- What is Machine Learning?

↳ Field of Study that gives computers the ability to learn <u>without being explicitly programmed.</u>

↳ Well-Posed Learning Problem:

↳ A computer program is said to learn from E if its P on T is improved after E.

↳ E: experience, P: performance measure, T: task.

- Machine Learning Algorithms:

↳ Gradient Descent:

↳ Iterative Optimization algorithm to find the (local) minimum of a function.

↳ Outline:

1) Start with some guessed input values for the attributes of the function.
2) Keep changing / tweaking each input value (simultaneous update) & compare rate of change of the output value from the previous set of input values.

↳ Compare input values around the previous input value.

↳ Actual Algorithm:

1) $\forall \theta_j$, where $j = 0 \ldots n$, choose initial values.
2) $temp_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta, \ldots \theta_n) \Rightarrow temp_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^i) - y^i) x_j^i \quad \forall\ j = 0 \ldots n$ * α influences step size (learning rate)
3) $\theta_j := temp_j$ for $j = 0 \ldots n$.
4) Repeat (2) & (3) until convergence. * Will converge even with fixed learning rate.

↳ Normal Equation / Linear Least Squares

↳ Analytive Optimization algorithm to find the (local) minimum of a function.

↳ Actual Solution:

1) $\forall \theta_j$ within $J(\theta_0 \ldots \theta_n)$

$$\Rightarrow \frac{\partial}{\partial \theta_j} J(\theta_0 \ldots \theta_n) = 0$$

* Where the rate of change w.r.t θ_j would be 0, given an infinitesimally small change of value in θ_j.

OR

2) Put the dataset into a matrix of dimension (number of training examples × (number of input attributes + 1)), X, & the outputs into a (number of training examples) - dimensional vector, y, & resolve the following (number of training examples) - dimensional vector, θ,

$$\theta = (X^T X)^{-1} X^T y$$

↳ If $X^T X$ is singular / non-invertible:

a) Redundant input attributes (if one input attribute is linearly-dependant to another)

b) Too many input attributes → use regularization.

* // Pseudoinverse property

$$X\theta = y$$
$$X^T X \theta = X^T y$$
$$(X^T X)^{-1}(X^T X)\theta = (X^T X)^{-1} X^T y$$
$$I\theta = (X^T X)^{-1} X^T y$$
$$\theta = (X^T X)^{-1} X^T y$$

↳ Slow to compute $(X^T X)^{-1}$ for datasets with large number of input attributes.

↳ $n \leq 10000$

↳ $(O(n^3))$

↳ Supervised Learning : Actively / Manually teaching the computer to do something. *Labeled → Predictions

↳ Algorithm given a training dataset with the right answers given (relationship between each input instance & output defined), and generates a function (a.k.a hypothesis / model).

↳ Through assuming Inductive Bias, the model can be used to approximate for real-world outputs for previously unseen input instances.

↳ Problems:

*Instances : Set of all attributes.

↳ Regression Problem: Predict a continuous value output.

↳ Linear Regression with One Variable / Univariate Linear Regression.

*One Input, One Output

↳ Use a "hypothesis" function to model the best-fit straight line for a dataset.

↳ $h_\theta(x) = \theta_0 + \theta_1 x \approx y$

↳ Need to determine θ_0 & θ_1, such that model is reasonably close to the examples in the training set.

↳ Minimization problem involving $\frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^i)-y^i)^2$ (Squared Error Cost Function), where (x^i, y^i) is the ith example & m is the size of the dataset.

↳ Utilize Gradient Descent to minimize Cost Function.

↳ The Cost Function for Univariate Linear Regression is a Convex Function, hence local minimum = global minimum.

↳ "Batch Gradient Descent": Each step of gradient descent uses all points in the dataset.

↳ Alternatively, use the "Linear Least Squares" method to locate optimum minimum.

↳ Linear Regression with Multiple Variables / Multivariable Linear Regression

*Multiple Inputs, One Output.

↳ Hypothesis Function : $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \dots \theta_n x_n \approx y$

$$x^i = \begin{bmatrix} x_1^i \\ x_2^i \\ \vdots \\ x_n^i \end{bmatrix} = \text{ith input vector with } n \text{ inputs (dimensions)}$$

↳ Can define $x_0 = 1$ to simplify notation

$\Rightarrow h_\theta(x) = \theta^T x$, where θ & x are n+1 dimensional vectors.

↳ Simplifies Cost Function definition :

$$\Rightarrow \frac{1}{2m}\sum_{i=1}^{m}(\theta^T x^i - y^i)^2$$

↳ Optimization Techniques for performing Gradient Descent involving ≥ 1 input attributes.

1) Ensure each input attribute is similarly scaled (normalize them to approximately a −1 to +1 range).
2) Ensure each input attribute is translated by the mean value of the input attribute.

$$x_1 \Rightarrow \frac{x_1 - \mu_1}{s_1}$$

3) Choose learning rate α s.t the evaluated value of the Cost Function decreases with each subsequent iteration.

*s_1 can be range (max−min) or standard deviation.

↳ Test with $\alpha = 1,\ 0.1,\ 0.01,\ \dots$ etc.

↳ Polynomial Regression :

↳ Can become Linear Regression when the exponentiation of the input attribute is applied to the input attribute's value before evaluation of the Cost Function.

↳ May become impractical when dealing with problems with too many input attributes.

↳ Regularized Linear Regression :

↳ Cost Function : $\frac{1}{2m}\left(\sum_{i=1}^{m}(h_\theta(x)-y)^2 + \sum_{j=1}^{n}\theta_j^2\right)$ *θ_0 not included in regularization term.

↳ Gradient Descent Update:

↳ Each Iteration:

1) $\theta_0 := \theta_0 - \frac{\alpha}{m}\sum_{i=1}^{m}\left((h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)}\right) \Rightarrow \theta_0 - \frac{\alpha}{m}\sum_{i=1}^{m}\left((h_\theta(x^{(i)}) - y^{(i)})\right) \quad \because x_0 = 1\ \forall i$

2) $\theta_j := \theta_j - \frac{\alpha}{m}\left(\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} + \lambda\theta_j\right) \Rightarrow \theta_j\left(1 - \frac{\alpha\lambda}{m}\right) - \frac{\alpha}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$

↳ Normal Equation :

$$\theta = \Big(X^TX + \lambda\underbrace{\begin{bmatrix} 0 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{bmatrix}}_{\text{Account for } \theta_0 \text{ term}}\Big)^{-1}X^Ty$$

↳ $\left(X^TX + \lambda\begin{bmatrix} 0 & & \\ & \ddots & \\ & & 1 \end{bmatrix}\right)$ is guaranteed to be invertable, as long as $\lambda > 0$.

↳ Classification Problem : Predict a discrete value output.

* Classification isn't actually a linear function.

* Sigmoid (Logistic) Function = $\frac{1}{1+e^{-z}}$.

↳ Binary Logistic Regression :

↳ $h_\theta(x) = \frac{1}{1+e^{-z}}$, where z is the output from $\theta^T x$.

↳ Estimated probability that $y=1$ on input x, given x parameterized by θ.

↳ $P(Y=0|x;\theta) + P(Y=1|x;\theta) = 1$.

↳ $h_\theta(x) = 1$ when $z = \theta^T x \geq 0$, & $h_\theta(x) = 0$ when $z = \theta^T x < 0$.

↳ The Decision Boundary is defined by the values of x for which the training set is effectively classified.

↳ Cost Function :

↳ $h_\theta(x)$ is not a linear function, so the Cost Function involving $h(\theta)$ used in Linear Regression would not be suitable here.

↳ Non - Convex : Local Optima ≠ Global Optima // "Convexity Analysis"

↳ $\frac{1}{m}\sum_{i=1}^{m} f(h_\theta(x^{(i)}), y^{(i)})$, where

$$f(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y=1 \\ -\log(1-h_\theta(x)) & \text{if } y=0 \end{cases}$$ // Principle of Maximum Likelihood Estimation.

↳ $f(h_\theta(x), y) = -y .* \log(h_\theta(x)) - (1-y) .* \log(1-h_\theta(x))$, ∵ y can only ever be 0 or 1.

↳ Utilize Gradient Descent to minimize Cost Function.

↳ $\theta := \theta - \frac{\alpha}{m} X^T(g(X\theta) - y)$ // where g applies the Sigmoid Function element wise.

↳ Other Algorithms : Conjugate Gradient, BFGS & L-BFGS.

↳ Multiclass Logistic Regression :

↳ One-vs-All Classification :

↳ Essentially do Binary Logistic Regression for each class in the training set

↳ End up with $h_\theta(x)$ for each class.

↳ On a new input instance x to make a prediction :

↳ Tie x to the class whose associated $h_\theta(x)$ gives the highest output.

Regularized Logistic Regression :

↳ Cost Function : $-\left(\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)} .* \log(h_\theta(x^{(i)})) + (1-y^{(i)}) .* \log(1-h_\theta(x^{(i)}))\right)\right) + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$ // .* denotes element-wise multiplication.

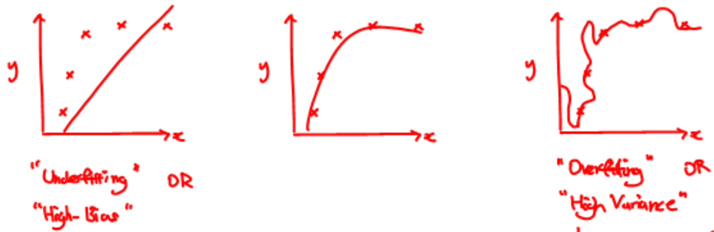
↳ Gradient Descent Update :

↳ Each Iteration :

1) $\theta_0 := \theta_0 - \frac{\alpha}{m}\sum_{i=1}^{m}\left(\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_0^{(i)}\right) \Rightarrow \theta_0 - \frac{\alpha}{m}\sum_{i=1}^{m}\left(\left(h_\theta(x^{(i)}) - y^{(i)}\right)\right)$ ∵ $x_0 = 1 \; \forall i$

2) $\theta_j := \theta_j - \frac{\alpha}{m}\left(\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)} + \lambda\theta_j\right) \Rightarrow \theta_j\left(1 - \frac{\alpha\lambda}{m}\right) - \frac{\alpha}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)}$

- Overfitting : When the learned hypothesis seems to fit the (current) data well, but actually doesn't, and will fail to fit unseen instances.
  - Comes about from having too many attributes.
  - Example:



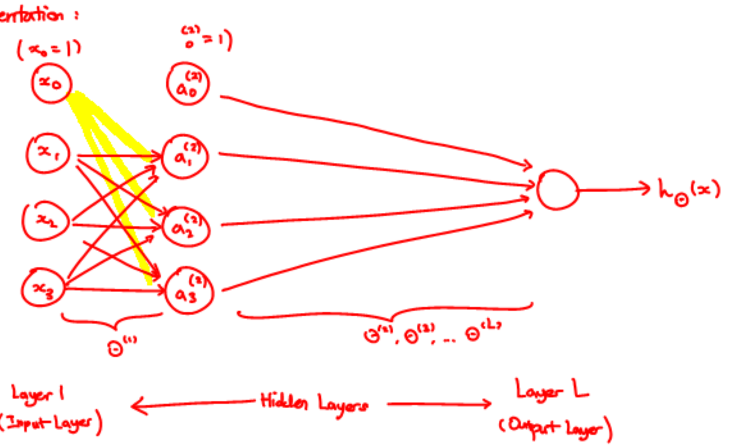
- The space of possible hypothesis functions is too large.
- Not enough data plots to constrain it.

- Solutions:
  1) Reduce the number of input attributes:
     - Manually, or via Model Selection Algorithm.
     - May throw away useful information.
  2) Regularization :
     - Reduce the magnitude of some higher order parameters θ (or all of them).
       - Modify the Cost Function to take increased contribution from those parameters θ.
       - Example :
         - $\frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x)-y\right)^2 + 1000\theta_3^2$
         - $\frac{1}{2m}\left(\sum_{i=1}^{m}\left(h_\theta(x)-y\right)^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right)$, where λ is the regularization term. * θ_0 not included.
           - Make the Cost Function directly account for the magnitude of the parameters.
           - λ should not be too large - will result in underfitting.
       - "Simplifies" Hypothesis.
     - Works well when there is a lot of input attributes, with each contributing a bit to predicting y.

↳ Neural Networks:

↳ Useful for large, non-linear hypothesises with many input attributes.

↳ Representation:



↳ Where:

↳ L is the total number of Layers in a network, S_ℓ is the number of neuron units in the network in layer ℓ, excluding bias unit.

↳ $\Theta^{(\ell)}$: matrix of parameters controlling function mapping from layer ℓ to layer $\ell+1$ // Θ is an array of (not necessarily equal dimension) matrices.

↳ $(\Theta^{(\ell)})^T$: matrix of parameters controlling function mapping from layer $\ell+1$ to layer ℓ.

↳ $\Theta_{ij}^{(\ell)}$: A modifier value for the strength of the link between $a_j^{(\ell)}$ & $a_i^{(\ell+1)}$.

↳ Dimension : $S_{\ell+1} \times ((S_\ell)+1)$ // Include the Bias Unit for the current layer.

↳ $a_j^{(\ell)}$: function of neuron j in layer ℓ

↳ $a_j^{(\ell)} = g(\Theta_{j0}^{(\ell-1)} x_0 + \Theta_{j1}^{(\ell-1)} x_1 + \ldots + \Theta_{jn}^{(\ell-1)} x_n) \Rightarrow a_j^{(\ell+1)} = g(\Theta_{j0}^{(\ell)} x_0 + \Theta_{j1}^{(\ell)} x_1 + \ldots + \Theta_{jn}^{(\ell)} x_n)$ * $a^{(1)} = x$.

↳ $a_0^{(\ell)} = 1$

↳ $h_\Theta(x) : a_0^{(\ell)} = g(\Theta_{j0}^{(\ell-1)} a_0^{(\ell-1)} + \Theta_{j1}^{(\ell-1)} a_1^{(\ell-1)} + \ldots + \Theta_{jn}^{(\ell-1)} a_j^{(\ell-1)})$

Usage:

↳ Utilize the "Feedforward" Propagation approach to make a prediction with a trained model.

↳ $h_\Theta(x)$ (or $a^{(\ell)}$) $= g(\Theta^{(\ell-1)} a^{(\ell-1)})$

↳ Intuition:

↳ One Logistic Unit can simulate a logic gate:

↳ Example: Assuming Binary Input Values:

↳ AND: $a(x) = g(-30 + 20x_1 + 20x_2)$, $\Theta^{(1)} = [-30 \ \ 20 \ \ 20]$

↳ OR: $a(x) = g(-10 + 20x_1 + 20x_2)$, $\Theta^{(1)} = [-10 \ \ 20 \ \ 20]$

↳ NOT: $a(x) = g(10 - 20x_1)$, $\Theta^{(1)} = [10 \ \ -20]$

* Sigmoid / Logistic Function: $g(z) = \frac{1}{1+e^{-z}}$

As $z \to -\infty$, $g(z) \to 0$, $z \to \infty$, $g(z) \to 1$

* Bias Unit: $x_0 / a_0^{(\ell)} = 1$

↳ Layers of Logistic Units simulating logic gates can output values corresponding to non-linear hypothesis (non-linear decision boundaries)

↳ Example:

↳ XNOR: NOT XOR: ((NOT x_1) AND (NOT x_2)) OR (x_1 AND x_2) : (x_1 NOR x_2) OR (x_1 AND x_2), $\Theta^{(1)} = \begin{bmatrix} 10 & -20 & -20 \\ -30 & 20 & 20 \end{bmatrix}$ $\Theta^{(2)} = [-10 \ \ 20 \ \ 20]$

↳ Consider that the units in each successive Layer creates increasingly complex outputs - based on combinations of outputs basic outputs from the previous Layer multiplied by differing weights.

↳ Example: Pixels → Patterns → Digit Parts → Digits.

↳ No longer constrained to having to use the input attributes directly to form the final hypothesis.

↳ Gets to "learn its own attributes".

↳ Multiclass Regression Problems:

↳ Essentially an extension of the one-vs-all method.

↳ Output layer would now have multiple output units, each corresponding to a class.

↳ Each output unit outputs the probability whether the inputs should be classified under the class they are in charge of.

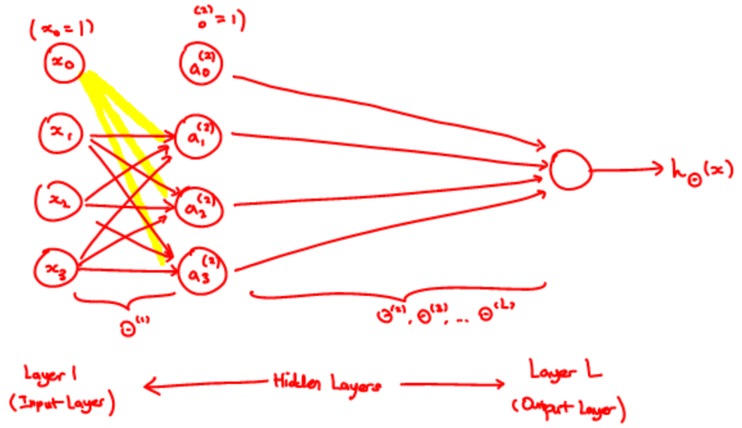
* Neuro-Rewiring:

↳ If the same piece of brain tissue can learn different functions, maybe it uses only one algorithm.

* A neuro network attempts to simulate the brain structure:

↳ A neuron takes in some inputs via dendrites and outputs an electrical charge along an axon to another neuron based on some computation done on the inputs.

neuron more generally uses a "rectifier" function.



- Training:
  - Cost Function: $-\frac{1}{m}\left(\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log\left(h_\Theta(x^{(i)})\right)_k + (1-y_k^{(i)})\log\left(1-h_\Theta(x^{(i)})\right)_k\right) + \frac{\lambda}{2m}\left(\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(\Theta_{ji}^{(l)}\right)^2\right)$

    * Apparently this Cost Function has many local minima, but all of equal quality.
    - Goal: Minimize Cost Function, $J(\Theta)$.
  - Utilize the "Back" Propagation approach to obtain trained values for each parameter.
    - Intuition:
      - $\frac{\partial}{\partial\Theta^{(l)}} J(\Theta^{(l)})$ would represent by how much each link between any 2 units between layer l & $l+1$ should be strengthened / weakened by to have the greatest minimizing impact on $J(\Theta)$.
      - If we want to alter the output of one neuron, we can, based on how it calculates its output:

        $a_i^{(l)} = g(\Theta_0^{(l-1)} a_0^{(l-1)} + \Theta_1^{(l-1)} a_1^{(l-1)} + \ldots + \Theta_n^{(l-1)} a_n^{(l-1)})$

        1) Tweak the strength of the links that connect to unit $a_i^{(l)}$.
           - Done through application of $\frac{\partial}{\partial\Theta^{(l)}} J(\Theta^{(l)})$ onto the current configuration of $\Theta^{(l)}$.
        2) Tweaking the strength of the neurons in the previous layer.
           - Implies recursive nature / backtracking through the network to perform (1) on prior hidden layer neurons.
    - Algorithm:
      1) Initialize $\Theta^{(l)}$ to contain random values $\forall l$ to break potential symmetry.
      2) Set $\Delta^{(l)} = 0 \; \forall l$, where $\Delta^{(l)}$ is a matrix representing the accumulated error w.r.t links between layers l & $l+1$.
         i) $\forall$ training examples $t = 1 \ldots m$ as vectors, $(x_1^{(t)}, x_2^{(t)}, \ldots, x_n^{(t)}, y_1^{(t)}, y_2^{(t)} \ldots y_K^{(t)})$
            - Set vector $a^{(1)}$ to be $x^{(t)}$, perform forward propagation to compute $a^{(l)}$ for $l = 2 \ldots L$ via $g(\Theta^{(l-1)} a^{(l-1)})$. // Remember to add bias.
            - Compute vector $\delta^{(l)}$, which represent the error of units in Layer l.
              - Error in Output Layer: $\delta^{(L)} = a^{(L)} - y$
              - Error in lth Layer: $\delta^{(l)} = (\Theta^{(l)})^T \delta^{(l+1)} .* g'(z^{(l)})$ // Remember to remove bias, & $l > 1$.

                $\Rightarrow \delta^{(l)} = (\Theta^{(l)})^T \delta^{(l+1)} .* (a^{(l)} .* (1 - a^{(l)}))$ // Assuming g was the Sigmoid Function.
            - $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$. // Column Vector × Row Vector
         ii) Given training examples $t = 1 \ldots m$ as a matrix (X is a matrix, Y is a matrix)
            - Set matrix $a^{(1)}$ to X, perform forward propagation to get matrices $a^{(l)}$, $l = 2 \ldots L$ via $g(a^{(l-1)} \Theta^{(l-1)T})$ // Remember to add bias
            - Compute matrices $\delta^{(l)}$, which represent errors of units in Layer l w.r.t each example in rows.
              - Error in Output Layer: $\delta^{(L)} = a^{(L)} - Y$.
              - Error in lth layer: $\delta^{(l)} = \delta^{(l+1)} * \Theta^{(l)} .* g'(z^{(l)})$ // Remember to remove bias, & $l > 1$

                $\Rightarrow \delta^{(l)} = \delta^{(l+1)} * \Theta^{(l)} .* (a^{(l)} .* (1 - a^{(l)}))$ // Assuming g was the Sigmoid Function.
            - $\Delta^{(l)} := \delta^{(l+1)} a^{(l)}$ / Matrix × Matrix.
      3) $D_{ij}^{(l)} := \frac{1}{m}\left(\Delta_{ij}^{(l)} (+ \lambda \Theta_{ij}^{(l)} \text{ if } j \neq 0)\right)$, where $D_{ij}^{(l)} = \frac{\partial}{\partial\Theta_{ij}^{(l)}} J(\Theta_{ij}^{(l)}) \; \forall l = 1 \ldots L$
         - Do manual computation of the gradient here to see if the above implementation actually works.
      4) Apply $D^{(l)}$ to $\Theta^{(l)} \; \forall l = 1 \ldots L$, & repeat (2) for the next iteration.
    - Optimizations:
      1) Instead of iterating through all the training examples at once, do it in randomly shuffled mini-batches instead.
- Architecture Considerations:
  - No. of Input Units = No. of Input Attributes.
  - No. of Output Units = No. of classifiers.
  - No. of Hidden Layers = Usually 1 is enough.
  - No. of Hidden Units / Hidden Layer: $\frac{\text{Num of Training Examples}}{(\alpha * (\text{Num of Input Units} + \text{Num of Output Units}))}$, where $\alpha = [2, 10]$.

↳ Unsupervised Learning : The computer learns by itself.

* Unlabeled → Labeled

- ↳ Algorithm given a dataset with no labels / no right answers given (can it find all structures within this dataset)
  - ↳ Allows us to approach problems with little / no idea of what our results should look like.
  - ↳ Clustering Algorithm : Identifies implicit data point groupings within a dataset.
  - ↳ Cocktail Party Algorithm.