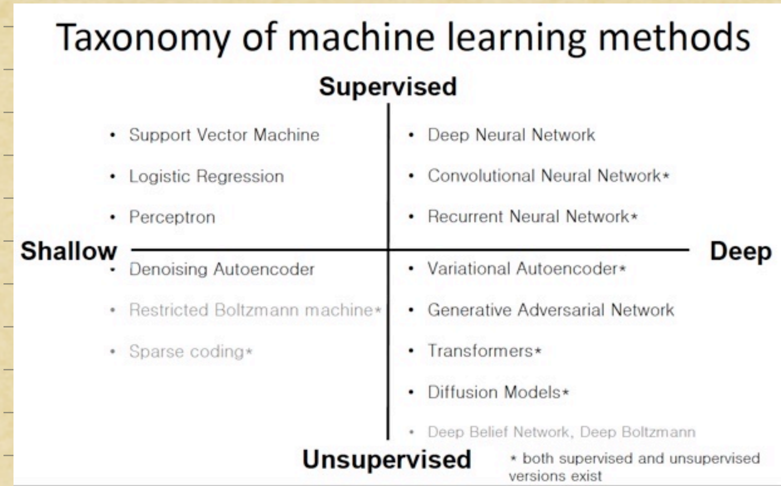
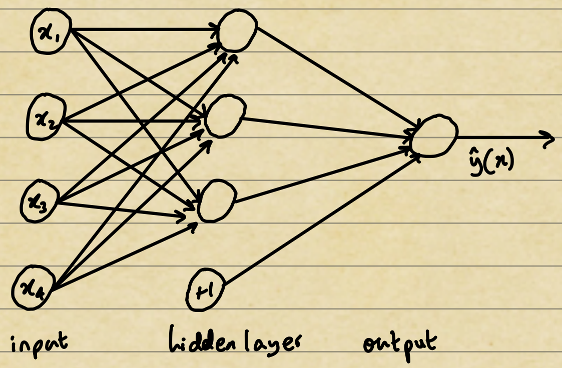
EECS 545 → Machine Learning

Lecture 11 → Neural Networks and Deep Learning

- The success of ML applications are dependent on having a good representation of data
- ML Engineers make sure to put emphasis on feature engineering
- Issues with hand crafted features,
  - (i) Need expert knowledge
  - (ii) Requires time consuming hand-tuning
- Key concept of deep learning is to learn multiple levels of representation of increasing complexity/abstraction



- Neural Network → equivalent to running multiple logistic regressions at the same time



- Feeding a vector of inputs through a series of logistic regressions will give us a vector of outputs that we can feed into another logistic regression function

- Linear neurons → simple, but limited in terms of representation power

$$h = b + \sum_i x_i w_i$$

- Rectified Linear Neurons → computes a linear weightage of the inputs and weights outputs a non-linear function of the input

$$z = b + \sum_i x_i w_i$$

$$h = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$$

- Sigmoid neurons → real valued output that is smooth and bounded function of their total output

$$z = b + \sum_i x_i w_i$$

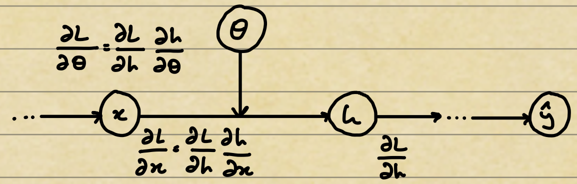
$$h = \frac{e^z}{1+e^z} = \frac{1}{1+e^{-z}}$$

- Softmax neurons → outputs sum to 1 with convenient derivatives

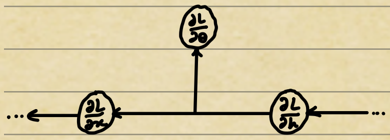
$$z_k = b^k + \sum_i x_i w_i^k$$

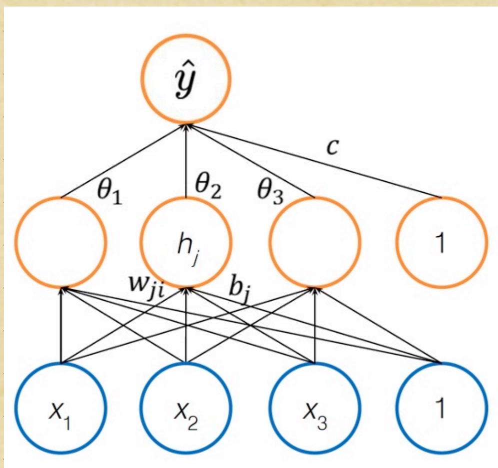
$$h_j = \frac{e^{z_j}}{\sum_k e^{z_k}} \qquad w_i^k \text{ is the weight vector for the k-th output}$$

- We update the weights of our NN using stochastic gradient descent
- To calculate the gradient of a complex function with multiple nested composite functions, we use backpropagation
  - Computing the gradient via the chain rule



- Assuming that $\frac{\partial L}{\partial h}$ is given, use the chain rule to compute the gradients
  - $\frac{\partial L}{\partial \theta} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial \theta}$, $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial x}$





- $h_j = f(\sum_i w_{ji} x_i + b_j)$
- $\hat{y} = \sum_j \theta_j h_j + c$
- Loss function → squared error, $(\hat{y} - y)^2$
- $\frac{\partial L}{\partial h} = 2(\hat{y} - y)$
- $\frac{\partial L}{\partial \theta_j} = \frac{\partial \hat{y}}{\partial \theta_j}\frac{\partial L}{\partial \hat{y}} = h_j \frac{\partial L}{\partial \hat{y}}$
- $\frac{\partial L}{\partial h_j} = \frac{\partial \hat{y}}{\partial h_j}\frac{\partial L}{\partial \hat{y}} = \theta_j \frac{\partial L}{\partial \hat{y}}$
- $\frac{\partial L}{\partial w_{ji}} = \frac{\partial h_j}{\partial w_{ji}}\frac{\partial L}{\partial h_j} = f' x_i \frac{\partial L}{\partial \hat{y}}$, where $f' = f'(\sum_i w_{ji} x_i + b_j)$

- For a NN with two hidden layers,
  - First hidden layer → $h_j^{(1)} = f(\sum_i w_{ji}^{(1)} x_i + b_j^{(1)})$
  - Second hidden layer → $h_j^{(2)} = f(\sum_i w_{ji}^{(2)} h_i^{(1)} + b_j^{(2)})$
  - Output → $\hat{y} = \sum_j \theta_j h_j^{(2)} + c$
- $\frac{\partial L}{\partial h_j^{(2)}} = \frac{\partial \hat{y}}{\partial h_j^{(2)}}\frac{\partial L}{\partial \hat{y}} = \theta_j \frac{\partial L}{\partial \hat{y}}$
- $\frac{\partial L}{\partial w_{ji}^{(2)}} = \frac{\partial h_j^{(2)}}{\partial w_{ji}^{(2)}}\frac{\partial L}{\partial h_j^{(2)}} = f'(z_j^{(2)}) h_i^{(1)} \frac{\partial L}{\partial h_j^{(2)}}$ where $z_j^{(2)} = \sum_i h_i^{(1)} w_{ji}^{(2)} + b_j^{(2)}$
- $\frac{\partial L}{\partial h_j^{(2)}} = \sum_j \frac{\partial h_j^{(2)}}{\partial h_i^{(1)}}\frac{\partial L}{\partial h_j^{(2)}} = \sum_j f'(z_i^{(2)}) w_{ji}^{(2)} \frac{\partial L}{\partial h_j^{(2)}}$ $\quad z_i^{(1)} = \sum_k x_k w_{ik}^{(1)} + b_i^{(1)}$
- $\frac{\partial L}{\partial w_{ik}^{(1)}} = \frac{\partial h_i^{(1)}}{\partial w_{ik}^{(1)}}\frac{\partial L}{\partial h_i^{(1)}} = f'(z_i^{(1)}) x_k \frac{\partial L}{\partial h_i^{(1)}}$

- E.g. for NN with loss function → cross entropy $-\sum_i y_i \log p_i$
  - Hidden layer → $h_j = f(\sum_i w_{ji} x_i + b_j)$
  - Output → $p_i = \frac{e^{h_i}}{\sum_j e^{h_j}}$

$$L = -\sum_i y_i \log \frac{e^{h_i}}{\sum_k e^{h_k}}$$
$$= -\sum_i y_i h_i + (\sum_i y_i)\log(\sum_k e^{h_k})$$
$$= \log(\sum_k e^{h_k}) - \sum_k y_k h_k$$

- $\frac{\partial L}{\partial h_j} = \frac{e^{h_j}}{\sum_k e^{h_k}} - y_j = p_j - y_j$

- $\frac{\partial L}{\partial w_{ji}} = \frac{\partial h_j}{\partial w_{ji}} \frac{\partial L}{\partial h_j} = f' x_i \frac{\partial L}{\partial h_j}$ where $f' = f'(\sum_i w_{ji} x_i + b_j)$

- Deep NNs → recursively stacking blocks of layers
- Computing the gradient is still via backpropagation

Backpropagation Algorithm

- Compute $\nabla_y L = \left[\frac{\partial L}{\partial y_1}, \dots, \frac{\partial L}{\partial y_n} \right]$ directly from loss function
- For each layer, compute gradients using the chain rule

$$\nabla_w L = \nabla_h L \, \nabla_w h$$
$$\nabla_x L = \nabla_h L \, \nabla_x h$$

- Issues with backpropagation includes gradient getting progressively more diluted
  - Changes may be minimal per iteration
- Easy to get stuck in local minima
- Typically requires a lot of labeled data