Optimization in ML

Perceptrons Review

- Inputs are feature values
- Each feature has a weight
- Sum is the activation: $z = \text{activation}_w(x) = \sum_i w_i \cdot f_i(x) = w \cdot f(x)$
- Sigmoid function:

$$\phi(z) = \frac{1}{1+e^{-z}} = \begin{cases} 1, \lim_{z \to \infty} \\ 0, \lim_{z \to -\infty} \end{cases}$$

- Chose w by doing maximum likelihood estimation:

$$\max_w ll(w) = \max_w \sum_i \log \mathbb{P}(y^{(i)} | x^{(i)}; w)$$

  with:

$$\mathbb{P}(y^{(i)} = +1 | x^{(i)}; w) = \frac{1}{1+e^{-w \cdot f(x^{(i)})}}$$
$$\mathbb{P}(y^{(i)} = -1 | x^{(i)}; w) = 1 - \frac{1}{1+e^{-w \cdot f(x^{(i)})}}$$

- Multiclass linear classification
  - Weight vector for each class w_y
  - Score (activation) of a class $w_y \cdot f(x)$
  - Prediction with the highest score wins $\triangleq y = \arg\max_y w_y \cdot f(x)$
  - Use the same maximum likelihood estimation function but with:

$$\mathbb{P}(y^{(i)} | x^{(i)}; w) = \frac{e^{w_{y^{(i)}} \cdot f(x^{(i)})}}{\sum_y e^{w_y \cdot f(x^{(i)})}}$$

Hill Climbing

- Start wherever
- Move to the best neighboring state
- If no neighbors are better than the current, quit
- Challenging with multiclass logistic regression b/c optimization is done over a continuous space $\Longrightarrow$ infinite neighbors

Gradients

- A gradient is a vector of partial derivatives with respect to each variable
- eg:

$$g(x, y) = x^2 y$$
$$\nabla g = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} = \begin{bmatrix} 2xy \\ x^2 \end{bmatrix}$$

  ## 1-D Optimization (Hill Climbing)

- Can sample random points w and take the w that maximizes the estimation
- Local hill climbing

  1. Evaluate $g(w_0 + h)$ and $g(w_0 - w)$
  2. Update w to the maximum of the two values
  3. Repeat until local max found or some time limit is reached

- Evaluate derivative to find the direction to step in:

$$\frac{\partial g(w_0)}{\partial w} = \lim_{h \to 0} \frac{g(w_0 + h) - g(w_0 - h)}{2h}$$

n Dimensional Optimization (Gradient Ascent)

- Perform update in uphill direction for each coordinate
- The steeper the slope (the higher the derivative) the bigger the step for that coordinate
- eg given $g(w_1, w_2)$ and hyperparameter for learning rate α:

$$w_1 \leftarrow w_1 + \alpha * \tfrac{\partial g}{\partial w_1}(w_1, w_2)$$
$$w_2 \leftarrow w_2 + \alpha * \tfrac{\partial g}{\partial w_2}(w_1, w_2)$$

- Same as

$$\nabla_w g(w) = \begin{bmatrix} \tfrac{\partial g}{\partial w_1}(w) \\ \tfrac{\partial g}{\partial w_2}(w) \end{bmatrix}$$

$$w \leftarrow w + \alpha * \nabla_w g(w)$$

- Alg $\in O(n)|_{n := |\text{dimensions}|}$
- α needs to be chosen carefully (generally s.t. update changes w by 0.1-1%)

Batch Gradient Ascent on the Log Likelihood Objective

$$w \leftarrow w + \alpha * \sum_i \nabla \text{log}(\mathbb{P}(y^{(i)}|x^{(i)}; w))$$

$$\max_w ll(w) = \max_w g(w)$$

Stochastic Gradient Ascent on the Log Likelihood Objective

- Once gradient on one training example has been computed, might as well incorporate before computing next one

$$w \leftarrow w + \alpha * \nabla \text{log}(\mathbb{P}(y^{(j)}|x^{(j)}; w))$$

$$\max_w ll(w) = \max_w g(w)$$

Mini-Batch Gradient Ascent on the Log Likelihood Objective

- Gradient over small set of training examples (mini-batch) can be computed in parallel

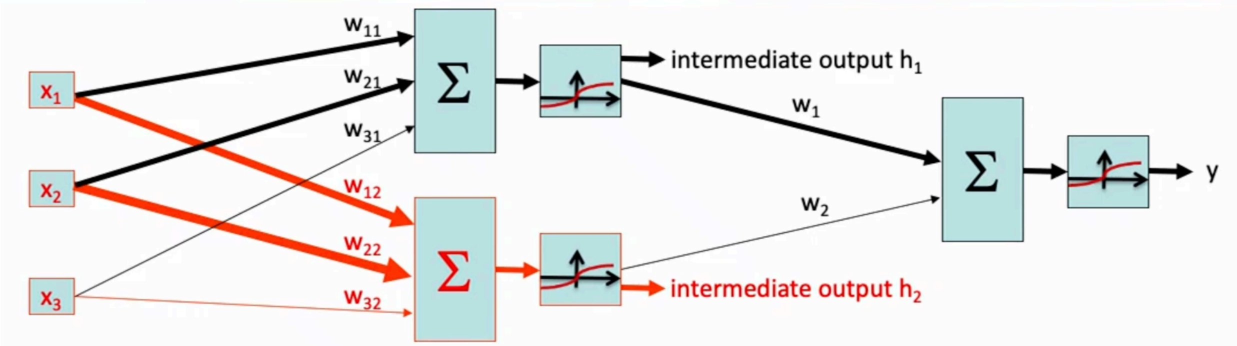
$$w \leftarrow w + \alpha * \sum_{j \in J} \nabla \log \mathbb{P}(y^{(j)}|x^{(j)}; w)\Bigg|_{J := \text{random subset of training examples}}$$

$$\max_w ll(w) = \max_w \sum_i \log \mathbb{P}(y^{(i)}|x^{(i)}; w)$$

Deep Learning and Neural Networks

- Goal $\triangleq$ avoid manual feature design; allow computer to learn features too
-
  - Different sets of weights for each individual summation
  - $h \triangleq$ hidden layers
- Finally:

$$y = \phi(w_1 h_1 + w_2 h_2) = \frac{1}{1 + e^{-(w_1 h_1 + w_2 h_2)}}$$



$$\text{intermediate output } h_2 = \phi(w_{12}x_1 + w_{22}x_2 + w_{32}x_3)$$
$$= \frac{1}{1 + e^{-(w_{12}x_1 + w_{22}x_2 + w_{32}x_3)}}$$

Figure 1: 525

$$y = \phi(w_1 h_1 + w_2 h_2)$$
$$= \phi(w_1\phi(w_{11}x_1 + w_{21}x_2 + w_{31}x_3) + w_2\phi(w_{12}x_1 + w_{22}x_2 + w_{32}x_3))$$

The same equation, formatted with matrices:

$$\phi\left(\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}\right)$$
$$= \phi\left(\begin{bmatrix} w_{11}x_1 + w_{21}x_2 + w_{31}x_3 & w_{12}x_1 + w_{22}x_2 + w_{32}x_3 \end{bmatrix}\right)$$
$$= \begin{bmatrix} h_1 & h_2 \end{bmatrix}$$

$$\phi\left(\begin{bmatrix} h_1 & h_2 \end{bmatrix}\begin{bmatrix} w_1 \\ w_2 \end{bmatrix}\right) = \phi(w_1 h_1 + w_2 h_2) = y$$

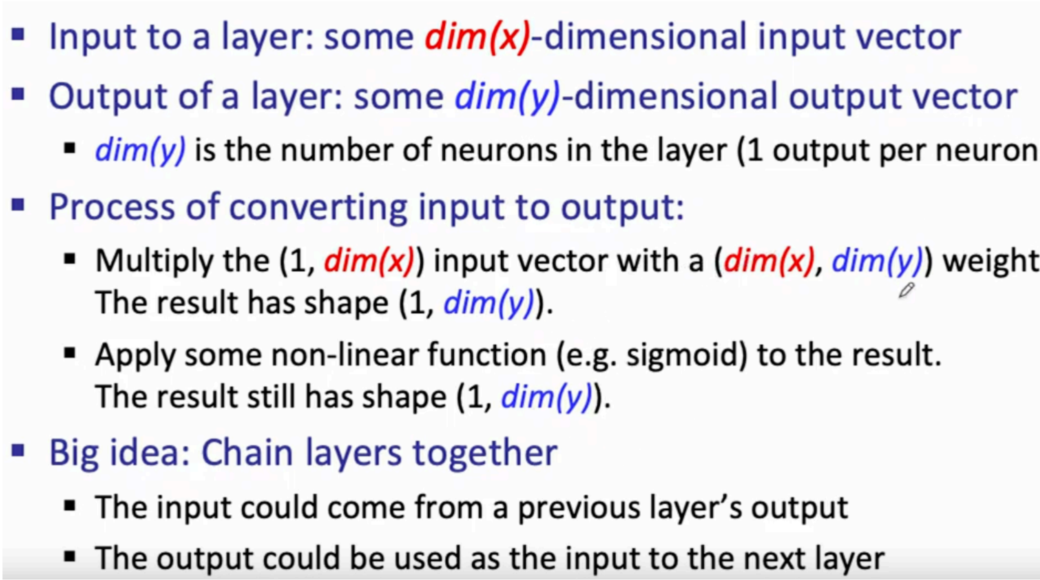
The same equation, formatted more compactly by introducing variables representing each matrix:

$$\phi(x \times W_{\text{layer 1}}) = h \qquad \phi(h \times W_{\text{layer 2}}) = y$$

- Same as:
  - Helpful to keep track of the shapes of different vectors and arrays
    - x has shape $(1, \dim(x))$
    - W_{layer1} has shape $(\dim(x), n)$
    - h has shape $(1, n)$
    - W_{layer2} has shape $(n, \dim(y))$
    - y has shape $(1, \dim(y))$
  - Can output an arbitrary $\dim(y)$ length vector
  - Layer 1 has weight matrix with shape (dim(x), n). These are the weights for n neurons, each taking dim(x) features as input. This transforms a dim(x)-dimensional input vector into an n-dimensional output vector.
  - Layer 2 has weight matrix with shape (n, dim(y)). These are the weights for dim(y) neurons,

each taking n features as input. This transforms an n-dimensional input vector into a dim(y)-dimensional output vector.

- Multi-layer Neural Network:



$$z_i^{(k)} = g\left(\sum_j W_{i,j}^{(k-1,k)} z_j^{(k-1)}\right)\Bigg|_{g := \text{nonlinear activation function}}$$

Batch Sizes

$$
\begin{aligned}
y_1 &= \phi(w_1 h_{11} + w_2 h_{12}) \\
&= \phi(w_1 \phi(w_{11}x_{11} + w_{21}x_{12} + w_{31}x_{13}) + w_2 \phi(w_{12}x_{11} + w_{22}x_{12} + w_{32}x_{13})) \\
y_2 &= \phi(w_1 h_{21} + w_2 h_{22}) \\
&= \phi(w_1 \phi(w_{11}x_{21} + w_{21}x_{22} + w_{31}x_{23}) + w_2 \phi(w_{12}x_{21} + w_{22}x_{22} + w_{32}x_{23}))
\end{aligned}
$$

Rewriting in matrix form:

$$
\begin{aligned}
& \phi\left(\begin{bmatrix} x_{11} & x_{21} & x_{31} \\ x_{12} & x_{22} & x_{32} \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}\right) \\
= \;& \phi\left(\begin{bmatrix} w_{11}x_{11} + w_{21}x_{21} + w_{31}x_{31} & w_{12}x_{11} + w_{22}x_{21} + w_{32}x_{31} \\ w_{11}x_{12} + w_{21}x_{22} + w_{31}x_{32} & w_{12}x_{12} + w_{22}x_{22} + w_{32}x_{32} \end{bmatrix}\right) \\
= \;& \begin{bmatrix} h_{11} & h_{21} \\ h_{12} & h_{22} \end{bmatrix}
\end{aligned}
$$

$$
\phi\left(\begin{bmatrix} h_{11} & h_{21} \\ h_{12} & h_{22} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}\right) = \phi\left(\begin{bmatrix} w_1 h_{11} + w_2 h_{21} \\ w_1 h_{12} + w_2 h_{22} \end{bmatrix}\right) = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}
$$

- Multilayer network with batches:

- Input to a layer: ***batch*** different ***dim(x)***-dimensional input ve
- Output of a layer: ***batch*** different ***dim(y)***-dimensional output
  - *dim(y)* is the number of neurons in the layer (1 output per neuron)
- Process of converting input to output:
  - Multiply the (*batch*, *dim(x)*) input matrix with a (*dim(x)*, *dim(y)*) wei
    The result has shape (*batch*, *dim(y)*).
  - Apply some non-linear function (e.g. sigmoid) to the result.
    The result still has shape (*batch*, *dim(y)*).
- Big idea: Stack inputs/outputs to batch them
  - The multiplication by weights and non-linear function will be applie
    (data point in the batch) separately.