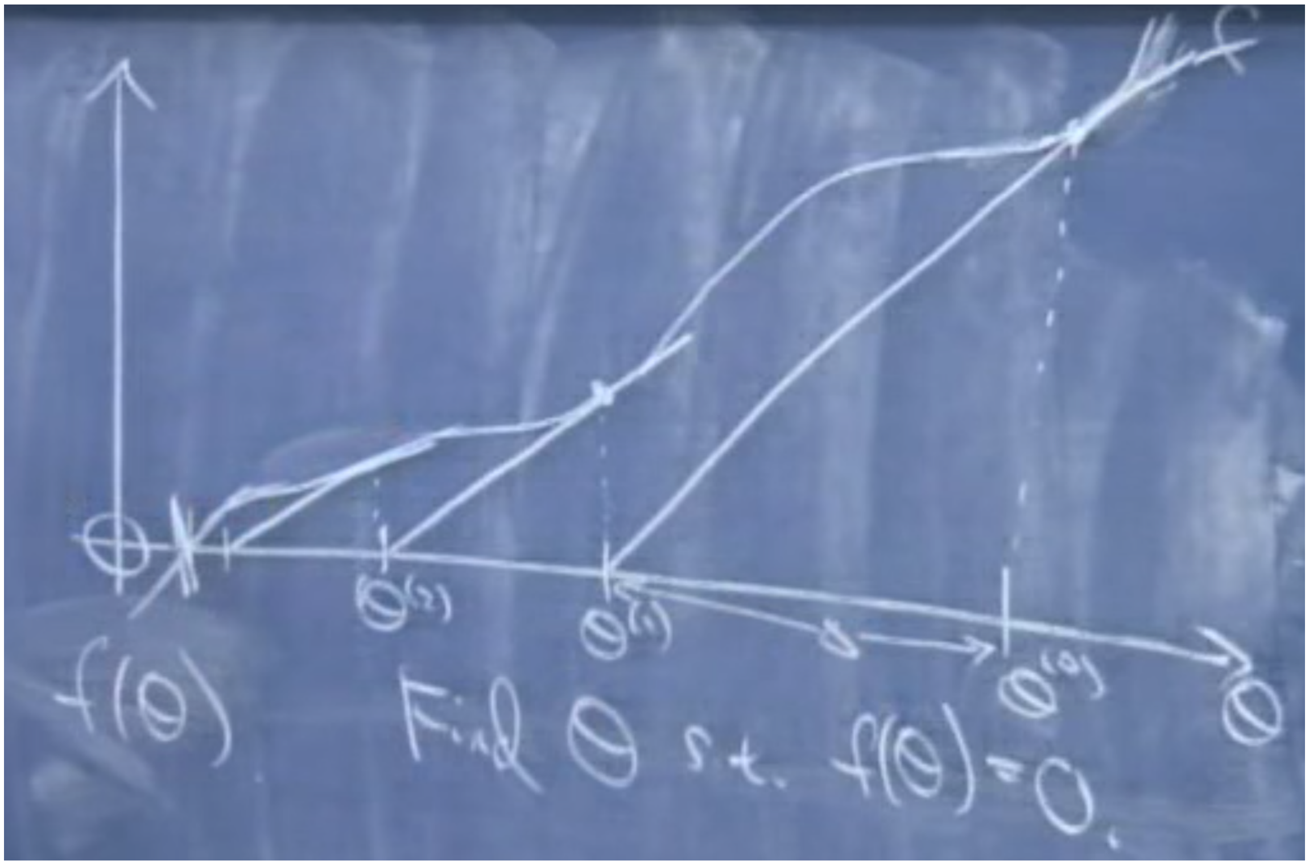
regression, gradient descent, linear models

- machine learning as defined by Tom Mitchell is getting better at a task according with some performance measure as the agent gets more experience
- supervised learning is when we give the algorithm a bunch of data and right answers and then have the agent predict the answer based on new data.
- broad categories in supervised learning is classification and regression: predicting a continued valued function is called regression. if its a discrete, finite set of things, its called classification. (these definitions are rough)
- learning theory is the theory of how and why learning algorithms work. it turns out we can prove certain theorems about probability of success rates of learning algorithms, understand how much data we need, under what circumstances (input distributions) a certain algorithm will do vs not so much and other useful things.
- having said that theory of learning algorithms is one thing but another thing is the art of becoming really good at training models, picking models, getting data etc. this is an art in itself and its very important to get good at this.
- unsupervised learning: “given some data would you please find interesting patterns in this data for me?”. example applications: 3d structure from 2-d image, cocktail party problem (removing background noise and separating out speaker voices). again, the question here is “would you please find structure in this data?”
- reinforcement learning: the problem formulation here is usually to take some sequence of actions in an environment so as to maximize some long terms core. eg: learning a controller for keeping a helicopter stable, playing a game where you find out you die or live at the end after a sequence of actions, where you don’t know which single action is good/bad.
- some notation:
  - m to denote the number of training examples
  - n is the number of features
  - note the m x n matrix with columns features and each row a training example 😃
  - $\vec{x}$ the input features
  - y the output/target variable
  - i^{th} training example as x^i, y^i
- The hypothesis is just another word for the result of training a supervised learning algorithm. a hypothesis, once trained, takes in input data and gives a prediction (either classification or regression)
- if we take a bunch of x, y, z data where x, y is the features and y is the output/target, we can construct a linear hypothesis like $z = \theta_0 + \theta_1 x + \theta_2 y$. the θ’s are called the parameters of the learning algorithm (or the hypothesis). we want to “learn” an appropriate θ’s.
- We will call $h_\theta(x)$ the output. remember, y is the training example. one thing we could minimize for a linear model is $J(\theta) = \frac{1}{2}\sum(h(x_i) - y_i)^2$. the $\frac{1}{2}$ is just convention.
- a couple of algorithms to find a good parameter vector θ:
  - gradient descent: note that we are trying to minimize $J(\theta)$. in general, this J is not convex. turns out the for the sum of least square cost for a linear hypothesis, as defined above, it is convex. exercise: prove this convexity of linear least squares.

  - in general, gradient descent depends on where we start out descending so we could very well end up at a local optimum.
  - in multivariable calc, we saw that the direction of steepest descent (gradient) is given by the vector $(f_x, f_y..)$
  - in gradient descent, we will basically take a step in the direction of greatest descent which is accomplished by starting at some θ and updating each component as $\theta_i := \theta_i - \alpha\frac{\partial J(\theta)}{\partial \theta_i}$ to iteratively reduce $J(\theta)$.
  - α is called the learning rate. for now, we will manually set it. the tradeoff here is that if its too small it will take too long to converge and if its too large it might overshoot the minimum.
  - at each step of iteration, if we do a update over the entire training set, it is called batch gradient descent.
  - If m is huge, we do something called stochastic gradient descent where at each step of iteration, we do an update over just one training example. so to do one update of θ we look at just one training example instead of all. this way we might wander around a little bit sometimes going up hill, but we are taking a step much much faster. in practice this turns out to be much faster than the batch version.
  - when should we use a linear hypothesis? if you plot your data and see a linear pattern! which algorithm to use, in general, dependent on the distribution of the dataset.
  - If we have the gradient vector $\nabla_\theta J$, we can write the update step as a vector equation:
    $\theta := \theta - \alpha\nabla_\theta J$
- if we have a real valued function of a matrix, its gradient is just the matrix with the same shape, with partial derivatives at the corresponding position.
- $A^TAx = A^Tb$ is a closed form solution of the linear least squares that we saw in the linear algebra class.
- Instead of a linear model like $\theta_0 + \theta_1 x$, we could have chosen a quadratic one, $\theta_0 + \theta_1 x + \theta_2 x^2$, or a super high degree polynomial like $\theta_0 + \theta_1 x + \theta_2 x^2 + +\theta_3 x^3 + \theta_4 x^4$. Point is choosing a super high degree polynomial will "overfit" to the idiosyncracies of the dataset but if there's not enough parameters and degrees, we might miss obvious patterns in the training data, which is "underfitting". these are both bad, and we will discuss them more formally later but notice that there is this issue of selecting features.
- the models we've discussed above have a fixed, finite number of parameters, so they are called parametric algorithms. non-parametric algorithms are those where the number of parameters grows with the data. in other words, even after learning, the amount of stuff (parameters) it will have to remember is a function of the size of the training set (some increasing function, usually linear). these algorithms are so often so that their performance is not super dependent on our choice of features. eg. locally weighted regression (lowess)
- if you have data that doesn't seem obvious what kind of model it fits (linear, quadratic etc), lowess might be a promising option. a simple version of lowess basically fits a linear regression line in a small neighborhood of the input and uses that linear model to predict the output. It does this by giving far away points a very low weight by choosing a J like$J(\theta) = \frac{1}{2}\sum w_i(h(x_i) - y_i)^2$ where $w_i = e^{\frac{-(x_i-x)^2}{2\tau^2}}$. If τ is large, then weights wane off relatively slowly and vice versa. τ is called the bandwidth parameter.
- Of course this is just one weight function. there are many other choices depending on how much more/less close/far points should be weighted for that data.
- Note the crazy thing with weighting: the weights depend on input x, so every time we try to do an inference we have to fit the weights. so basically we have to remember the data. this is not atypical for nonparametric learning algorithms. there are ways to make this more efficient so we don't discuss here.

- why least squares error:
  - in linear regression, we could think of the true function as $y = \theta^T x + \epsilon$ where ϵ is some error that's independent of our features but might depend on some other uncaptured features. we'll put a gaussian model on ϵ with mean 0, variance σ^2. Using these two facts, that means the data is really $N(\theta^T x, \sigma^2)$. (gaussian, btw, just turns out to be a good error model in a lot of cases, maybe due to central limit theorem which says the sum of iid random variables is gaussian). the errors here are iid normal mean 0 and variance σ^2. Now, MLE is a thing that chooses the parameters so as to choose the data as probable as possible. Note that we have data and we have a model $y = \theta^T x + \epsilon$, and we are trying to find θ's so we maximize the probability of our training set. Taking the log likelihood and differentiating, it turns out that to maximize the probability of our training set, we have to minimize... the least squares error! That is a mathematical justification of least squares cost. Note that this is based on the gaussian error model which is "justified" by central limit theorem. So least squares is maximum likelihood assuming gaussian error in data!
- as a side point, ng says something about the bayesian view being that θ is fixed unknown, but the frequentist approach is to think of θ as a random variable.
- next we'll do a binary classification example $y \,\epsilon\, 0, 1$. we could use linear regression for classification problems and but it turns out linear regression is quite sensitive to outliers and sometimes isn't a natural fit.
- first lets define a function called sigmoid function (also called logistic function). we will justify this later but the definition is: $h_\theta(x) = \frac{1}{1+e^{-z}}$ where $z = \theta^T x$. it looks like a S. note that this lies between (0,1) and is asymptotic on both ends. one interpretation of this is $p(y = 1|x; \theta) = h_\theta(x)$ and $p(y = 0|x; \theta) = 1 - h_\theta(x)$. this can be written as $p(y|x; \theta) = h(x)^y (1 - h(x))^{1-y}$. with that probability interpretation, we proceed to maximize the likelihood → that is find θ that maximizes the probability of training data. the likelihood function is a product of many sigmoids, so we take log likelihood and maximize that. the function to minimize turns out to be of a similar form as the J in regression, but not the same.
- Note that logistic regression is trying to draw a straight line (in high dimensional space) that separates the data into two parts. This is because we are learning θ so that $\theta^T x$ is a good classifier. If its big its one class and if it is small it is another class.
- we will just mention the perceptron algorithm here and come back to it later:
  - the perceptron function is defined as $g(z) = 1$ if $\theta^T x \geq 0$, else $g(z) = 0$. note that this is a step function with a a discontinuity at $z = \theta^T x = 0$
  - the perceptron hypothesis is just $h_\theta(x) = g(\theta^T x)$.
- we've used stochastic and batch gradient descent till now to train stuff: linear and logistic regression. now we'll look at another way to train them: newton's method.



- newton's method helps us find the roots the x's for which $f(x) = 0$. in the convex case, we'll take the log likelihood and make its derivative 0 to get the maximum.
- we start at a point, find the derivative, extend that to intersect with the x-axis, then pick that as the new x and keep going until we converge to a point close enough to the root.
- there is also a multivariate version of this that Andrew doesn't prove/give intuition for 😕. It uses the hessian.
- this often runs faster than gradient descent (think why). it basically skips the intermediate steps and takes large steps initially and smaller steps as we get closer to 0.
- andrew ng defines something called the exponential class of distributions and generalized linear models which we don't go into here.
- softmax regression is a generalization of logistic regression to K classes, instead of binary classification (2 classes) like we saw in logistic regression.

$$h_\theta(x) = \begin{bmatrix} P(y=1|x;\theta) \\ P(y=2|x;\theta) \\ \vdots \\ P(y=K|x;\theta) \end{bmatrix} = \frac{1}{\sum_{j=1}^{K} \exp(\theta^{(j)\top} x)} \begin{bmatrix} \exp(\theta^{(1)\top} x) \\ \exp(\theta^{(2)\top} x) \\ \vdots \\ \exp(\theta^{(K)\top} x) \end{bmatrix}$$

- Note that the vector above sums up to 1, as expected

- the cost function we use is:

$$J(\theta) = -\left[\sum_{i=1}^{m}\sum_{k=1}^{K} 1\left\{y^{(i)} = k\right\} \log \frac{\exp(\theta^{(k)\top} x^{(i)})}{\sum_{j=1}^{K} \exp(\theta^{(j)\top} x^{(i)})}\right]$$

  The above is derived similar to how we derived a cost function for the logistic regression case: by computing the MLE. That is, by computing the θ_i's that maximize the probability of the data.
- When $K = 2$, softmax is just logistic regression.