ML Coursera summary

[linear regression (LR) 2](#_Toc19287416)

[logistic regression 2](#_Toc19287417)

[Neural Networks (NN) 4](#_Toc19287418)

[Forward propagation 4](#_Toc19287419)

[Backward propagation (BP, optimization stage) 5](#_Toc19287420)

[Built-in optimization functions (fminunc, etc) 8](#_Toc19287421)

[Test to see gradients make sense 8](#_Toc19287422)

[Rand. init. of thetas 8](#_Toc19287423)

[All Together – steps to build Neural Network 9](#_Toc19287424)

[Where to apply what 9](#_Toc19287425)

[Diagnostics 9](#_Toc19287426)

[Support Vector Machines (SVM) 12](#_Toc19287427)

[Why is there a margin in SVM? 13](#_Toc19287428)

[Kernals – Landmarks 13](#_Toc19287429)

[Using an SVM 14](#_Toc19287430)

[Unsupervised Learning 15](#_Toc19287431)

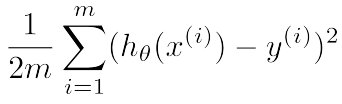
[Clustering algo #1: K-means algo 15](#_Toc19287432)

[Dimensionality reduction 15](#_Toc19287433)

[Anomaly detection 17](#_Toc19287434)

[Recommender systems 17](#_Toc19287435)

# linear regression (LR)

predict number from קבוצה רציפה based on features. try to get theta to fit examples as good as possible.

h**θ**(x) = x0theta0 + x1theta1 etc. or THETA\*X (matrix).

**J(theta)** - func that measures accuracy of hypothesis h(x). in LR: (see above). Called **square error func.**

**Note on Gradients:** you might think: “why not just go only in the direction of the most impactful feature in the function? E.g. for grad [2,1], just go only in x axis?”. Good question. The intuitive answer: you’re forgoing the whole other boost for a small addition in the “promising” axis. If it was that much better, the vector would naturally tend towards there. So, where’s the limit of forgoing one axis for the other to not lose out? **Exactly on the gradient**. Cool. (but why??)

**Gradient descent** – iteratively minimize J as a function of each theta, by reducing each theta by it’s gradient, i.e. the maximal reduction (vector opposite to gradient). In LinReg (and also LogReg):

Is gradient of each theta (the h func being different in Lin and Log regression).

**tricks for gradient descent** - normalize range, check learning rate with plot, seeing if cost always goes down as iterations get higher (otherwise make alpha smaller).

**Polynomial regression** - instead of linear, we can use many different curves that suit the dataset (new variables can be functions of old vars, e.g. x2 = x1­2), etc.

**Normal equation:** creating matrix X = [x1;x2;…], vector y [y1;y2;...], and calc derivative of J(theta)=0 (to find min/max + knowing J is convex = min) using linear algebra. We get that derivative of j(theta)=0 when theta=[XTX]-1XTY. see [here](https://github.com/siavashaslanbeigi/ml_notes/blob/master/docs/hand_written_notes/nb1_mult_linreg.pdf). Might be uninvertable, in which case use pinv or remove redundant features (e.g. two features describing height, one in ft the other meters).

# logistic regression

predict discrete value based on examples. use **Sigmoid/Logistic** funcion to normalize all values to a curve from 0 to 1. So:

**hθ (X)** = g(θ\*X). Y=1|h(x)>=0.5, else y=0. (g is sig func)

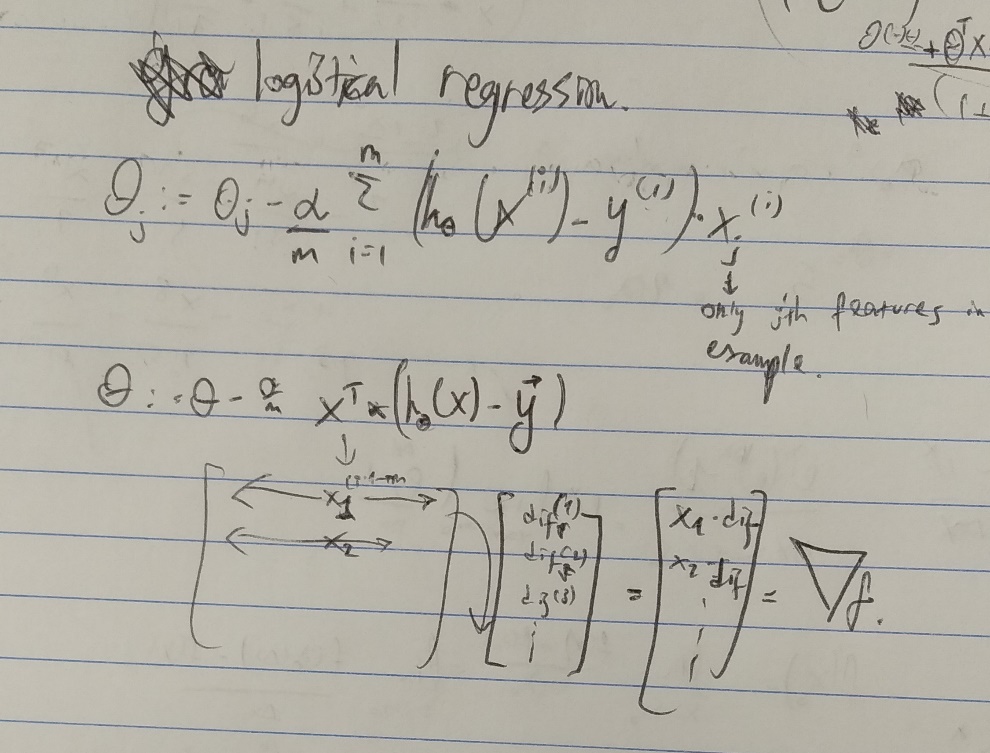
**decision boundary -** the boundary line created by THETA\*X (hypothesis func), such that one side of the line is larger than 0 (and since g(x)>0.5|x>0, we find that y=1), the other smaller (and there y=0). the training set isn't needed after we find the decision boundary, we just test against the boundary from then on.

**Cost(h**θ**(x),y) -** how much did h(x) miss from y. J is computed via Cost func. *J*(*θ*)=1*m*∑*i*=1*m*Cost(*hθ*(*x*(*i*)),*y*(*i*)).

in linear reg, is 1/2(h(x)-y)^2. would make logistic reg be non-convex (bc is applied over sigmoid and others), so use log. (y is the answer used from the training set).

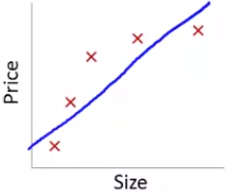
 or -ylog(h(x)) –(1-y)log(1-h(x)).

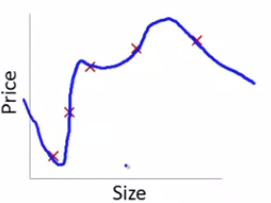
**Gradient descent (with matrix version) computations:**

****

**Conjugate gradient,BFGS,L-BFGS:** other optimization algos like gradient descent, faster (automatic alpha), more complex. Use software libs, find good ones.

**Multiple Classification algo: One Vs. All (LogReg):** do logistic regression hθi(x) on each category i being y=1 and the rest y=0, then for prediction on given example, prediction=max*i*(*h*(*i*)*θ*(*x*)), i.e. the class that was the most sure that the example is in his group.

**Underfitting or high bias -** having too little features for dataset.Has a bias to think the data grows linearly in spite of the data.

**Overfitting or high variance –** having too many features for dataset. too many curves, i.e. high variance (e.g. adding too many polynomials). Fits the data **too** well as to only be applicable to those examples, but not general enough for new examples.

underfitting

We want the fitting to be “just right”. HOW?

overfitting

Consider two overfittings:

1. Too many polynomials.
2. Many (let’s say linear) features. Different aspects of house that define price.

Fix:

1. Visualize data, see if has too many features. Only works for 1,2,3 dimension datasets.
2. Reduce no. of features (good for polynomial problem, but in house loses info that is useful.
   1. Manually find only the necessary features.
   2. Selection algo. Selects the right features for you.
3. Regularization: reduce magnitude of parameters. (good for houses, just use a bit of each one).

**Regression Regularization:** penalize theta params as to not be too high and create overfitting (option 2 in fixes). Concretely, change J(θ) =

i.e. add penalty for a theta being high. Need to be careful with lambda (**regularization parameter)** so it won’t be too high and we’ll get underfitting (thetas will just be really small and h(x) will be more or less linear), or too low and then we won’t actually penalize params.

Application:

1. **In grad descent**, we iteratively reduced J by reducing by negative grad. Here, the gradient turns out:

, and when factoring out theta, iterative function turns out:

.

****Since alpha, lambda and m are >0, we see intuitively that lambda reduces the size of theta each iteration (the penalization).

1. And in the **normal equation:** (where L is identity matrix except L(0,0)=0, since we don’t penalize theta0. In addition, the term in brackets is always invertable, thanks to lamda\*L.

# Neural Networks (NN)

## Forward propagation

The brain’s neurons have **dentrites**, that are inputs to the neuron. The cell then does some computation on those inputs, then outputs a pulse to the **axon(s)**, or the output(s).

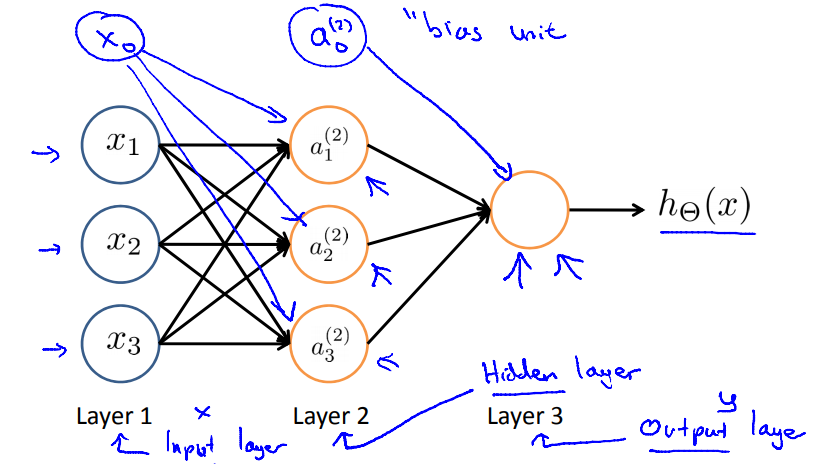
If we have a large input feature space and a non-linear hypotheses (diagram isn’t linear), using LogReg with quadratic or cubic features will create around n^2/2 or n^3 features, and becomes too hard to solve. That’s where NN comes in.

Instead, NN uses **layers**. Each unit in a layer has a logistic activation function (sigmoid func) from units in prev layer (with unique thetas).

**Bias unit** – node in prev layer, unit zero, which always has 1 as its value. Used for computing logistic func of next layer. It itself doesn’t have a logistic func though, as its always 1.

**activation nodes** - ‘a’ nodes are in hidden layers, and are called activation nodes.

**Big Theta Θ (capital letter this time)** - matrices are created per layer to hold thetas for all units in next layer on the current one. If layer j has sj units, and j+1 has sj+1 units (not including unit zero in each layer), then Θ(j) is an sj+1x(sj+1) matrix (+1 for zero unit or bias var in prev layer). So, **for example**: Θi,j(l) is the theta for the j’th feature in the i’th unit of layer l.



**Z(j) vector –** is equal to Θ(j-1)\*a(j-1)(including a0(j-1)). Then, aj (not including a0j)= g(zi) (instead of having to write the usual way, i.e aj=g(Θ(j-1)\*a(j-1)), and we get the more common g(z) display used in logistical functions. Meaning, every unit in layer “receives” its Z, and all the unit does is compute g(Z) as its output.

Example – **Boolean logic**: for classifying an OR gate (at least around OR, i.e. 0.9 and 0 is good too), we’d use Θ=[-10,20,20] for input layer, and no hidden layers. g(z)~1 from 4.6 onward, same for 0 with -4.6. in addition, z (from sigmoid) >=10 iff x1 or x2, otherwise z=-10. And so finally, using the weights above, hΘ(x)=1 iff x1 OR x2. Using more layers can allow you to classify more complex logic, like XNOR (layer1-2 does AND, NAND, and L2-3 does OR) or XOR.

**Multiple Classification –** use multiple output units and use an algo like one vs. all. hΘ(x) gives a vector of output size, and class selected is that which is most sure (highest prob) that the correct output is him.

## Backward propagation (BP, optimization stage)

**L –** no. of layers in net.

**Sl** – no. of units in layer l (not including bias unit).

**K = SL** =no. of units in output layer.

**Cost func**:

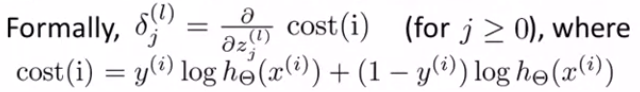
Since **y** is a vector e.g. [0; 0; 1; 0], we check not only that the correct classification was correct, but also that the others were wrong. That is done with the k sigma part. Yk and h(x)k represent output no. k or element k in the y vector. Triple sigma at end is sigma of all big theta values from all layers.

**Finding for gradient**

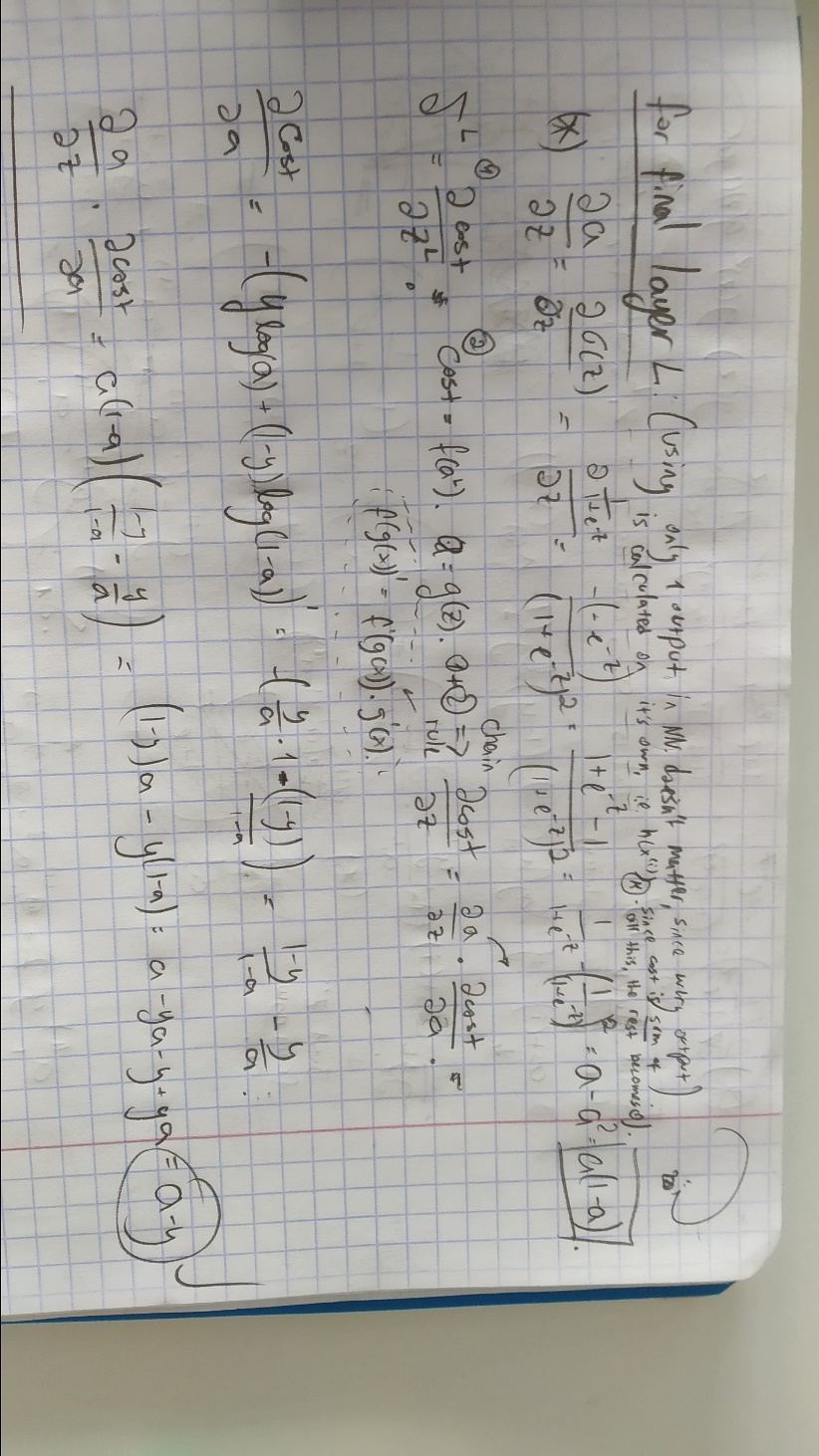
**Reminder:** in LinReg/LogReg gradient descent, we use a function to calc the error of the hypothesis answer given in h(x) in respect to the actual answer y times the X of current parameter (to see how much of the error the theta should account to himself, according to my intuition).

In NN, you can’t see the error of non-last layers, bc we don’t know the optimal activation unit of the next layer yet, since it is only a piece in the puzzle of the final layer, which we don’t know the optimal value (a\*theta) of yet. So, first we need to find the error of the last layer (with multiple outputs maybe) in respect to y (possibly a vector for multiple output NNs).

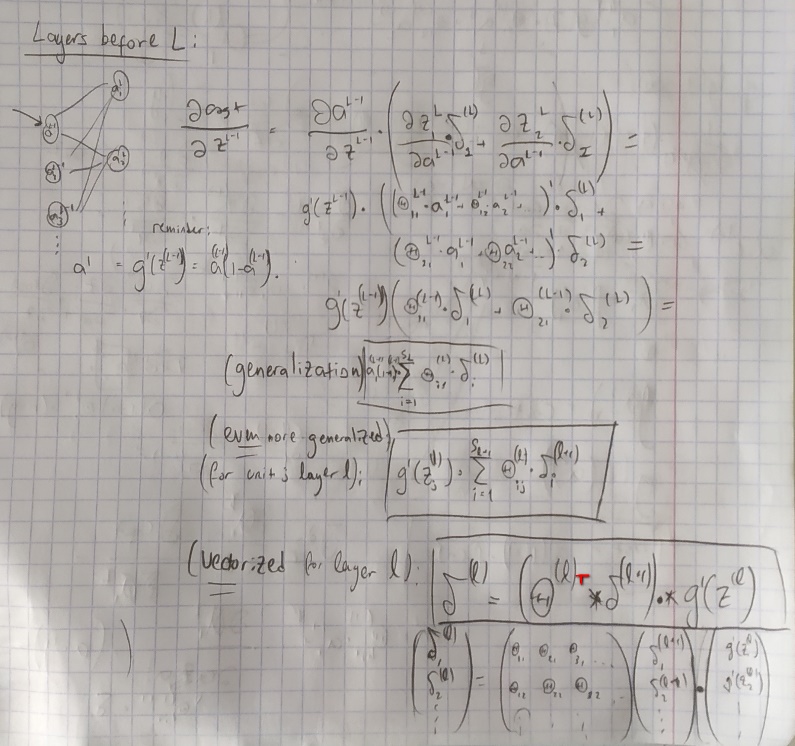
#### Stage 1: the Z’s, or: δi(l)

δi(l) is the (partial) DERIVATIVE of i’th unit’s Z element in layer l. We’re treating Zj(l), i.e. al-1\*Θjl-1 as **one** parameter in our calculations. Since Z and a are just a sigmoid func apart, we’re basically asking: what’s the best way to update this **unit** in order to make my output better? 

specifically for the output units, we get (taking all at once as a vector, image shows proof for one unit):

**δ(L) = a(L)-y. proof (more for knowing I did it, less for readability 😊): **

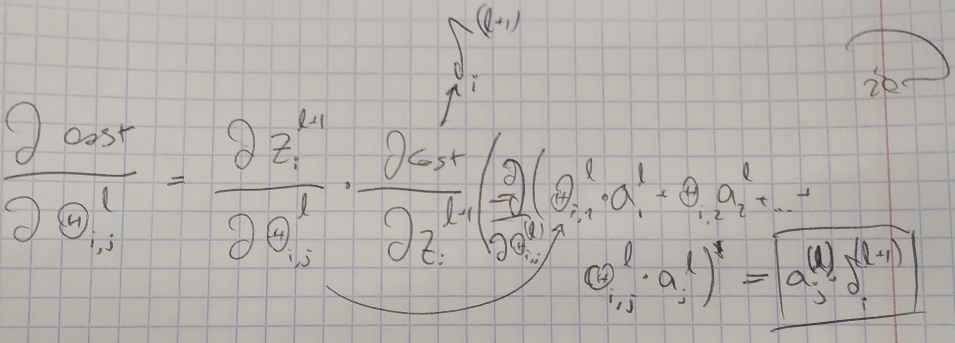
For every other layers, δj(l)= . Deriving that is pretty easy. Just chain derive from z(j) to cost func, by going from zi(j) to z(j+1) (all of the next layer) (chaining zj to aj, then aj to z(j+1)), and from z(j+1) to cost func is (image below looks complicated bc I generalized it a ton).



#### Stage 2: now the thetas Θ

Now we know what all the δs are, we know the derivatives of Zs. Since we in fact want to find out the derivatives of the **weights** (i.e. Θi,j(l)), in order to adjust the weights to minimize **J**. reminder: Θi,j(l) is the weight given to unit j in layer l for calculating unit i in layer l+1. Just **one** single connection. Unlike z which is a sum of connections.

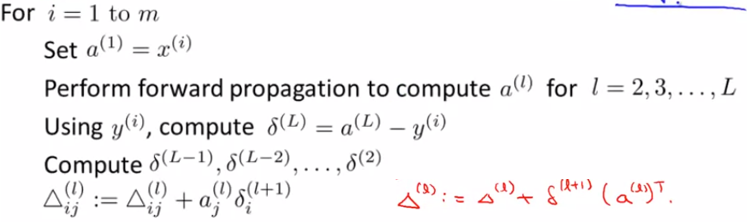
So, to calc Θi,j(l), we’ll just chain derive it to its next layer’s Z, multiply by that Z’s corresponding derivative (delta), and we’re done! Concretely:



(all factors without thetai,j(l) have a derivative of 0).

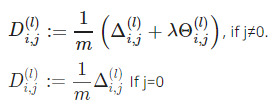
All that’s left is calc all that for **every** example, sum every corresponding theta from each example together into a matrix called Δ, divide by num of examples (and add regularization factor for each thetai,j(l) unless i=0), and we have **∇J(Θ)!!!!!**

Calculating Δ:



(red is vectorized version).

Then finally the mean derivative:

  done.

## Built-in optimization functions (fminunc, etc)

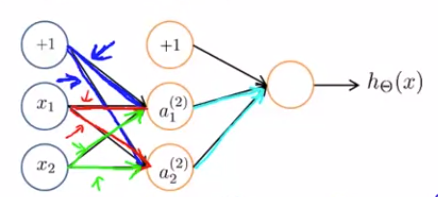
In optimization funcs, the thetas and derivatives must be vectors, so you need to unroll the theta matrixes into vectors and concat them together (with ;). but since YOU are who gives them the func and gradient func, you can just roll them back up (reshape) and use them as normal. It just uses vectors as a formality.

### Test to see gradients make sense

Do approx. by feeding J (that received an unrolled thetaVec) the same thetas but once instead of theta(i) gives theta(i+epsilon), and once with theta(i-epsilon), then divide by 2epsilon (approx. The slope). Do that in a loop, each time put approx. in vector, then compare to **DVec**, the derivative computed by BP (unrolled). **DON’T FORGET TO TURN OFF THIS COMPARISON ON ACTUAL LEARNING STAGE!!!!!!**

### Rand. init. of thetas

fminunc and the like require an initial theta vector. We set it to 0 in LogReg bc every paramter was unrelated to each other. Conversely, in NN we can’t set everything to 0, bc then we’d get that every inner unit has the same values, see image):



Since all weights are the same, x1 will output from both lines the same value, same for bias and x2. Meaning, a1(2)=a2(2), and we’re just doing tons of computations for a layer that essentially has 1 unit. So, use rand and epsilon to init to some small value around 0.

rand(10,11) \* (2 \* INIT\_EPSILON) - INIT\_EPSILON;

## All Together – steps to build Neural Network

1. Randomly initialize the weights
2. Implement forward propagation to get hΘ(x(i)) for any x^{(i)}x(i)
3. Implement the cost function
4. Implement backpropagation to compute partial derivatives
5. Use gradient checking to confirm that your backpropagation works. Then disable gradient checking.
6. Use gradient descent or a built-in optimization function to minimize the cost function with the weights in theta.

Or [see site, pretty good](https://www.coursera.org/learn/machine-learning/supplement/Uskwd/putting-it-together).

Remember that after finding the gradient (the whole procedure), that’s all just **ONE** iteration, that we checked vs the numerical derivative (the approx.), and if we were to do gradient descent, we’d do:

ThetaVec :=ThetaVec – alpha\*DVec

And do the whole thing all over again.

# Where to apply what

## Diagnostics

If we see the algo is very far off, there are a few routes to go down:

* Get more training examples – fixes high variance
* Try smaller sets of features – fixes high variance
* Try getting additional features – fixes high bias
* Try adding polynomial features – fixes high bias
* Increasing λ – fixes high variance
* decreasing λ – fixes high bias

Trying to go down one route by gut feeling is usually a waste of time. We’ll make some diagnostic tool to see more clearly what’s going wrong.

**Train set/test set** – partition dataset to let’s say 70/30, using 70% to train the algo, and test it on the remaining test set (partition should be random). Either use cost func to determine accuracy, or check misclassification rate (ratio of misclassification (on an absolute value)): 

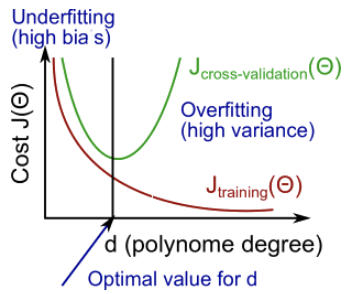
Then see average error rate. How much the system outputs wrong final (0 or 1) answers.

As we know, a low error rate in the training set doesn’t mean much, bc the hypothesis might be overfitted.

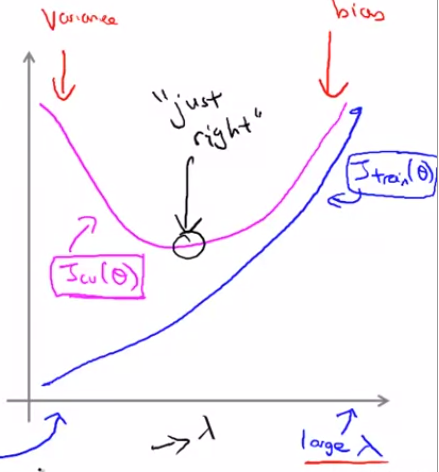
**How many polynomials?**

We can try using different models with different no. of poly degrees (d), train each model, then test on “cross validation” (CV) partition to see who had lowest cost. Now we have (allegedly) the best model. Notice we used the CV to train the d parameter (which represents how many degrees to use). So, we need to test if THAT choice worked (just like we need to test the training set’s optimal solution). So on the “best model”, we finally use the test set to see how good it is.

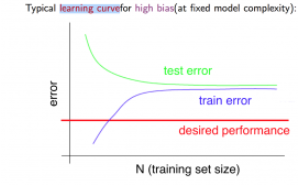
**Underfitting/overfitting as func of poly degree:**

****

**Underfitting/overfitting as func of lambda (regularization) size:**

****

**Underfitting/overfitting as func of training set size (w/ little to no regularization):**

****intuition – on high bias (underfitting), training set won’t be able to optimize even itself, so error will quickly converge to test error.

intuition: as training set grows, overfitting becomes “harder”, that is, it’s harder to be too precise with many examples, and will slowly converge with test error around desired performance.

**Diagnosing NN’s**

* A neural network with fewer parameters is prone to underfitting. It is also computationally cheaper.
* A large neural network with more parameters is prone to overfitting. It is also computationally expensive. In this case you can use regularization (increase λ) to address the overfitting.

**Reminder:**

* normalize = make everything between 0 and 1 by subtracting mu and dividing by sigma.
* Regularization = penalize parameters for being too big.
* If you try to find a lambda for a func, after finding optimal theta, when checking cost func, DON’T put in the lambda again, as we want to see that the lambda gave us as a final result for theta (it make theta more moderate, now we want to see what that theta is. Not regularize it again).

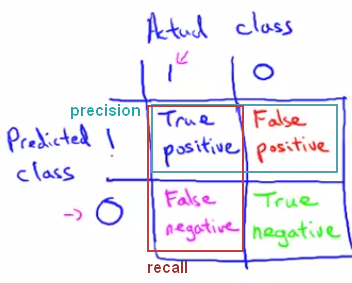
**Adding poly features in linear regression:**

Hypothesis func doesn’t know that the other params are polys of the original param. Just uses them as more ways to fit in with as much testing examples as possible. Though when we plot the data, we’ll show the graph with each poly acting as the poly it’s representing, since in fact that was what the hypothesis was doing, without knowing it.

In summary, DON’T JUMP INTO ONE SOLUTION, DO ANALYSIS FIRST.

**Skewed classes**

In most situations, seeing the accuracy of the h func is good enough, but what if one of the classes is almost non-existent? What if a dataset of people with cancer only has 0.5% examples with cancer? In these cases, a problem arises that it would be sometimes better to just always assume the answer would be benign and you’d get a high accuracy. Here comes in:

* Precision: tp/(tp+fp) – out of what we predicted was positive, how correct were we?
* Recall: tp/(tp+fn) out of what actually is positive, how much (in percentage) did we correctly predict?

Even if class is skewed, it can’t cheat and get high recall and precision.

Note: y=1 is the more rare class.

**Single real number evaluation metric –** one number that tells you how good you’re doing (instead of having to look at tons of examples, etc.). for example: accuracy (sometimes). Problem: if we use precision and recall, we now can’t as easily evaluate which algo is better, bc we have to look at 2 parameters.

* Solution 1: average of 2 params. Not good, bc one might be really high (high threshold) and the other really low, which is crap
* **F1 score**: . Usually just called F-score, it also takes in account if one of the scores is really low (if one param is 0, F-score will be 0, while avg will be >0).
* Other possible formulas, F-score is usually used.

Having a higher threshold for predicting y=1 (i.e. higher than h(x)>0.5, say h(x)>0.8), will result in:

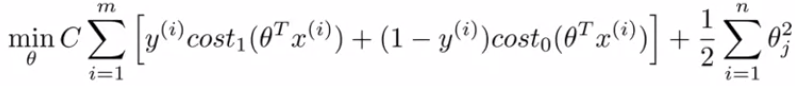
* Higher precision (we are more sure of our answers)
* Lower recall (we miss opertunities)

And lower threshold will do opposite.

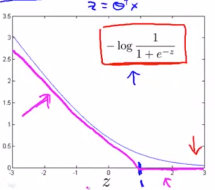
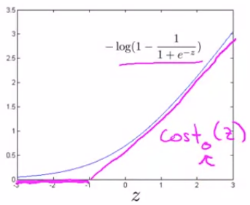
**When is lots of data good? Requirements:**

1. You have enough features (way to test: if a human expert in that field could predict result based on those features)
2. You can get lots of data
3. You have a complex, powerful function that can fit complex functions (like a NN with many hidden layers).

# Support Vector Machines (SVM)



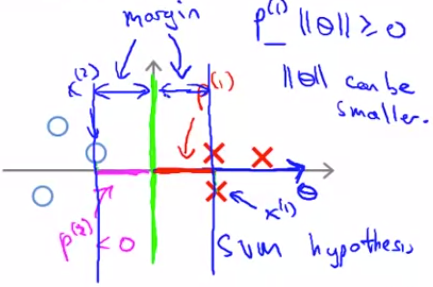
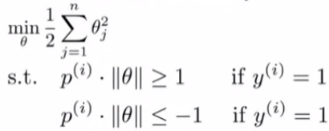
Mostly like logistic regression, only:

* Inner cost1 and cost0 are like similar to sigmoid, only they look like this: (
* instead of lambda we have C (which is kind of like 1/lambda). We adjust the hypothesis, not the regularization term.
* No 1/m averaging factor (doesn’t make a difference to the minimization).
* After finding θ, we just output:
  + H(x) = 1 if θTX>=0
  + H(x) = 0 otherwise.

SVMs are sometimes called “large margin classifiers”, bc they draw a line with a margin usually, since (as seen in images to the right) in order to have the cost be 0, theta’\*X must be more than just a little right (has to be at least 1 or -1, depending on y=1 or y=0).

**Reminder:** vector dot multiplication: u●v = ||u|| \* ||v|| \* cos(θ). Or – (signed) length of v’s projection on u times length of u. see [3b1b video](https://www.youtube.com/watch?v=LyGKycYT2v0) for further intuition.

## Why is there a margin in SVM?

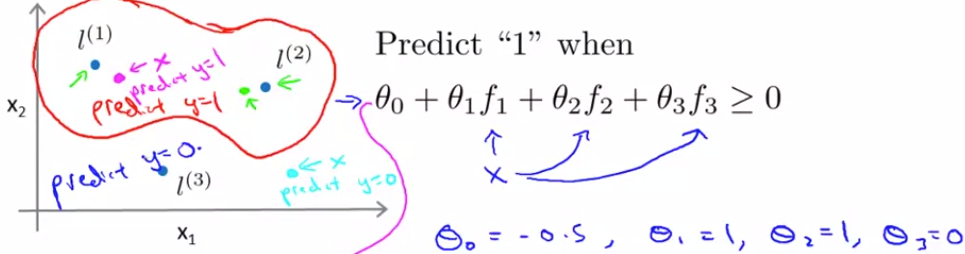
****Since the minimization func is basically ->  
we find that the theta vector will get as close as possible to the other example vectors, in order to be able to be small (bc the regularization factor demands that) and still have p(i) (the projection of the examples on it) be large enough to surpass 1/-1. Since the theta vector is like a 1d axis that the other vectors are projected onto (either on the negative or positive side, the origin being theta0), the boundary will be at the origin, with a margin on either side. The reason: **examples close to the origin will have a small projection,** and so it will be harder for an example to pass the ‘1’ mark, unless theta is very large, which it tries not to be (bc of regularization). Meaning, it’ll usually end up being wrong twice (both for y=0 and y=1), and cost for both factors of the cost func (see 2 graphs above) or make theta large in order to cover for it.

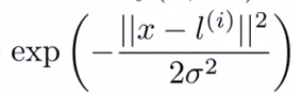
And so, where there are examples close to the origin (or rather the decision boundary line) probably won’t be the minimal theta. Thanks to the regularization factor, once the theta vector is in its optimal direction, it will be as small as possible to still let everyone pass the ‘1’ mark.

**Note** that only the examples that are in the margin will affect the decision boundary. Everything that’s already ok (i.e. |theta\*x|>=1), have a cost of 0, and have no effect (see graphs on last page).

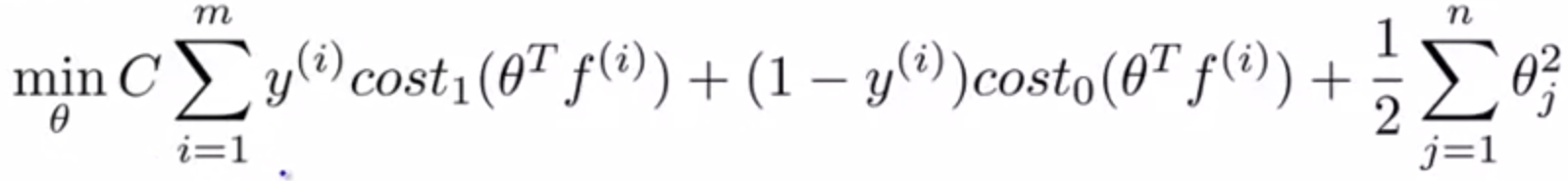
## Kernals – Landmarks

we can define landmarks (spots on the (hyper)plane) that are key points for showing if that area is 1 or 0. Then, by using some similarity func fi (**the kernel func)**for every landmark that outputs 1 the more an example is close (or **similar**) to its landmark li, and using those funcs as features for our SVM, we can create non-linear funcs for SVMs.



Example for kernel (Gaussian kernel func):

How to find landmarks:

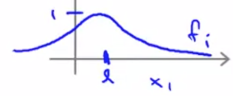
1. Define all examples as landmarks, and create vector f comprising of the m kernel funcs (sim(x,l(i)), one for each landmark (+f0=0)
2. For every example (x(i),y(i)), compute f(x(i)) for every kernel func, res goes to feature vector f(i).
3. Find minimal theta: use minimization func, only now x(i) becomes f(i). or rather

Notice that here n=m, since every feature corresponds to an example made into a landmark. In addition, theta2 is usually written as θTMθ. M is a matrix based on the kernel you use, for efficiency in large scale datasets (need to understand later).

1. Hypothesis func: y=1 if thetaTf>=0.

**Why not use kernels for logistic regression as well?** Turns out it only works well with SVMs bc of how it is built… why?

**Bias and variance with params C and sigma:**

* Bigger C = more variance, less bias, prone to “overfitting”. And vice-versa.
* Bigger sigma2 (from kernel func) = more easily get 0, i.e. more bias, less variance (easier to get true, smoother func: And vice-versa.

## Using an SVM

don’t code it yourself, but must choose:

* C
* Kernel
* Sigma2 (if you choose a non-linear kernel).

And need to do:

* Might need to implement kernel func if isn’t already premade.
* Feature scaling, so to avoid having the SVM focus on features that tend to have high numbers.

Other kernels: poly kernel (XTl + const)degree, string kernel, chi-square, histogram intersection. Almost never used. Must satisfy “mercer’s theorm”.

Guide to SVM w/ gaussian kernel vs logistic regression/ SVM linear kernel:

* n is large (relative to m): logreg/svmlin.
* n is small, m is medium (n=1-1000,m=10-10,000): SVM w/ gaussian kernel.
* n is small, m is large(50k+): add features, then logreg/SVMlin (svm gaussian struggles a bit with huge datasets)

SVM is convex, so global min!

can also use NN for everything, might be a bit slower though.

# Unsupervised Learning

You get lots of points x(i), and try to find some structure. For example: clustering algos, divide points into groups.

## Clustering algo #1: K-means algo

Idea: we start with k centroid points (centers of clusters), find which examples are closest to each centroid (minimizing distance with respect to (wrt) C), and then move centroid to avg of its examples, (minimizing distance wrt mu). By the end, should have minimum cost and k groups.

|  |  |
| --- | --- |
| Steps: | Cost func: |

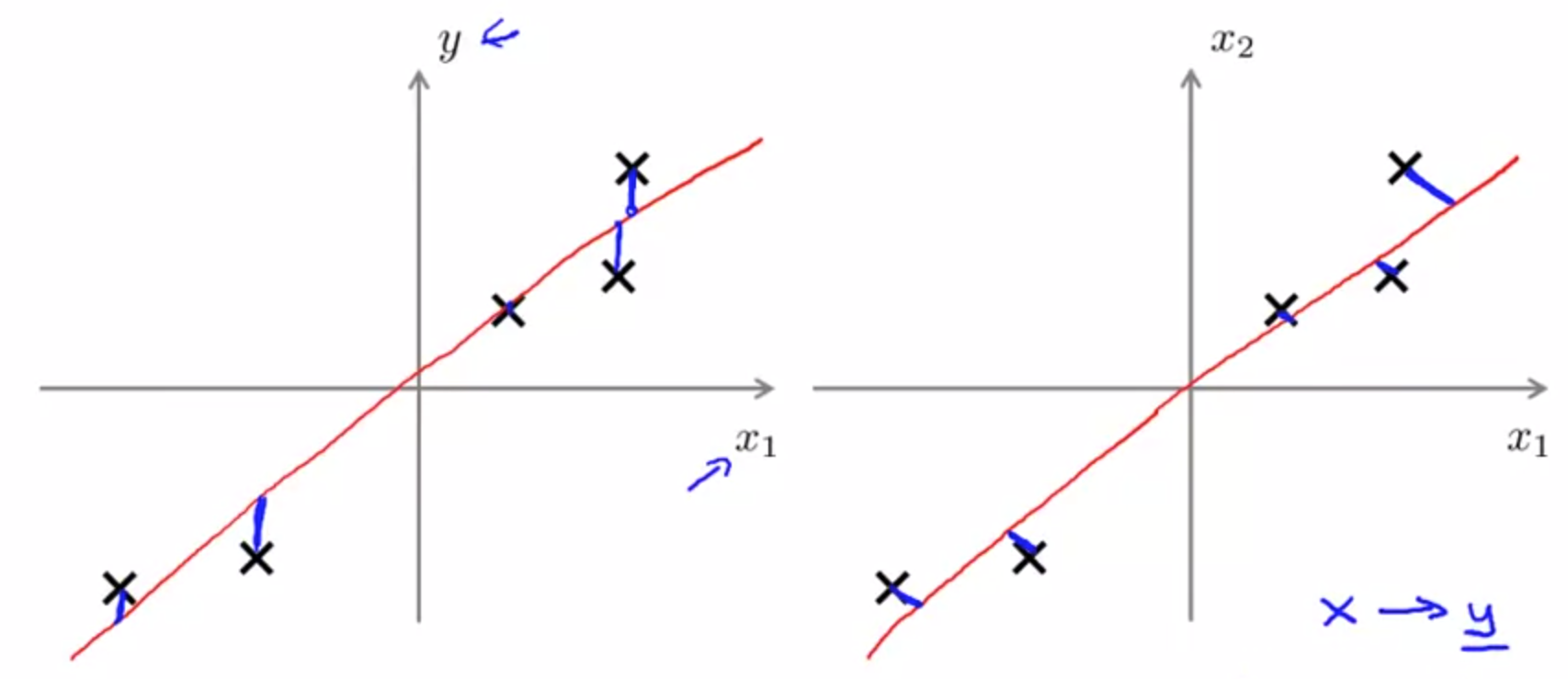
Init cluster centroid at rand example.

* If k<=10, do multiple rand inits, to reduce chances of getting stuck at local optima.
* If k>10, don’t bother. Will be more or less the same.

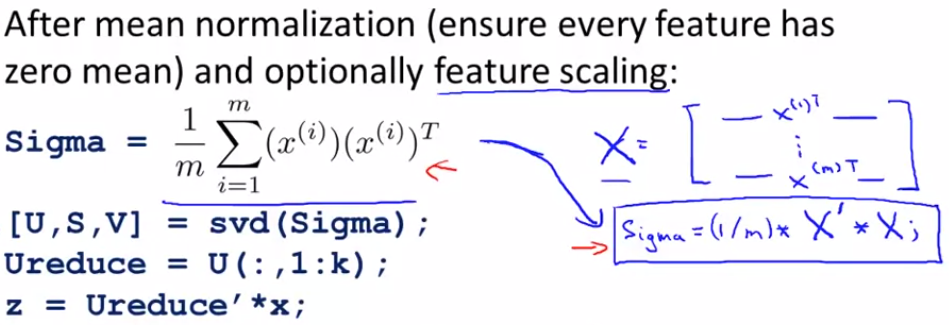
Ways to chose no. of clusters:

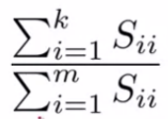
* Elbow method (usu doesn’t work, not always so clear cut).
* Manually, by asking what you want.

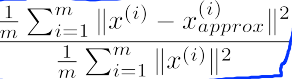
## Dimensionality reduction

Reduce dimension of features to reduce size and time that the learning algo takes to optimize hypothesis func. Want the **distance** of examples from projection surface (defined by vectors we want to find) to be minimal, to minimize data loss. In image, left is linreg, where error is vertical miss, bc we want to predict y. in Principal Component Analysis (PCA) (on right), we want **distance** of examples from line, there’s no y here AT ALL.

Algo for finding minimal vectors:



* Svd is like eigenvalue (=Singular Value Decomposition).
* U is the matrix built out of vectors that makes up the minimal distance surface in N dimensions. We will use only k vectors out of the n that U returns, calling it UReduce.
* To go back to N dimensions, use mxn Xapprox =mxkZ\*kxnUreduceT.

To compare different K’s, use - the S from [U,S,V], which calcs percentage of variance retained. (instead of using ->   
which btw shows **1-X%** of variance retained).

Then choose k that retains X% of variance (or of original data for intuition).

Actual steps for reducing dimension for speedier training of hypothesis:

1. Use training set (without y value, just the features) to obtain U
2. using S find optimal K.
3. take K vectors from U, creating UReduce.
4. Map every x(i) -> z(i) via X\*UReduce (new size: MxK).

Now we have the reduced examples, train hypothesis func with new Z’s as if they were X.

For new example x:

1. Map x->z
2. Predict z using h(z).

Other uses for PCA:

* Visualization (reduce to k=2,3)
* Prevent overfitting – DON’T DO THIS, DOESN’T WORK WELL, NOT MEANT FOR THIS. Will lose data, and doesn’t take into account result Y. instead just use regularization.
* Also don’t jump straight into PCA, see if the raw data works first.

# Anomaly detection

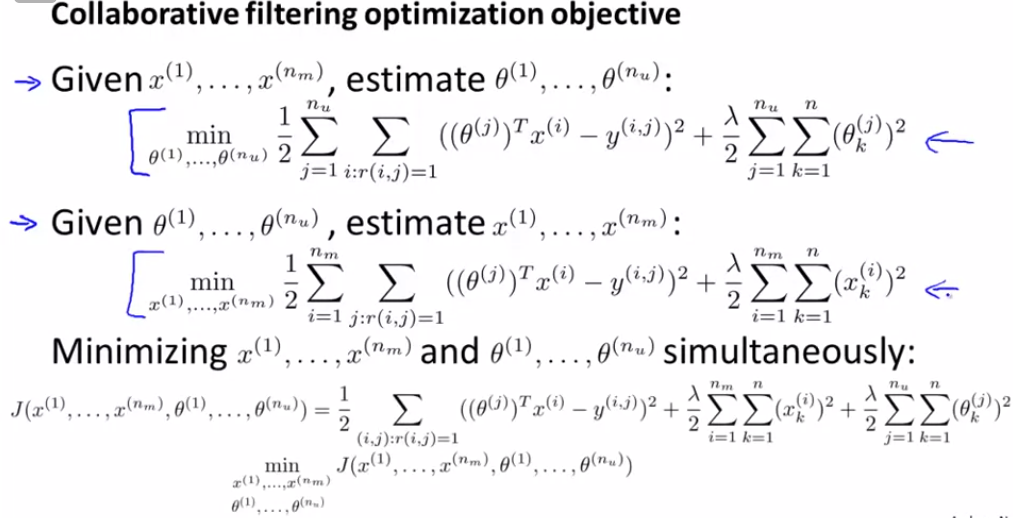
Background: like in azure security for example, we want to see if a new example is outside of the cluster it should be (cpu using too much percentage, etc.), and then detect it as an anomaly and report it to some authority. Use it when you don’t have many positive examples, and many different things can go wrong. Idea is to model how a normal situation should look like, and report any anomalous examples.

Make normal distribution model from training, then test on CV (for tuning epsilon etc) and test set.

* Use log(X+c), ^,etc if histogram (hist func) of features don’t look gaussian.
* Try to make/add features that give anomalous values for anomaly examples. Maybe a relationship between two features (cpuload/data traffic)

# Recommender systems

Based on user rating, recommend new items to user (movies, products, etc).



Part 1: given degree of each movie’s “genre” (i.e. how much action is it 0-1,romance, etc), and a list of ratings given by users (r(i,j)=1 are movies i that user j actually rated, with score y(i,j)), minimize vector θ(j) to be as precise as possible (theta j is vector specific to user j).

Part 2: given **users’** degree of enjoyment from “genres” (given by θ(j) for each user j) and their ratings for subset of movies, predict x(i) for each movie i.

Part 3: just roll with it 😊. Since we don’t actually have either vectors θ or X, start them off at small rand values, (also guess no. of features for each example), and slowly minimize cost wrt the users’ actual ratings.

**Note:** If some user didn’t give any ratings yet, we want to give them a default rating of the avg rating of that movie given by others (so as to not have everything 0, then we can’t recommend anything to him).

Finally, for user j, on movie i predict (after computing mean of each movie and subtracting that from scores): θ(j)\*X(i) + µi.

All in one vector implementation for prediction: X\*ΘT + µ (with X and Θ being matrix of transposes of every x(i) and θ(j), respectively).

**Extra bonus:** to find some new movie j that is related to i, min(|x(i)-x(j)|).

**Final note:** features of X aren’t actually genres, that was just for intuition. In reality it will be some quality that the learning algo found.

# Large Datasets

If we see a high variance, if we get lots of data we’ll get a better result (first check to see that it’s needed). But, in order to not have to do massive computations for every gradient descent step, we’ll use:

## Stochastic Gradient Descent (and variants)

i.e., instead of mean of derivatives of every example, just step according to derivative of first example, then second, third, etc. So while regular G.D. steps after going over all examples, SGD steps after 1 example. Will wiggle around global minimum and zigzags around contours but is much faster. Shuffle examples first though.

### Mini-Batch Gradient Descent

Same idea, just use b examples per step instead of 1. Takes best of both worlds. More likely to converge faster, and faster than regular batch (i.e. step after all m examples).

### Checking for convergence

Instead of plotting cost of all examples as func of no. of iterations, before computing gradient of example i, save its cost, then after 1k examples plot mean of their costs. The more examples per point the smoother will be. If plot is diverging, lower alpha.

### Online Learning

Learning algo that continually gets more data (e.g. new costumers always feeding more data). Solution: for every new examples (someone decided to use/not use your sevices, i.e. y=1/0 + product he wanted + price and other params you thought would be appropriate for user, i.e. x), update theta using (x,y), that specific example.

What’s good is that it automatically changes based on user preferences, and don’t need to save data. You just use it to update params, then throw it away.

## Map Reduce and Data Parallelism

e.g. in mini-batch GD, just split b into different processers/cores/computers, each processor computes part of the examples, then add them up and div by b (i.e. total no. of examples computed for that step).

**Note:** requires it possible to express the learning stage as a sum. (most algos can be expressed this way, e.g. J func of linreg, logreg, even NN (split m examples to x processors, each one doing FP and BP on its subset of examples, the compute mean of sum of result derivatives).

# Photo OCR example

Segment project into pipeline (not all of which has to be using ML, in this case it is):

1. recognize text (via using sliding window with possibly varying window size and step-size/stride). Even if window is bigger, will scale down to algo’s learnt size.
2. segment text into chars (by supervised learning).
3. Recognize chars

Possible way to get more data: make synthesized data:

1. in text, use different system fonts instead of taking pictures.
2. Distort given examples: warp images, add noise to sounds (meaningful noise, like cars in bg, bad phone connection, etc. not just pure artificial noise).

**Tip:** sometime, it’s pretty easy to get more data (say, 10x) (assuming that’ll help you), you just have to notice that fact and put in the work (especially when original dataset is small).

## Ceiling Analysis

In a pipeline, try to see what’s most worth focusing on by manually making one module of the pipeline give 100% accuracy, then see the improvement. If it’s negligible, don’t focus on that module. Focus on the main bottlenecks of the pipeline.

**Don’t trust your gut feeling. Structure stuff much more.**