Artificial Intelligence Foundation – JC3001

Lecture 43: Neural Networks -III

Prof. Aladdin Ayesh (aladdin.ayesh@abdn.ac.uk)

Dr. Binod Bhattarai (binod.bhattarai@abdn.ac.uk)

Dr. Gideon Ogunniye, (g.ogunniye@abdn.ac.uk)

November 2025

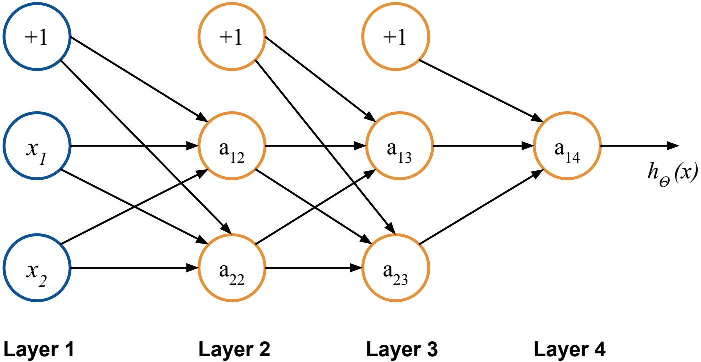
Material adapted from:
Russell and Norvig (AIMA Book): Chapter 21
Andrew Ng (Stanford University / Coursera)

Course Progression

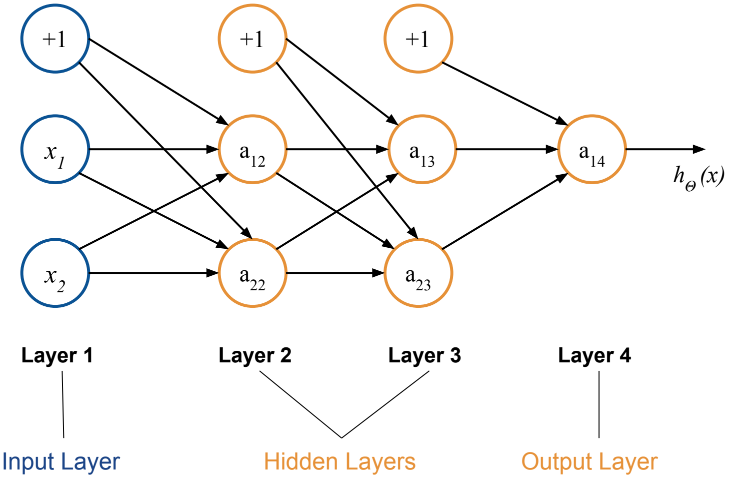
- Part 1: Introduction
  1. Introduction to AI ✓
  2. Agents ✓
- Part 2: Problem-solving
  1. Search 1: Uninformed Search ✓
  2. Search 2: Heuristic Search ✓
  3. Search 3: Local Search ✓
  4. Search 4: Adversarial Search ✓
- Part 3: Reasoning and Uncertainty
  1. Reasoning 1: Constraint Satisfaction ✓
  2. Reasoning 2: Logic and Inference ✓
  3. Probabilistic Reasoning 1: BNs ✓
  4. Probabilistic Reasoning 2: HMMs ✓
- Part 4: Planning
  1. Planning 1: Intro and Formalism ✓
  2. Planning 2: Algorithms & Heuristics ✓
  3. Planning 3: Hierarchical Planning ✓
  4. Planning 4: Stochastic Planning ✓
- Part 5: Learning
  1. Learning 1: Intro to ML ✓
  2. Learning 2: Regression ✓
  3. **Learning 3: Neural Networks**
  4. Learning 4: Reinforcement Learning
- Part 6: Conclusion
  1. Ethical Issues in AI
  2. Conclusions and Discussion

- Multilayer Perceptron

- Deep Learning Overview

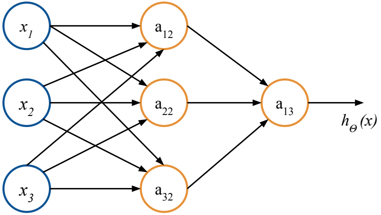

+1
+1
+1
x1
a12
a13
a14
hΘ(x)
x2
a22
a23
Layer 1
Layer 2
Layer 3
Layer 4

Multilayer Perceptron



Multilayer Perceptron

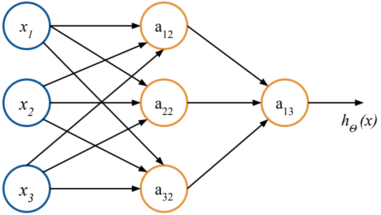
Forward Propagation



- a_{ik} = "activation" of unit i in for input k
- $\boldsymbol{W}^{(j)}$ = matrix of weights controlling function mapping from layer j to layer $j+1$

Multilayer Perceptron

Forward Propagation

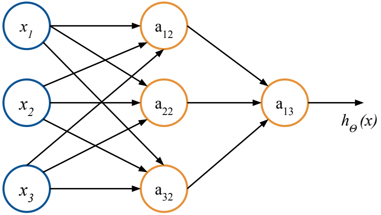


- a_{ik} = "activation" of unit i in for input k
- $\boldsymbol{W}^{(j)}$ = matrix of weights controlling function mapping from layer j to layer $j+1$

- $a_{12} = g(\boldsymbol{W}_{10}^{(1)} * x_0 + \boldsymbol{W}_{11}^{(1)} * x_1 + \boldsymbol{W}_{12}^{(1)} * x_2 + \boldsymbol{W}_{13}^{(1)} * x_3)$ or $a_{12} = g(z_{12})$

Multilayer Perceptron

Forward Propagation

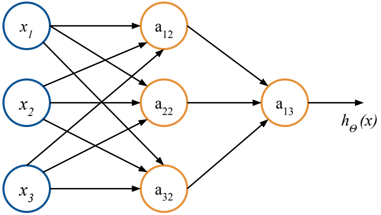


- a_{ik} = "activation" of unit i in for input k
- $\boldsymbol{W}^{(j)}$ = matrix of weights controlling function mapping from layer j to layer $j+1$

- $a_{12} = g(\boldsymbol{W}_{10}^{(1)} * x_0 + \boldsymbol{W}_{11}^{(1)} * x_1 + \boldsymbol{W}_{12}^{(1)} * x_2 + \boldsymbol{W}_{13}^{(1)} * x_3)$ or $a_{12} = g(z_{12})$
- $a_{22} = g(\boldsymbol{W}_{20}^{(1)} * x_0 + \boldsymbol{W}_{21}^{(1)} * x_1 + \boldsymbol{W}_{22}^{(1)} * x_2 + \boldsymbol{W}_{23}^{(1)} * x_3)$ or $a_{22} = g(z_{22})$
- $a_{32} = g(\boldsymbol{W}_{30}^{(1)} * x_0 + \boldsymbol{W}_{31}^{(1)} * x_1 + \boldsymbol{W}_{32}^{(1)} * x_2 + \boldsymbol{W}_{33}^{(1)} * x_3)$ or $a_{32} = g(z_{32})$

Multilayer Perceptron

Forward Propagation



- a_{ik} = "activation" of unit i in for input k
- $\boldsymbol{W}^{(j)}$ = matrix of weights controlling function mapping from layer j to layer $j+1$

- $a_{12} = g(\boldsymbol{W}_{10}^{(1)} * x_0 + \boldsymbol{W}_{11}^{(1)} * x_1 + \boldsymbol{W}_{12}^{(1)} * x_2 + \boldsymbol{W}_{13}^{(1)} * x_3)$ or $a_{12} = g(z_{12})$
- $a_{22} = g(\boldsymbol{W}_{20}^{(1)} * x_0 + \boldsymbol{W}_{21}^{(1)} * x_1 + \boldsymbol{W}_{22}^{(1)} * x_2 + \boldsymbol{W}_{23}^{(1)} * x_3)$ or $a_{22} = g(z_{22})$
- $a_{32} = g(\boldsymbol{W}_{30}^{(1)} * x_0 + \boldsymbol{W}_{31}^{(1)} * x_1 + \boldsymbol{W}_{32}^{(1)} * x_2 + \boldsymbol{W}_{33}^{(1)} * x_3)$ or $a_{32} = g(z_{32})$
- $h_{\boldsymbol{W}}(x) = a_{13} = g(\boldsymbol{W}_{10}^{(2)} * a_{02} + \boldsymbol{W}_{11}^{(2)} * a_{12} + \boldsymbol{W}_{12}^{(2)} * a_{22} + \boldsymbol{W}_{13}^{(2)} * a_{32})$ or $a_{13} = g(z_{13})$

Multilayer Perceptron

Cost Function

Based on logistic regression:

$$Loss(\boldsymbol{w}) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log h_{\boldsymbol{w}}(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_{\boldsymbol{w}}(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n} \boldsymbol{w}_j^2$$

$$Loss(\boldsymbol{W}) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} log(h_{\boldsymbol{W}}(x^{(i)}))_k + (1 - y_k^{(i)}) log(1 - (h_{\boldsymbol{W}}(x^{(i)}))_k)\right]$$
$$+ \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}} (\boldsymbol{W}_{ji}^{(l)})^2$$

- $h_{\boldsymbol{W}}(x) \in \mathbb{R}^K; (h_{\boldsymbol{W}}(x))_i = i^{th} output$

Multilayer Perceptron

Cost Function

Based on logistic regression:

$$Loss(\boldsymbol{w}) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log h_{\boldsymbol{w}}(x^{(i)}) + (1-y^{(i)}) \log(1-h_{\boldsymbol{w}}(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n} \boldsymbol{w}_j^2$$

$$Loss(\boldsymbol{W}) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} log(h_{\boldsymbol{W}}(x^{(i)}))_k + (1-y_k^{(i)}) log(1-(h_{\boldsymbol{W}}(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}} (\boldsymbol{W}_{ji}^{(l)})^2$$

- $h_{\boldsymbol{W}}(x) \in \mathbb{R}^K; (h_{\boldsymbol{W}}(x))_i = i^{th} output$
- $\delta_k = y_k^{(i)} log(h_{\boldsymbol{w}}(x^{(i)}))_k + (1-y_k^{(i)}) * log(1-(h_{\boldsymbol{W}}(x^{(i)}))_k)$

Based on logistic regression:

$$Loss(\boldsymbol{w}) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log h_{\boldsymbol{w}}(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_{\boldsymbol{w}}(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n} \boldsymbol{w}_j^2$$

$$Loss(\boldsymbol{W}) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} log(h_{\boldsymbol{W}}(x^{(i)}))_k + (1 - y_k^{(i)}) log(1 - (h_{\boldsymbol{W}}(x^{(i)}))_k)\right]$$
$$+ \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}} (\boldsymbol{W}_{ji}^{(l)})^2$$

- $h_{\boldsymbol{W}}(x) \in \mathbb{R}^K; (h_{\boldsymbol{W}}(x))_i = i^{th} output$
- $\delta_k = y_k^{(i)} log(h_{\boldsymbol{w}}(x^{(i)}))_k + (1 - y_k^{(i)}) * log(1 - (h_{\boldsymbol{W}}(x^{(i)}))_k)$
- $\sum_{i=1}^{m}\sum_{k=1}^{k} \delta_k$

Multilayer Perceptron

Cost Function

Based on logistic regression:

$$Loss(\boldsymbol{w}) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log h_{\boldsymbol{w}}(x^{(i)}) + (1-y^{(i)}) \log(1-h_{\boldsymbol{w}}(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n} \boldsymbol{w}_j^2$$

$$Loss(\boldsymbol{W}) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} log(h_{\boldsymbol{W}}(x^{(i)}))_k + (1-y_k^{(i)}) log(1-(h_{\boldsymbol{W}}(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}} (\boldsymbol{W}_{ji}^{(l)})^2$$

- $h_{\boldsymbol{W}}(x) \in \mathbb{R}^K; (h_{\boldsymbol{W}}(x))_i = i^{th} output$
- $\delta_k = y_k^{(i)} log(h_{\boldsymbol{w}}(x^{(i)}))_k + (1-y_k^{(i)}) * log(1-(h_{\boldsymbol{W}}(x^{(i)}))_k)$
- $\sum_{i=1}^{m}\sum_{k=1}^{k} \delta_k$
- $\frac{\lambda}{2m}\sum_{L=1}^{L-1}\sum_{i=1}^{S_l}\sum_{j=1}^{S_{l+1}} (\boldsymbol{W}_{ji}^{(l)})^2$

$$Loss(\boldsymbol{W}) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} log(h_{\boldsymbol{W}}(x^{(i)}))_k + (1 - y_k^{(i)}) log(1 - (h_{\boldsymbol{W}}(x^{(i)}))_k)\right]$$
$$+ \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}} (\boldsymbol{W}_{ji}^{(l)})^2$$

$min_{\boldsymbol{W}} Loss(\boldsymbol{W})$

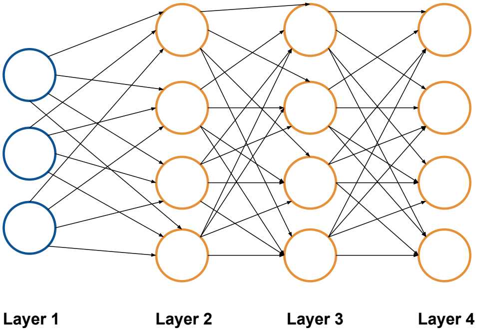
Need to compute:

- $Loss(\boldsymbol{W})$
- $\frac{\partial}{\partial \boldsymbol{W}_{ij}^{(l)}} Loss(\boldsymbol{W})$

- First Forward Propagation
- Intuition $\delta_j^{(l)}$ = "error" of node j in layer l

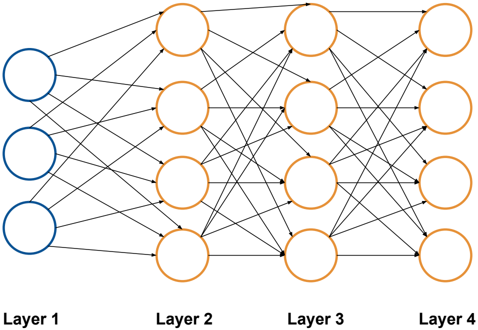
For each output unit (layer L = 4)

- $\delta_j^{(4)} = a_j^{(4)} - y_i$; vectorisation: $\delta^{(4)} = a^{(4)} - y$



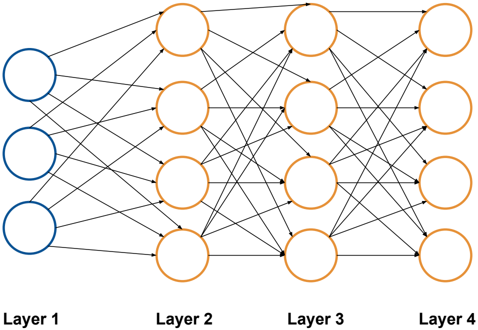
For each output unit (layer L = 4)

- $\delta_j^{(4)} = a_j^{(4)} - y_i$; vectorisation: $\delta^{(4)} = a^{(4)} - y$
- $\delta^{(3)} = \boldsymbol{W}^{(3)} * \delta^{(4)} * g^{'}(z(3))$ where $g^{'}(z(3)) = a^{(3)} * (1 - a^{(3)})$
- $\delta^{(2)} = \boldsymbol{W}^{(2)} * \delta^{(3)} * g^{'}(z(2))$ where $g^{'}(z(2)) = a^{(2)} * (1 - a^{(2)})$



For each output unit (layer L = 4)

- $\delta_j^{(4)} = a_j^{(4)} - y_i$; vectorisation:
  $\delta^{(4)} = a^{(4)} - y$
- $\delta^{(3)} = \boldsymbol{W}^{(3)} * \delta^{(4)} * g^{'}(z(3))$
  where $g^{'}(z(3)) = a^{(3)} * (1 - a^{(3)})$
- $\delta^{(2)} = \boldsymbol{W}^{(2)} * \delta^{(3)} * g^{'}(z(2))$
  where $g^{'}(z(2)) = a^{(2)} * (1 - a^{(2)})$
- No $\delta^{(1)}$ - this is the input vector



Algorithm 1 Backpropagation algorithm.

Training Set $\{(x^{(1)}, y^{(1)}), ..., (x^{(m)}, y^{(m)})\}$
Set $\Delta_{ij}^{(l)} = 0$ (for all l, i, j).
for $i = 1$ to m **do**
    Set $a^{(1)} = x^{(i)}$
    Perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, ..., L$
    Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$
    Compute $\delta^{(L-1)}, \delta^{(L-2)}, ..., \delta^{(2)}$
    $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} * \delta_i^{(l+1)}$ ▷ *Accumulate partial derivatives over* m
$D_{ij}^{(l)} = \frac{1}{m}\Delta_{ij}^{(l)} + \lambda \boldsymbol{W}_{ij}^{(l)}$ if $j \neq 0$
$D_{ij}^{(l)} = \frac{1}{m}\Delta_{ij}^{(l)}$ if $j = 0$

$$\frac{\partial}{\partial \boldsymbol{W}_{ij}^{(l)}} Loss(\boldsymbol{W}) = D_{ij}^{(l)}$$

- Number of neurons per layer
  - Function of the number of inputs/outputs?
  - Depends primarily on:
    - Number of instances/attributes
    - Noise in data
    - Complexity of the target function
    - Data distribution
  - Rule of thumb:
    - Number proportional to the number of inputs
    - The more, usually the better, however
      Much slower
      Possibility of overfitting

- Local minima
  - Cost function is non-convex, this susceptible to local minima
  - Use stochastic gradient descent
- Numeric problems
  - Normalize data
  - Use other more robust optimization algorithms

- At some point in training, the network suffers overfitting
- Alternatives:
  - Stop training before that happens (**early stop**)
  - Prune irrelevant connections and neurons (**prunning**)
  - **Regularization**

- We saw online updates (or sequential update)
- Possible to perform batch updates (after seeing all training instances)
- Which is best?
  - Depends on the application (no free lunch)

Online/Sequential Stochastic Gradient Descent

- Weights updated after **each instance** in random order
- Requires **less memory**
- **Faster!**
- Less susceptible to local minima
- Disadvantage: could be **unstable**
  - Requires a schedule for controlling the learning rate

Batch Updates

- Weights updated after seeing **all instances**
- More accurate estimate of the gradient vector
- **Stabler!**
- Much slower!
- Local Minima!

- Multilayer Perceptron
- Deep Learning Overview

- You already know at least one Deep Learning architecture:
  Multilayer Perceptron (MLP)
- Strictly speaking a "deep" architecture just needs to have more than two layers
  This is not very interesting
- There are many architectures for deep learning, each of which arranges such deep layers in specific ways:
  - Siamese Networks
  - Convolutional Neural Networks (CNNs)
  - Recurrent Neural Networks (RNNs) - e.g. LSTMs, GRU
  - Transformers
  - ...

Neural Networks Summary

- Perceptron
  - One layer.
  - Converges in any linearly separable problem.
- Multilayer Perceptron
  - Hidden layers
  - Works for non-linear problems
- Deep Learning overview

Any Questions.