**Artificial Intelligence**

**Session 7**

1. Brain doesn’t seem to have a CPU
   1. Many simple, parallel, asynchronous units- neurons
   2. Each neuron is a single cell, with
      1. some relatively short fibres, called dendrites
      2. one long fibre, called an axon
   3. End of axon branches out into more short fibres
   4. Each fibre “connects” to dendrites and cell bodies of other neurons
      1. Connection is actually a tiny gap, called a synapse
   5. Axons are transmitters, dendrites are receivers
2. Neuron: Fibres of surrounding neurons emit chemicals (neurotransmitters)
   1. Neurotransmitters move across synapse and change electrical potential applied to cell body
   2. Sometimes transmission across synapse increases or decreases the potential
   3. When potential reaches threshold,
      1. electrical pulse (action potential) travels down axon
      2. eventually reaching all the branches
      3. causing them to release their neurotransmitters
   4. And so on through the network
3. Neural architecture: neuroplasticity
   1. Neuroplasticity or brain plasticity is brain’s ability to change during life
      1. Can reorganise itself by forming new connections between brain cells (neurons)
   2. Neuroplasticity occurs
      1. At beginning of life: when immature brain organises itself
      2. After brain injury: to compensate for lost functions or maximise remaining ones
      3. Throughout life: whenever something new is learned and memorised
      4. When you become a domain expert, brain areas that deal with this skill will grow
      5. For instance, London taxi drivers have larger posterior hippocampus (region used for complex spatial reasoning) than London bus drivers
   3. Learning involves both adjustment of strength of connections as well as establishment of new connections between neurons.

|  |  |
| --- | --- |
| **Biological Neural Network** | **Artificial Neural Network** |
| Soma | Perceptron |
| Dendrite | Input |
| Axon | Output |
| Synapse | Weight |

1. Perceptrons
   1. Many connectionists use same neural model
   2. Collection of units, each of which has
      1. some weighted inputs from other units
         1. inputs represent degree to which other units active (firing)
         2. weights represent how much unit affected by activity of other units
      2. threshold that sum of weighted inputs exceeds to cause firing
      3. single output that connects to inputs of other units
         1. if unit fires, resulting action potential goes this way  
              
            Fig:  
            Each input node corresponds to a weights node. All the weight node point to a “Weighted Sum” node which, in turn, points to a step function (output).
   3. x\_i are inputs (i < n), reals between 0 and 1 (standard net) or -1 and 1 (bipolar net)
   4. w\_i are weights, reals
   5. w\_n is usually used for threshold, with x\_n = 1(bias)
   6. s is the weighted sum of inputs including the threshold (activation level)
   7. h is the output
      1. Output computed using function g(.) that reflects how far perceptron’s activation level below or above 0
   8. Loose approximation to what happens in biology
      1. Perceptrons transmit information via real-valued numbers as inputs arrive
         1. real neurons fire all the time, changing their firing rate, from few pulses per second to few hundred pulses per second
         2. spiking neural networks simulate this directly—beyond scope of current module
      2. Learning in neural network achieved by adjusting weights
         1. Architecture doesn’t change during learning
         2. But weights in perceptrons are not fixed, can change
   9. Basic perceptron computes:  
        
      h = g( X ⋅ W )

Where:  
X ⋅ W = sum of w\_i \* x\_i

* + 1. g is activation or transfer function
    2. simple step function: g(s) = 1 if s >= 0.5 and 0 otherwise, or
    3. more complex, continuous functions (esp. sigmoid) can be used
       1. all have same purpose: push output values towards extremes (0 and 1)
    4. without this nonlinearity, network can only learn simple linear functions
    5. sigmoid function is differentiable, whereas step function is not
  1. **Hardwiring a perceptron**:

Perceptrons can simulate basic logic gates, such as AND:

|  |  |  |  |
| --- | --- | --- | --- |
| **a** | **b** | **a+b-1** | **output** |
| 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 0 | -1 | 0 |

* + 1. Changing one weight gives an OR gate:

|  |  |  |  |
| --- | --- | --- | --- |
| **a** | **b** | **a + b + 0** | **output** |
| 1 | 1 | 2 | 1 |
| 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 |

* 1. Perceptron can be trained to compute specific function
     1. Start with perceptron with any values for weights (usually random)
     2. Choose an example, apply input, let perceptron compute the answer
     3. If answer right
        1. do nothing
     4. If answer wrong
        1. modify weights by adding or subtracting input vector (or some fraction of it)
        2. add to increase the output, subtract to decrease it
     5. Iterate over all input vectors, repeating as necessary, until perceptron learns what we want, within predefined error limit

* 1. Training the logical OR function:

|  |  |  |  |
| --- | --- | --- | --- |
| **a** | **b** | **bias** | **output** |
| 0 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 |

Initial weights all 0 ‣ weight vector, W = (0 0 0) • Cycle through inputs and change weights as necessary

* 1. Training the logical AND function:

|  |  |  |  |
| --- | --- | --- | --- |
| **a** | **b** | **bias** | **output** |
| 0 | 1 | 1 | 0 |
| 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 |

Initially weights all 0 ‣ weight vector, W = (0 0 0) • Cycle through inputs and change weights as necessary • Make your own table: ‣ InputWeights Result Action

1. Training perceptrons
   1. Intuition behind basic training algorithm:
   2. If unit should have gone on, but didn’t, increase influence of inputs that are on
      1. adding input (or fraction thereof) to weights will move in right direction
   3. If unit should have been off, but was on, decrease influence of units that were on
      1. subtracting input (or fraction thereof) from weights moves in right direction
   4. “fraction thereof” is the learning rate, 0 < a <= 1
      1. controls trade-off between speed and stability of convergence
2. Generalising the training procedure
   1. Consider continuous output values
   2. Start with perceptron with any weight values (0 or random)
      1. Select training example (X, y)
      2. Feed in input X, let perceptron compute the answer h = g(X⋅W)
      3. If answer right, do nothing
      4. If answer wrong:
         1. add this factor to each weight: Δwi = α ( y – h ) xi
         2. a is learning rate, 0 < a <= 1; it prevents values from jumping around wildly
         3. y is desired output
         4. h is actual output
      5. Iterate over all input vectors, repeat until stopping criterion reached (weight vector converges, or deltaW <= given constant)
         1. each cycle through input data is an epoch
3. Single-layer perceptron networks
   1. In single layer perceptron network there are
      1. input units
      2. output units
   2. As many perceptrons as output units
      1. each input unit connected to every output unit
      2. each connection has weight
      3. each output has soft thresholding function, e.g. sigmoid (as shown)
   3. Can learn multiple functions with one training sequence
      1. outputs are now vectors (as inputs always were)
4. Single-layer network training
   1. Can generalise weight updating algorithm for each perceptron in the net, given differentiable activation function, g:
      1. deltaW\_{i,j} = a ⋅ x\_i ⋅ ( y\_j – h\_j ) ⋅ g′( sum of (w\_{k, j} ⋅ x\_k))
         1. a : learning rate, which controls the amount of adjustment at each step
         2. y\_j, h\_j : desired and actual node outputs, respectively
         3. -( y\_j – h\_j ) : error in current node (j) output, compared with the example.
         4. g′ : derivative of activation function; for sigmoid, g′(x) = g(x) ( 1 – g(x) )
         5. sum of (w\_{k, j} ⋅ x\_k): value before activation at the current node
         6. (yj – hj) ⋅ g′( sum of (w\_{k, j} ⋅ x\_k))) called the delta value
      2. g′ adjusts error correction amount depending on slope of activation function
   2. Stopping criterion for training is based on overall error, E
      1. E = 0.5 \* sum of (y\_i – h\_i)^2
      2. This can be forced to 0 or some allowable level of approximation
5. The X-OR problem
   1. As seen from graph, classifier is computing straight line between two linearly separable sets of points, dividing space in two
      1. with more inputs, it would be plane or hyper-plane in higher-dimensional space
   2. If no straight-line partitions set, single perceptron cannot learn classification
      1. Exclusive OR is one such case
6. Solution is to build neural networks from layers of perceptrons
   1. So far, perceptron had two layers of values, one input and one output
      1. weighted connections between them
   2. For multiple layers, we use # layers of values to say how many there are
7. Proven that network of 3 layers can approximate any function
   1. So one hidden layer of perceptrons
   2. Cybenko’s theorem, 1989, for sigmoid activation;
8. Multi-layered NNs:
   1. How do we learn the weights?
   2. Assume input and target are known
   3. Assume weights have been initialised
   4. Backpropagation
      1. Compute error E between target output and actual output
      2. Find dE/dw\_i for all weights w\_i in output layer
      3. Repeat (2) for hidden layer(s)
      4. Update all weights by w\_{i, new}= w\_{i, old} \* -a \* dE/dw\_i , where a is the learning rate
      5. Compute new error E
      6. If E below tolerance, stop
      7. If E greater than tolerance, repeat process
9. Suppose we have :  
     
   Inputs: i1 = 1, i2 = 1  
   Initial weights: [ w1=-2, w2=1, w3=1, w4=-1, w5=1, w6=1 ]   
   Target: o = 0  
   Learning rate: a = 1  
     
   The inputs are two nodes that point to both h1 and h2 nodes each. So i1 points to h1 and h2 with weights w1 and w3, a2 points to h1 and h2 with weights w2 and w4.  
     
   h1 and h2 point to o with w5 and w6 respectively.  
     
   Rather than threshold, output of hidden and output layer neurons is logistic or sigmoid function  
     
   y=1/(1+exp^{-x})  
     
   Has nice property that dy/dx= y(1-y)  
     
   h\_1,in=w1 i1+w2 i2 =-2⋅1+1⋅1=-1   
   h\_2,in=w3 i1+w4 i2=1⋅1-1⋅1=0   
   h\_1,out = 1/(1+exp^(- h\_1,in))= 0.26894   
   h\_2,out = 1/(1+exp^(- h\_2,in))=0.5   
   o\_in= w5 h\_1,out+w6 h\_2,out=0.76894   
   o\_out = 1/(1+exp^(-o\_in))=0.68329   
   E=(ot – o\_out)^2 /2=0.23344  
     
   Partial derivatives:  
     
     
     
   del E / del w5 = (del o\_in/del w5) \* (del o\_out/del o\_in) \* (del E/del o\_out)  
     
   del E/del w6 = 0.073934  
     
   del E / del o\_out = o\_out – ot = 0.68329  
     
   del o\_out/del o\_in = o\_out(1 – o\_out) = 0.21640  
     
   del o\_in/del w5 = h\_1,out = 0.26894  
     
   del E/del w1 = (del h\_1,out/del w1) \* (del h\_1,out/ del h\_1,in) \* (del E/ del h\_1,out)  
     
   del E/del w1 = 0.02907  
   del E/del w2 = 0.02907  
   del E/del w3 = 0.03697  
   del E/del w4 = 0.03697  
     
   del E/del h\_1,out = (del E/ del h\_1,out) \* (del o\_out/ del o\_in) \* (del o\_in/ del h\_1,out) = 0.68329 ⋅ 0.21640 ⋅ 1   
     
   del h\_1,out/ del h\_1,in = h\_1,out (1 - del h\_1,out) = 0.26894(1- 0.26894)  
     
   del h\_1,in/del w1 = i1 = 1  
     
   Finally, for all weights: w\_new = w\_old – (del E/del w\_old)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | Initial | One iteration | 200 iterations |
| Weights | W1 | -2 | -2.03 | -2.371 |
| W2 | 1 | 0.971 | 0.630 |
| W3 | 1 | 0.963 | 1.939 |
| W4 | -1 | -1.04 | -0.061 |
| W5 | 1 | 0.96 | -0.295 |
| W6 | 1 | 0.926 | -2.916 |
| Error | | 0.233 | 0.222 | 0.00251 |

1. Network architectures and training:
   1. Can be as many input, hidden and output units as you like
   2. Consider simple feedforward architecture, where:
      1. each unit connected to every unit in next layer, & only next layer
      2. each connection has weight
      3. each unit in hidden layer has nonlinear activation function, e.g. sigmoid or ReLU (rectified linear unit, g(x) = max(0, x))
   3. No known procedure to compute required number of units
      1. Too many units leads to long training time & overfitting to data
      2. Too few units leads to inability to learn
      3. Skill, judgement and experience are required!
2. Multilayer feedforward networks:
   1. Now there are two sets of weights, one for each layer
3. Activations functions:  
     
   X = net = sum of (w\_i x\_i + w\_0), Y = o = sigma(net)  
   1. Step function: straight-line S shape with maximum value 1 and minimum 0.
   2. Sign function: straight-line S shape with maximum value 1 and minimum -1.
   3. Sigmoid function: curvy-live S shape with maximum value 1 and minimum 0.
   4. Linear function: linear
   5. Rectified Linear Unit: max of (0, +x). All -x values are equalled to 0.
4. The role of bias:
   1. Net = sum of (w\_i x\_i) + w0 x0
   2. O = sigma(net)
   3. W0 = -theta
   4. The threshold where the neuron fires should be adjustable
   5. Instead of adjusting the threshold we add the bias term
   6. Defines how strong the neuron input should be before the neuron fires
5. Deep neural networks
   1. Deep networks are those with many hidden layers between input & output layers
   2. Hidden nodes do not directly receive inputs nor send outputs to external environment
6. Multilayered NNs

|  |  |
| --- | --- |
| W\_ji | Weight associated with ith input to hidden unit j |
| W\_kj | Weight associated with jth input to output unit k |
| Y\_j | Output of jth hidden unit |
| O\_k | Output of kth output hidden unit |
| n | Number of inputs |
| nH | Number of hidden neurons |
| K | Number of output neurons |

y\_j = sigma(sum of(x\_i w\_ji))  
o\_k = sigma(sum of(y\_j w\_kj))  
o\_k = sigma(sigma(sum of(xi w\_ji))w\_kj)

1. Representational Power
   1. Boolean functions: Every boolean function can be represented exactly by some network with two layers
   2. Continuous functions: Every bounded continuous function can be approximated with arbitrarily small error by network with 2 layers
   3. Arbitrary functions: Any function can be approximated to arbitrary accuracy by network with 3 layers
   4. But we do not know
      1. appropriate number of hidden neurons
      2. proper weight values
2. NNs as function approximators
   1. Binary Classification: Target Values 0 or -1 (negative) and 1 (positive)
   2. Regression: Target values continuous values [-inf, +inf]
   3. Ideally output = target: o approximately equal to t
3. Multi-class classification:
   1. Classes are represented as 1-hot vectors
   2. 4 classes require 4 output neurons
4. Training – Gradient Descent
   1. efficient algorithm for finding minimum value of function
   2. start with random weights & improve them iteratively
   3. Update weights each iteration to reduce error function
   4. Step in direction that decreases error E
   5. Opposite to derivative of E: delta w\_i = - n(del E/del w\_i)
   6. derivative: direction of steepest increase
5. Learning Rate:
   1. Learning rate: determines step size in direction of steepest decrease.
   2. usually takes small values, e.g. 0.01, 0.1
   3. If it takes large values then weights change a lot -> network unstable
6. Testing the trained network
   1. Can train network to model a dataset in as much detail as we like
   2. However, we want network to generalise
      1. i.e., not just learn values we told it (we know them already) but estimate (assumed continuous) functions that gave rise to them
   3. If we train perfectly, we may over-fit, and miss that generality
   4. To overcome this problem, divide data into training sets and test sets, and perform cross-validation
      1. for example, reserve 10% of the data for testing, and use 90% for training
         1. compare the predicted values on test inputs with their (known) correct outputs
      2. better, use 10-fold cross-validation:
         1. partition the data into 10 subsets
         2. train on 9 subsets, test on the remaining subset
         3. repeat using each subset once as the testing set
7. Batch Gradient Descent
   1. All N examples fed to network.
   2. Weights updated only after all examples presented
      1. 1 epoch of training
   3. batch-size is B = N.
      1. N/B = 1 iterations per epoch
   4. For each example
      1. Each weight w\_i updated (by delta w­\_i) based on average gradient over N examples
      2. Weights updated only per epoch
   5. Stochastic Gradient Descent
      1. M randomly chosen examples fed to network.
         1. Batch-size = M (M usually 32, 64, …, 512)
         2. M/B iterations per epoch
      2. Weights updated M/B per epoch.
         1. Updates are noisy, good for training
8. Backprop stopping criteria
   1. When gradient magnitude (or delta w\_i ) is small, i.e.
   2. When maximum number of epochs has been reached
   3. When error on the validation set does not decrease
      1. Error might decrease in training set but increase in validation set (overfitting!)
9. Learning rate decay
   1. If loss increases, learning rate is too high
   2. If loss goes down slowly, learning rate is low
   3. Can have learning rate high at first, then decrease after some epochs
10. Regularization Standard technique to reduce overfitting in Machine Learning   
    Regularisation techniques for NNs include:
    1. Early Stopping
    2. Dropout
    3. L2 regularization (weight decay)
    4. Augmentation
    5. More exotic ones (e.g. mix-up)
11. Dropout
    1. During training neurons are randomly dropped out Srivastava et al. A simple way to prevent neural networks from overfitting, JMLR 2014
    2. don’t modify the error function but the network itself
    3. Probability neuron preserved is p
    4. prevents neurons from co-adapting too much
    5. Each neuron should create useful features on its own without relying on other hidden units to correct its mistakes
    6. Test time: outgoing weights of neuron multiplied by p
12. Data augmentation:
    1. One of the best ways to avoid overfitting is more data
    2. Can artificially generate more data by introducing artificially more variations
    3. For images: flip left-right, colour, rotate, random cropping, etc.
13. Convolutional Neural Networks
    1. CNNs have been very successful in computer vision
    2. Method of choice for extracting features from images, video and audio
    3. Object recognition has been one of the most difficult problems in AI
14. ImageNet Competition – Object Classification:
    1. Classify 1000+ objects
    2. State-of-the-art before 2012: ~26% error
    3. New state-of-the-art in 2012 with deep networks: ~15%
    4. Tricks: Large dataset, GPUs for training, Augmentation, Dropout, ReLU
15. AlexNet
    1. Krizhevsky, Sutskever & Hinton: “ImageNet Classification with Deep Convolutional Neural Networks”, NIPS 2012
    2. It’s a deep network = many layers
    3. Each layer is either convolutional or subsampling layer
    4. Final layers are fully connected layers
16. Convolution Motivation
    1. Standard Fully Connected (FC) layer is not suitable for images
       1. Assume grayscale image (1 channel) of resolution 256x256
       2. Single neuron connected to all pixels will require 2562 = 65K params
       3. If first layer: 128 of those neurons, in total 65k\*128= 9M params!
    2. Convolution Layer:
       1. Divide the image into 3x3 windows.
       2. Apply the same FC layer for all windows
       3. A single neuron will have 3x3 params, and the whole layer 1K params!
    3. Max-pooling
       1. Decrease spatial dimensions as we add more layers
17. CNNs for face recognition
    1. Major application of Computer Vision is face recognition
    2. With CNNs researchers reported super-human accuracy
    3. feature maps produced at different resolutions. At early layers these feature maps “look like faces”. they become more abstract at later layers
18. Fully Convolutional Architectures
    1. From image to label map
    2. No Fully Connected layers exist
    3. For example assume we want to label every pixel with an object class label (semantic segmentation)
    4. The loss is applied in the end at each spatial location
19. Transfer Learning
    1. Pre-train network on one (usually large) dataset
       1. ImageNet is the best candidate
    2. Use the weights as initialisation for a new dataset
       1. If the new dataset is small then this works much better than initialising from scratch
20. Multi-task learning
    1. Train a single network to perform several tasks
       1. A single network is shared for all tasks
       2. Specialised networks (called heads) for each task
       3. In some cases multi-task learning improves accuracy
21. Unsupervised Learning
    1. ImageNet pretraining works well but we want to do better
    2. Collecting and labelling a bigger dataset very expensive (does not scale up)
    3. Solution: pretrain a network on large amounts of data without labels!
    4. Need to define task that does not require human annotation. E.g.: Image jigsaw solving
22. Neurally inspired computing
    1. Much neural network research makes biologically implausible assumptions about how neurons work (e.g. backpropagation)
       1. NNs are neurally inspired rather than brain science
    2. No NN models distinguish humans from dogs or worms
       1. Whatever distinguishes higher cognitive capacities (language, reasoning) may not be apparent at this low level of analysis
    3. Relation between NN and “symbolic AI”?
       1. Some claim NN models don’t have symbols and representations
       2. Others think of NNs as simply being an “implementation-level” theory
       3. A major weakness of NNs is their inability to explain outputs (hidden nodes and weights have no individual, explicit meaning)
       4. NNs started out as a branch of statistical pattern classification; although practice outstripped theory for a while, underlying theory of NNs is well understood
23. Neural networks: then and now
    1. Neural networks have enjoyed/suffered a checkered history
       1. switch being in and out of fashion each decade or so
    2. After initial excitement about 'electronic brains', progress (and funding) slowed with inability of single-layer networks to learn functions like XOR
    3. Interest revived as it became clear that multi-layer networks could be trained with backpropagation (1986) to learn arbitrary functions
    4. Progress slowed again as backprop failed to train deeper networks
    5. Since ~2006, another resurgence of interest in neural networks has been observed, with new methods of training deep networks (deep learning)
    6. State of art speech recognition, language understanding, computer vision, music transcription and strategy game playing all use deep neural networks