**Machine Learning**

**Session 6**

1. Artificial Neural Networks:
   1. Nodes (cf neurons)
   2. Adaptive Weights (cf synaptic strength)
   3. Defined by:
      1. Interconnection pattern between layers of neurons
      2. The activation function that converts a neuron’s weighted input to its output activation.
      3. The learning process for updating the interconnection weights
   4. Can model arbitrarily complex non-linear functions of inputs.
2. Fixed Versus Adaptive Basis Functions
   1. Previously we used fixed linear combinations of features.
   2. Limited by the curse of dimensionality
   3. In contrast, Neural Nets adapt the parameters of the basis functions during training.
3. Perceptron: A Simple Model Neuron
   1. Uses the step function as activation
      1. Linear Threshold Unit (LTU)  
           
         Three independent nodes (theta1, theta2, …, thetaN) pointing to a node (theta0) that sums up the n nodes and uses the step function to an output node.
4. Neuron as a Logistic Regression Unit:
   1. h\_{theta}(x) = 1 / (1 + exp^{-theta \* x})
   2. Can think of a logistic regression unit as a neuron (function) that multiplies the input by the parameters (weights) and squashes the resulting sum through the sigmoid.
   3. Logistic Regression Unit => Perception if weight strength goes to infinity, so sigmoid => step.
5. Feed Forward Neural Net
   1. Connected set of logistic regression units
      1. Arranged in layers.
      2. Each unit’s output is a non-linear function (e.g., sigmoid, step function) of a linear combination of its inputs.
   2. Input layer size:
      1. Size of input to classify/regress
   3. Output layer size
      1. Size of target for classification/regression
   4. Any number of hidden neurons
   5. Any number of hidden layers
6. A sigmoid/Logistic Neuron:
   1. Three nodes:
      1. Bias = x0 = 1
      2. Input 1 = x1
      3. Input 2 = x2
   2. Pointing towards summation node that uses h\_{theta}(x), the sigmoid function.
   3. X = vector<xtheta, x1, x2>, theta = <theta0, theta1, theta2>
7. A Single Neuron:
   1. Three inputs to one summation node
8. A neural network:
   1. Layers of single neurons.
   2. Input layer:
      1. X = vector<xtheta, x1, x2>, each input individually pointing to the all the neurons in the next layer.
   3. Hidden layer 1, 2 & 3 (grid system of neurons, each pointing to the all the neurons in the next layers individually, with numbers in brackets indicating rowColumn):

|  |  |  |
| --- | --- | --- |
| H\_{theta}(x) 11 | H\_{theta}(x) 21 | H\_{theta}(x) 31 |
| H\_{theta}(x) 12 | H\_{theta}(x) 22 | H\_{theta}(x) 32 |
| H\_{theta}(x) 13 | H\_{theta}(x) 23 | H\_{theta}(x) 33 |

* 1. All pointing to the output layer.

1. Summary:
   1. Neural nets as interconnected layers of (e.g., sigmoid / logistic regression) neurons/units.
   2. Layer l+1’s inputs are layer l’s outputs.
   3. Can have any number of layers.
2. Linearly Non-Separable Problems E.g., the famous XOR problem, nonlinearly separable classification.
3. How can a NN help?
   1. It can combine decision boundaries at each level of the network.
      1. The decision of one layer can become the input of the next
   2. This can make more complex boundaries overall.
4. Logistic Unit AND:
   1. Three inputs = { xtheta = 1, x1 = 0/1, x2 = 0/1 } as nodes
   2. Three vertices = { wtheta = -1.5, w1 = 1, w2 = 1 } pointing to a node that uses h\_{theta}(x).

|  |  |  |
| --- | --- | --- |
| A | B | A AND B |
| 0 | 0 | <0 |
| 0 | 1 | <0 |
| 1 | 0 | <0 |
| 1 | 1 | >0 (class 1) |

* 1. What weights will give AND?
     1. Recall H\_w(x) = 1; if w1x1 + w2x2 + w0 >0, otherwise H\_w(x)=0

1. Logistic Unit OR:
   1. Three inputs = { xtheta = 1, x1 = 0/1, x2 = 0/1 } as nodes
   2. Three vertices = { wtheta = -0.5, w1 = 1, w2 = 1 } pointing to a node that uses h\_{theta}(x).

|  |  |  |
| --- | --- | --- |
| A | B | A AND B |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

* 1. What weights will give AND?
     1. Recall H\_w(x) = 1; if w1x1 + w2x2 + w0 >0, otherwise H\_w(x)=0

1. Logistic Unit XOR:
   1. Three inputs = { xtheta = 1, x1 = 0/1, x2 = 0/1 } as nodes
   2. Three vertices = { Impossible } pointing to a node that uses h\_{theta}(x).

|  |  |  |
| --- | --- | --- |
| A | B | A AND B |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

* 1. What weights will give AND?
     1. Recall H\_w(x) = 1; if w1x1 + w2x2 + w0 >0, otherwise H\_w(x)=0

1. Neural Net XOR:
   1. Three inputs = { xtheta = 1, x1 = 0/1, x2 = 0/1 } as nodes
   2. Weights are as vertices with values:
      1. xtheta = < -0.5 (first node), -1.5 (second node) >
      2. x1 = = < 1 (first node), 1 (second node) >
      3. x2 = = < 1 (first node), 1 (second node) >
   3. Pointing to two nodes that point to one node x\_{10} = 1
2. Neural Net XOR:
   1. Three inputs = { xtheta = 1, x1 = 0/1, x2 = 0/1 } as nodes
   2. Weights are as vertices with values:
      1. xtheta = < -0.5 (OR), -1.5 (AND) >
      2. x1 = = < 1 (OR), 1 (AND) >
      3. x2 = = < 1 (OR), 1 (AND) >
   3. Pointing to two nodes marked as OR and AND, that point to one node x\_{10} = 1 with weight -0.5 and called AND
3. Summary:
   1. Single neurons can only do linear decision boundaries.
   2. Networks of neurons can do (arbitrarily) complicated non-linear decisions.
4. Typical Activations
   1. Each neuron computes:  
        
      h(w^T\*x) = h(summation of w\_i\*x\_i + w\_0)
   2. Activation functions h(a)
      1. Perception (step-function activation)
         1. Issue: Not differentiable w.r.t weights.  
              
            H(a) = 1 if a > 0, -1 if a < 0.
      2. Sigmoid:  
           
         h(a) = 1 / (1 + exp^{-a})
      3. Tanh:  
           
         H(a) = (exp^{a} – exp^{-a}) / (exp^{a} + exp^{-a})
   3. Rectified Linear Unit (ReLU)
   4. max(0, x); converts all negative values to 0; linear graph
   5. Widely used in Deep Neural Nets
   6. Accelerates convergence during training
   7. Leaky ReLU
5. Unpacking the prediction mode:
   1. What’s the prediction made?  
        
      One input layer with nodes <x0, x1, x2>  
      One hidden layer with nodes <h10, h11(x), h12(x), h13(x)  
      One output layer with node <h21(x)>
   2. h11(x) = h(w1 10 + w1 11x1 +w1 21x2)   
      h12(x) = h(w1 20 + w1 21x1 +w1 22x2)   
      h13(x) = h(w 1 30 + w1 31x1 +w1 32x2)   
      h21(x) = h(w^2 10 + w^2 11h1 1 +w^2 12h1 2 +w^2 13h1 3)
   3. h2 k(x) = h\_k \* (summation of w^2\_{kj} dot h\_j \* (summation of w^1\_{ji} x\_i))  
        
      h2(x) = h(W^2 dot h(W^1 x))
   4. Suppose we used linear activation h.
      1. What happens to the prediction?  
           
         h2(x) = h(W^2 dot h(W^1 x))  
         h2(x) = W^2 W^1 x  
         h2(x) = W’ x  
           
         No matter how many layers…. – Still a simple linear model.
6. Neural Nets can do many tasks
   1. Single output regression
   2. Multiple output regression
   3. Binary classification.
   4. Multiclass classification.
   5. Binary multi-label classification.
   6. … by changing the activation function of output layer.
7. Same network for all upcoming examples:   
     
   One input layer with nodes <x0, x1, x2>  
   One hidden layer with nodes <h10, h11(x), h12(x), h13(x)  
   One output layer with node <h21(x)>
8. Neural Nets can do many tasks:
   1. Single output regression:
      1. Identity activation: h^2(a) = a
         1. Linear combination of previous layer is range +/- infinity
      2. h11(x) = h(w1 10 + w1 11 x1 +w1 21 x2)   
         h12(x) = h(w1 20 + w1 21x1 +w1 22 x2)   
         h13(x) = h(w 1 30 + w1 31x1 +w1 32 x2)
      3. h21(x) = h(w2 10 + w2 11h1 1 +w2 12h1 2 +w2 13h1 3)
   2. Multiple output regression:
      1. Identity activation: h^2(a) = a
         1. Linear combination of previous layer is range +/- infinity
      2. Use multiple output units
      3. h21(x) = h(w2 10 + w2 11h1 1 +w2 12h1 2 +w2 13h1 3)
      4. h22(x) = h(w2 20 + w2 21h1 1 +w2 22h1 2 +w2 23h1 3)
   3. Binary classification:
      1. Sigmoid activation: h^2(a) = 1 / (1 + exp^{-a})
         1. Linear combination of previous layer squashed into range [0,1].
      2. h21(x) = sigma(w2 10 + w2 11h1 1 +w2 12h1 2 +w2 13h1 3)
   4. Multiclass/Multi-label classification:
      1. Sigmoid activation: h2(a) = 1 / (1 + exp^{-a})
         1. Linear combination of previous layer squashed into range [0,1]
      2. Use multiple output units in 1-of-K encoding.
      3. h21(x) = sigma(w2 10 + w2 11h1 1 +w2 12h1 2 +w2 13h1 3)  
         h22(x) = sigma(w2 20 + w2 21h1 1 +w2 22h1 2 +w2 23h1 3)  
         h23(x) = sigma(w2 30 + w2 31h1 1 +w2 32h1 2 +w2 33h1 3)
9. Summary:
   1. Non-linear activations are essential
   2. Different activations and different structures for different tasks
10. Cost Functions:
    1. Regression:
       1. As with linear & logistic regression, to train a Neural Net we’ll need a cost function.
          1. And training data D={x,y} pairs….
       2. Regression prediction:
          1. h2(x) = h2(w2 10 + w2 11h1 1 +w2 12h1 2 +w2 13h1 3)  
             h2(x) = w2 10 + w2 11h1 1 +w2 12h1 2 +w2 13h1 3
       3. Regression cost: Mean Squared Error:
          1. E(w) = (1 / N) \* (summation of (h2(x\_n) – y\_n)^2)
    2. Classification:
       1. As with linear & logistic regression, to train a Neural Net we’ll need a cost function.
          1. And training data D={x,y} pairs….
       2. Classification prediction:  
            
          h2(x) = h(w2 10 + w2 11h1 1 +w2 12h1 2 +w2 13h1 3)  
          h2(x) = sigma(w2 10 + w2 11h1 1 +w2 12h1 2 +w2 13h1 3)
       3. Classification cost:  
            
          E(w) = (1 / N) \* (summation of log(h2(x\_n) + (1 – y\_n) log(1 – h2(x\_n)))
    3. Computing the Cost:
       1. To train NN, find weights to minimise the cost.
       2. We know how to write code to compute E(w)
          1. Do the “forward propagation” and make predictions.
          2. Use predictions to compute cost.
       3. …. But how to minimise it efficiently?
    4. Minimising the cost:
       1. Cost is non-Convex:
          1. No global minima.
          2. Require iterative gradient methods to train.
          3. Gradient descent will only converge to local minima.
       2. For Linear/Logistic regression, each weight had a straightforward derivative.
       3. For neural net, complicated by many layers of interdependent weights.
11. Backpropagation Algorithm:
    1. Derivative of cost w.r.t any weight w\_{ji}:
       1. Depends on activation: the linear combination of inputs before non-linearity.
       2. Activation:
          1. Every neuron does something like: a\_{j} = summation of w\_{ji} \* z\_{i}
             1. (z may inputs or previous layer’s outputs)
          2. Before doing non-linearity: h(a\_{j})
       3. Weight update via activations:  
            
          Partial derivatives:  
          (del E(w) / del w\_{ji}) = (del E(w) / del a\_{j}) \* ((del a\_{j} / del w\_{ji}))  
            
          (del a\_{j} / del w\_{ji}) = z\_{i}  
            
          (del E(w) / del a\_{j}) = delta\_{i}  
            
          (del E(w) / del w\_{ji}) = delta\_{i} \* z\_{i}
    2. Iterate:
       1. Propagate forward to find activations a and h(a) of all internal and output units. [Done]
       2. Evaluate error term delta for output units [TBD].
       3. Backpropagate output deltas to obtain internal deltas:  
            
          delta\_{j} = h’(a\_{j}) \* summation of w\_{kj} \* delta\_{k} (Activation function must be differentiable!)
       4. Use deltas to get a gradient update for each weight:  
            
          (del E(w) / del w\_{ji}) = delta\_{i} \* z\_{i}  
            
          (del E(w) / del w\_{kj}) = delta\_{k} \* z\_{j}
12. Output Error:
    1. Regression:
       1. Output error term: (del E(w) / del a\_{k}) = delta\_{k}
       2. For linear regression:
          1. Cost: Square Deviation
          2. Activation: Identity  
               
             E\_n(w) = 0.5 \* (h(a\_j) - y)^2  
               
             (del E(w) / del a\_{k}) = h’(a\_k) \* (h(a\_k) - y)  
               
             h(a\_k) = a\_k, h’(a\_k) = 1  
               
             (del E(w) / del a\_{k}) = delta\_{k} = (h(a\_k) - y)
    2. Binary Classifier:
       1. Output error term: (del E(w) / del a\_{k}) = delta\_{k}
       2. For binary classifier
          1. Cost: Square Deviation (or cross-entropy)
          2. Activation: Sigmoid.  
               
             E\_n(w) = 0.5 \* (h(a\_j) - y)^2  
               
             (del E(w) / del a\_{k}) = h’(a\_k) \* (h(a\_k) - y)  
               
             h(a) = sigma(a) = 1 / (1 + exp^{-a})  
             h’(a) = sigma’(a) = sigma(a) \* (1 - sigma(a))  
               
             (del E(w) / del a\_{k}) = sigma’(a\_{k}) \* (sigma(a\_{k}) - y)
13. Training Overview
    1. Implement forward propagation to get h(x) for any x.
    2. Implement cost function computation.
    3. Implement backpropagation to compute partial derivatives.
    4. Iterate forward & backward propagation.
14. Summary
    1. Training neural nets:
       1. Use backpropagation to minimise their cost function.
       2. Errors at later nodes are backpropagated to compute errors at earlier nodes.
       3. Errors at each node give gradient update for that node.
15. Batch vs Online
    1. Updates in past couple slides are gradients w.r.t one single input n.
       1. Online Gradient Descent:
          1. Iterate over data n, and weights w(i, j):   
               
             w\_{ji} = w\_{ji} – (alpha \* (del E\_n(w) / del w\_{ji}))
       2. Batch Gradient Descent:
          1. Iterate over weights w(i, j):  
               
             w\_{ji} = w\_{ji} – (alpha \* summation of (del E\_n(w) / del w\_{ji}))
16. Gradient checking:
    1. When implementing gradient-descent algorithms like backprop….
    2. May be useful to check correctness of your derivatives numerically.
    3. Can implement finite difference numerical differentiation to check them.
    4. I.e., perturb the current weight and re-compute the network’s error. Get gradient from this change in error.  
         
       (del E(w) / del w\_{ji}) = delta\_{j} \* h\_{i}  
       should be equal to  
       (del E(w) / del w\_{ji}) = (E\_n \* (w\_{ji} + epsilon)) – (E\_n \* (w\_{ji} - epsilon)) / 2\*epsilon
    5. Aside: You could also implement gradient descent by numerical differentiation, but slow!
       1. Each forward propagation costs O(W).
       2. Each weight must be perturbed individually at cost O(W).
    6. Overall cost O(W^2):  
         
       (del E(w) / del w\_{ji}) = (E\_n \* (w\_{ji} + epsilon)) – (E\_n \* (w\_{ji} - epsilon)) / 2\*epsilon
    7. – I.e., perturb the current weight and re-compute the network’s error. Get gradient from this change in error.
17. Initialization:
    1. For gradient methods need to pick initial weight vectors w.
       1. Zero initialization:
          1. Every hidden unit gets the same input.
          2. Nothing different to backprop.
          3. Network never learns anything
       2. Solution:
          1. Init randomly in [-epsilon, +epsilon]
18. Local minima:
    1. Neural Net costs are not Convex
    2. Solutions:
       1. Accept local minima.
       2. Online rather than batch gradient descent may help jitter out of minima.
       3. Repeatedly restart from different random initial conditions, take the best performing network. (Expensive)
       4. …momentum, etc.
19. Overfitting in Neural Nets
    1. Neural Nets can have lots of parameters (weights)
    2. 10s of millions in practice!
    3. Overfitting is a risk, especially if limited data.
    4. Solutions: Regularization:
       1. Use L2 regularizer/weight decay as we did with linear/logistic regression (needs update to gradients).
    5. Solutions: Early Stopping:
       1. At each iteration of gradient descent, check the network cost on validation set.
       2. Stop once validation error starts to increase (although train error will still be decreasing).
20. NN: Design Decisions
    1. Network Architecture
       1. # of Hidden Layers.
       2. # of Hidden Nodes.
    2. (# of input & output relatively easy)
    3. Cost function choice
    4. Activation function choice
    5. Learning rate
21. Training Overview (2)
    1. Choose architecture
    2. Initialize weights to small random numbers
    3. Implement forward propagation to get h(x) for any x.
    4. Implement cost function computation.
    5. Implement backpropagation to compute partial derivatives.
    6. Iterate forward & backward propagation.
    7. Use gradient checking to debug.
    8. Disable gradient checking once debugged.
    9. Early Stop training once validation error increasing.
    10. (Repeatedly retrain to try and find different local minima)
22. Why were NNs out of favor?
    1. Slow to train:
       1. Vanilla gradient descent inefficient compared to convex optimisers (e.g., Kernel-SVM).
    2. Local minima & no better than alternatives:
       1. Kernel-SVMs can do “equally good” non-linear classification and get global optimum quickly.
    3. Over-fitting:
       1. Neural nets have lots of parameters.
       2. They seemed easy to overfit, and tricky to regularize them
23. Why are NNs back?
    1. Slow to train: Solved? – In ~2012, GPU computation used to speed up operations required for backprop x10-20 times (per GPU!) over CPU.
    2. Over-fitting: Solved?
       1. “Big Data” in 2017 vast available databases can constrain vast number of parameters.
       2. New regularization techniques: Dropout.
    3. Local minima: Doesn’t matter?
       1. With other stuff taken care of, local minima are usually pretty good.
    4. Technical advances:
       1. RELU activation.
       2. Deep neural nets can be better than shallow. (Wide as well)
    5. Feature learning
       1. Can input “raw data” an learn the features.