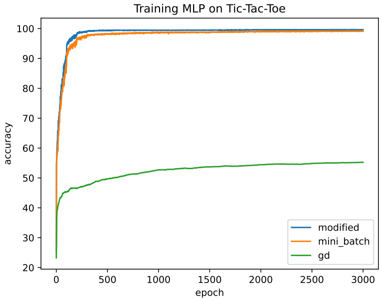



Image source: Henry Du

- Both SGD methods work well
- Both reach optimal or almost-optimal play
- GD learns much more slowly
- Still poor after 3000 epochs

# Summary

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