Feature selection:

bias/variance

regression Lasso -penalizes extra features -just assigns 0 coefficient to features

SelectPercentile and SelectKBest will give us the features plus TfIdf Vectorizer

tools/email\_reprocess.py does feature seleciton......

Parmeter Tuning: GridCV -- finds params for classifiers almost automatically

Principal Components Analysis:

- find new coordinate system - only by move and rotation

- brings down dimensionality of features by creating composite features

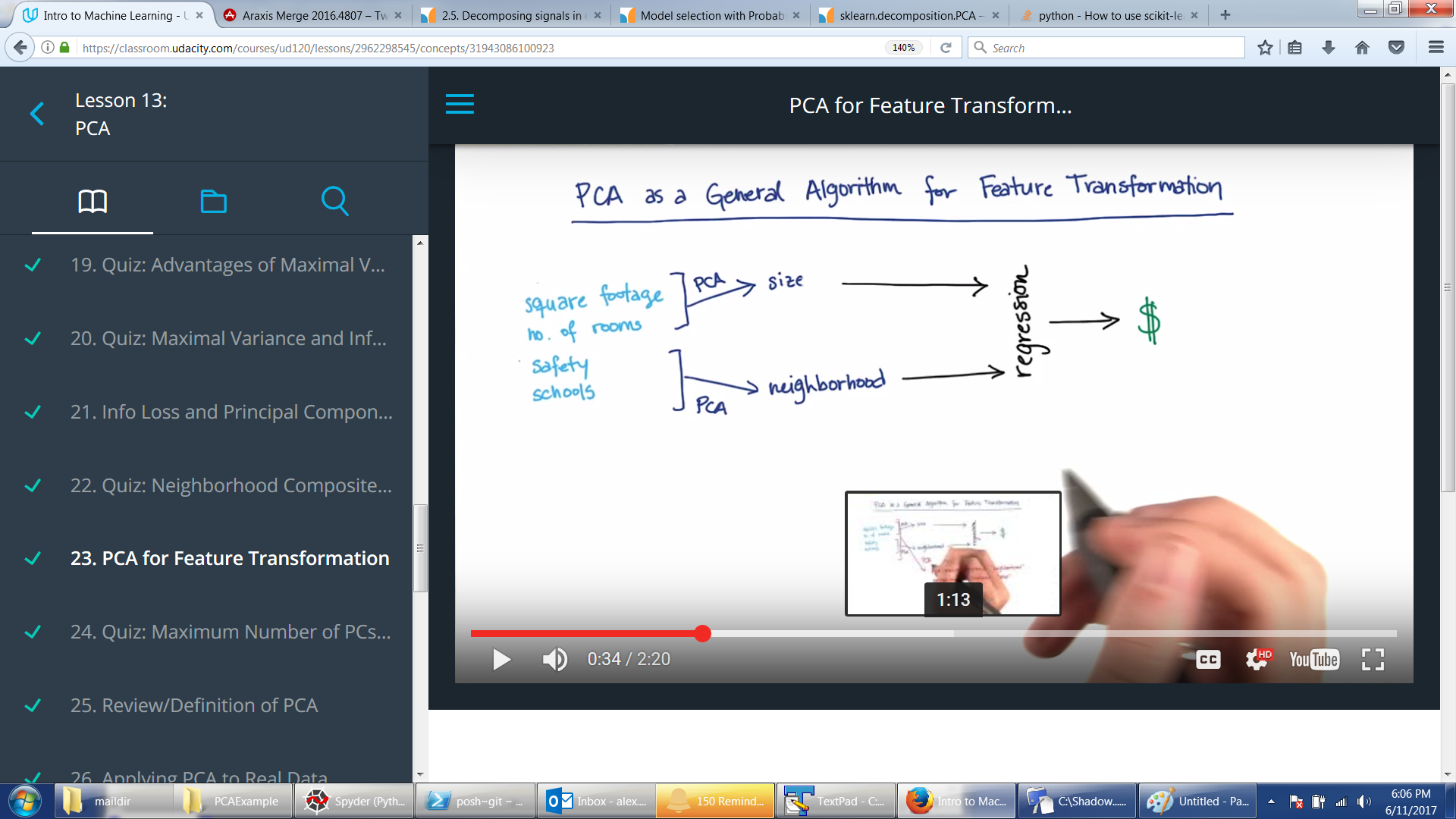
- exrmely useful for UNSUPERVISED learning

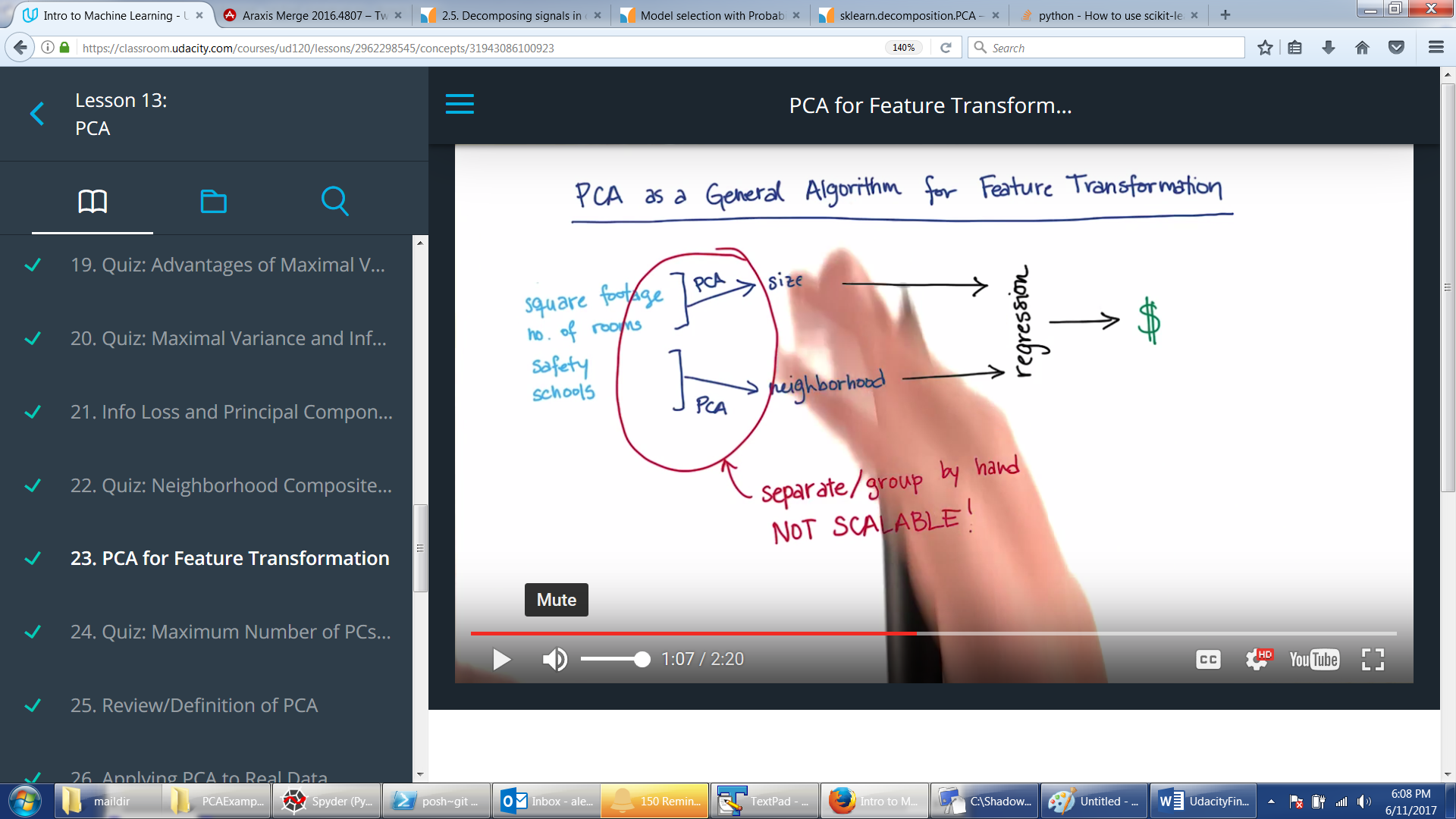
- principal component - direction of maximum variance (mathematical variance)

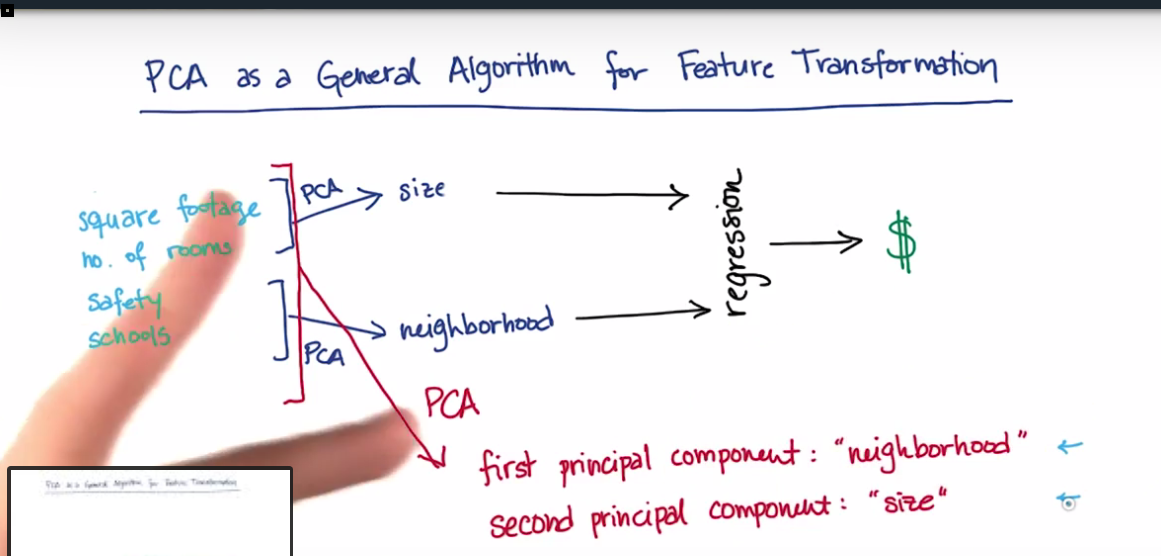
- the reason it is like that is becuase it minimizes information loss -

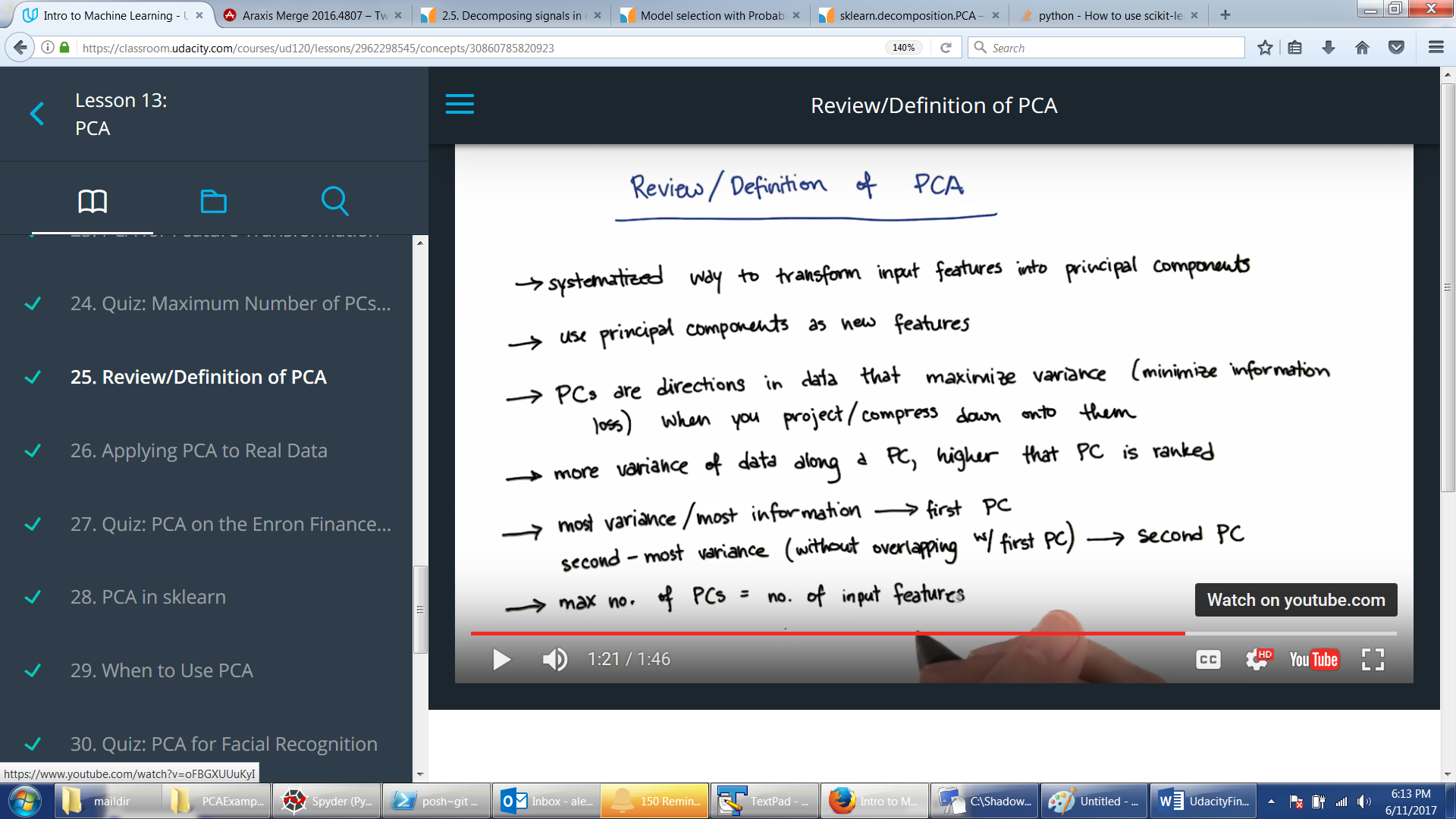
Remember: continuos things -- regression

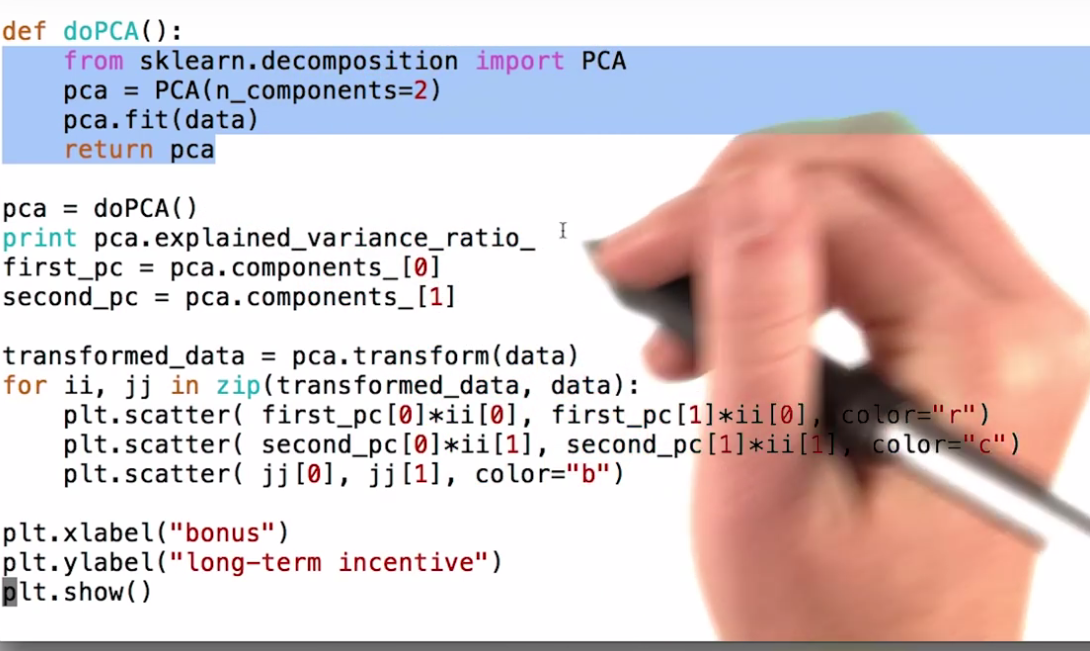
non-continuos – depends

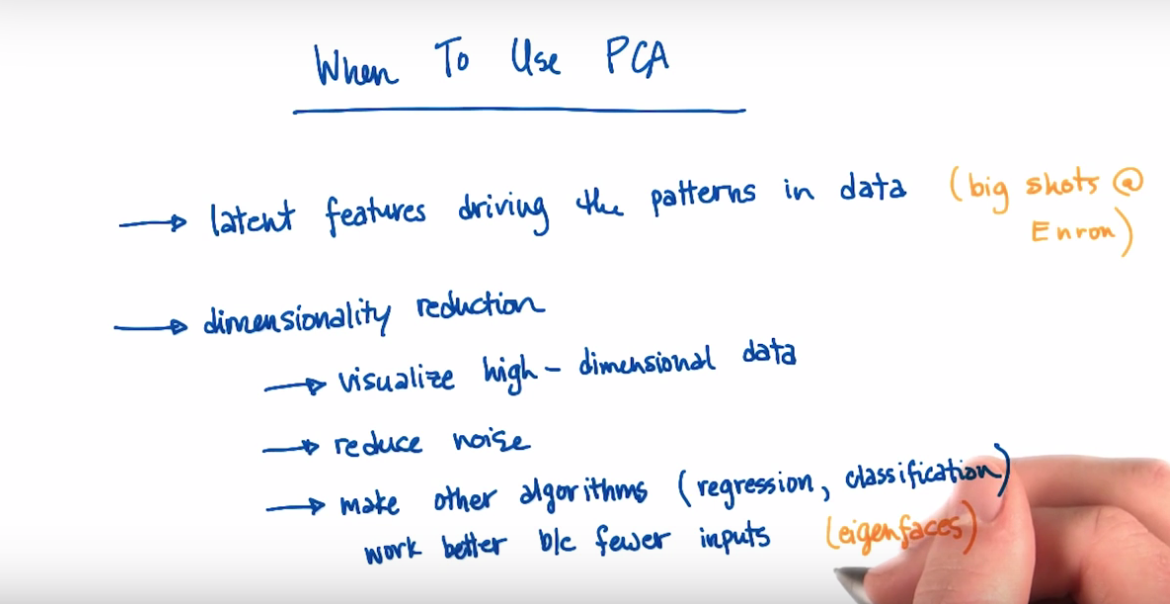




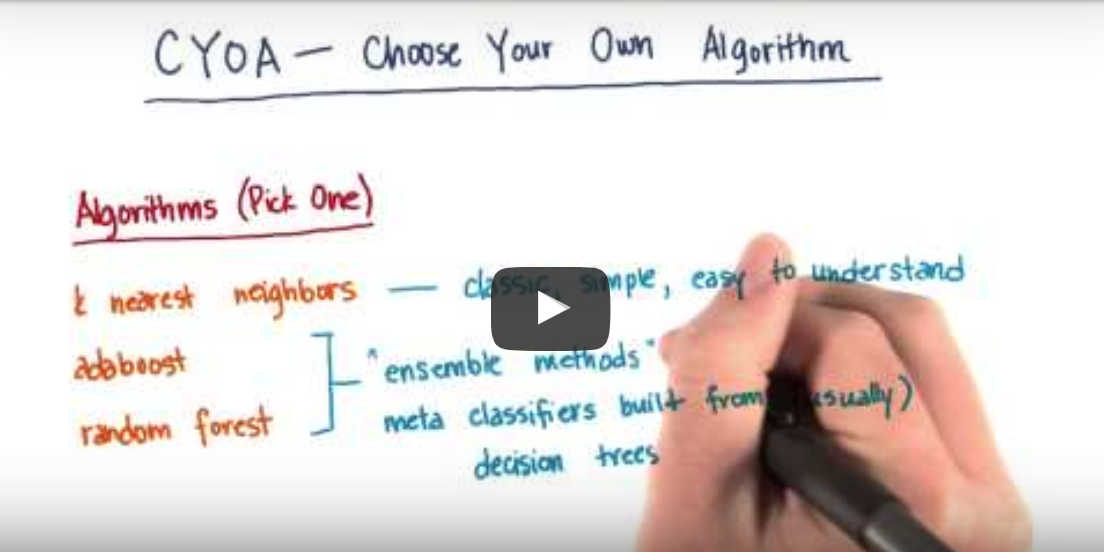








ALgo most likely useful:



Deep Learning Nanodegree Foundation

Neural Networks.

Important: each node is NOT a function like OR/AND/NAND. Each node is the same function applied to inputs and weight. See my code for Boolean perceptron. The only thing training does is finds values of weights such that

Wx + b is either accepted or rejected.

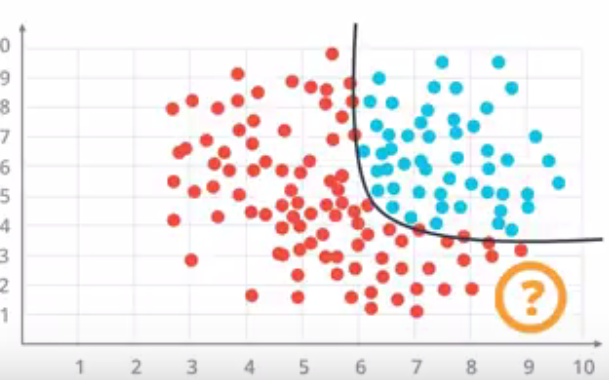
If singe step function --- is Wx + b >=0 accept else reject.

See my code for linear perceptron.

Linear is not cool because it does not allow for gradual training. You move weights a little but still get 1 – so you don’t know if it was a good move or bad. SO there is continuous function usd 1/1 + e\*\*-x. This brings us to the world of probabilities. Probability is at 50 when Wb+x = 0.

Important training trick: if a point was misclassified the weight are changed slightly to bring the ‘dividing line’ closer to the point. It is done by adding or subtracting xi\*learn\_rate from wi (se LinearPercetpron.py)

What to do if data is not divisible by line? It is a non-linear regression:



Here instead if easy trick with bringing line closer to point as with linear perceptron we use Log Erro function to evaluate how far we are form the point. This function should be continuous and differentiable. This is where the magic is.

Conceptually we are just trying to minimize ERROR when we train where conceptually ERROR is the SUM of distances of all misclassified points form the “dividing line” though it may not be a line at all.

The sigmoid 1/(1+e-x) is not such ERROR function its jus continuous substitute for step\_function. At this point Udacity still did not tell me what ERROR calculating function is

**Non-binary output.**

Non-binary sigmoid is something that produces not Yes/NO but probability of A is u, probability of B is v, probability of C is w. Given the KNOWN linear model with KNOWN weights. This is something I do not understand. But it is called Softmax (see my code for softmax). The probabilities should sum up to 1. SO they use normalized exponent. Basically here it is: suppose you KNOW a set of outputs (list of numbers> 2). Then run softmax(List) to get probabilities – make sure List is sorted in descending order so list of probabilities is also in descending order.. Then given linear model will produce 1 number and then you run softmax() on this one number and see in what segment it fits – then you predict who it was A, B, or C depending on where that number got in.

**Non-numeric input**

Create input variables. For example if inputs are duck, beaver, walrus and we pass duck then use it as 1,0,0. Walrus will be 0,0,1. This is called ***one-hot encoding***

**Maximum Likelihood**

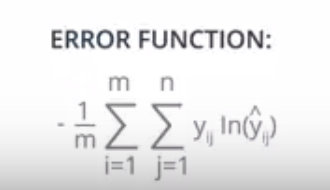
We are getting closer. Seems like the ERROR calc telling us which way to adjust is based on maximizing probability of correct classification….So Maximizing Probability is minimizing ERROR function

This is how it is done: suppose you have KNOWN outputs you are using for training. Take random W vector. And run through sigmoid for each output. For the ones the random model identified correctly leave the probability calculated by sigmoid. For the ones identified incorrectly assign 1-probabilityCalculatedBySigmoid. Now multiplication of all these probabilities is probability P that the MODEL is correct. We want to maximize this. But multiplying is BAD – we use log(P) --- but then we just want to minimize this. Minimize MAGNITUDE of log(P) where P is Product of all probabilities calculated for each point. This log(P) number we are trying to minimize is called Cross-Entropy

Look at crossEntropy.py to see how I coded it for 1/0 binary outputs vs nice vecrtorized thing they did. They did it as SUM(yi \* Pi + (1-yi)\*(1-Pi)) because depending on Yi either first or second term is 0 so easy to vectorize.

*Now, what if it is multi-class entropy instead of binary entropy*

In this case each point is just sum of

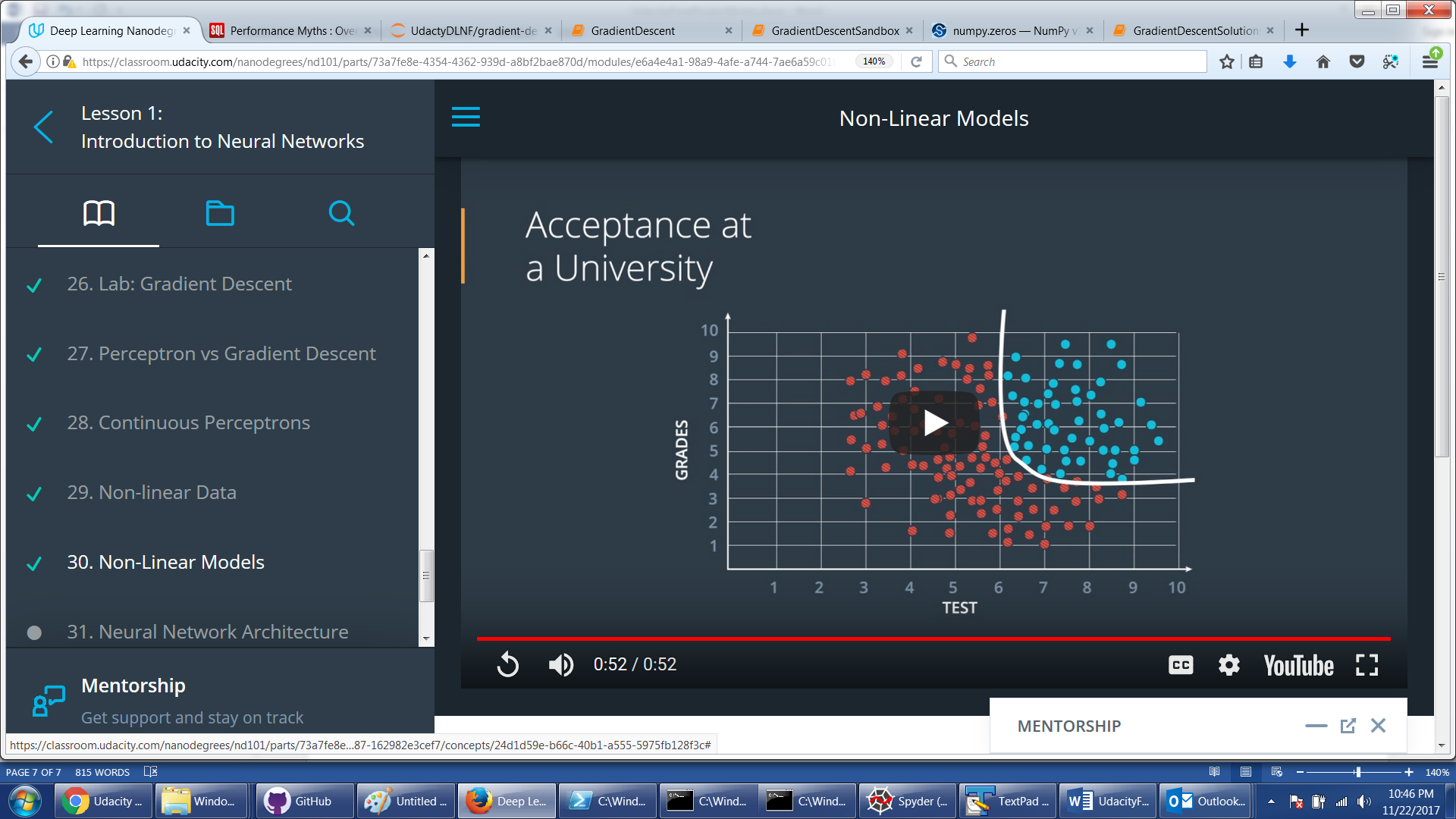


**GRADIENT DESCENT**

See my Jupyter Notebook page GradienDescent. Big point: GradientDescent adjust the separation even for correctly classified points (because in between 0 and 1). Binary perceptron does not adjust for correctly classified point – its just 1. So GradientDescent is better.

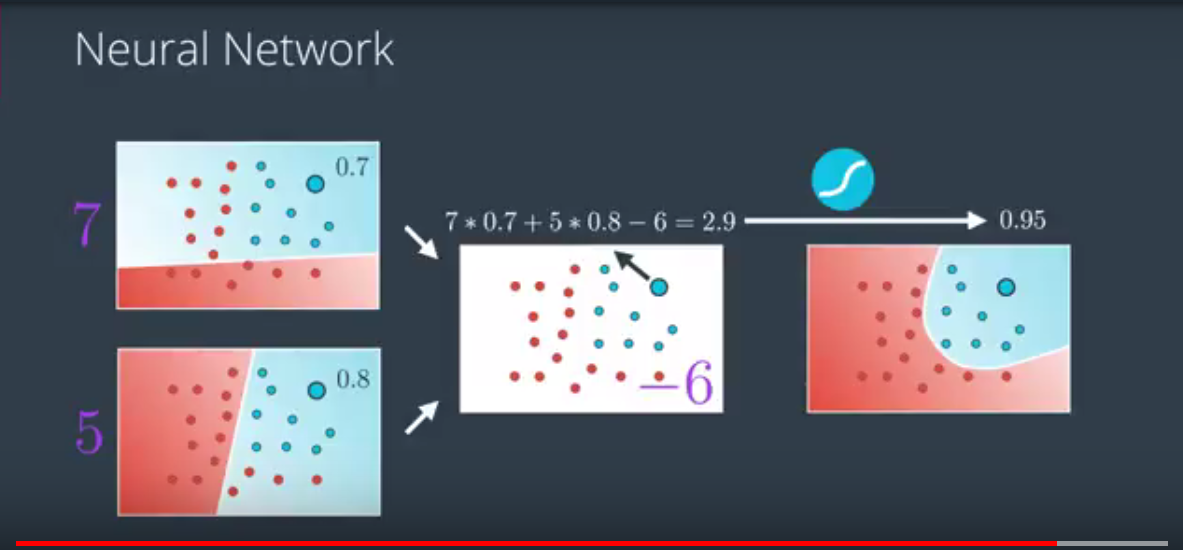
**But what if it is not a line separatable data**

For data like this we create probablilty function but plotting points that are equally likely to be on one side or the other:



So the way to get that curve is to “combine” two lines. See leftmost layer in below picture.

But the trick is that “combinig” means apply Sigmoid(Sigmoid(model1) \* weight1 + Sigmoid(model2)\* weight2). Picture below show is – and it becomes nothing more than the same perceptron with trainable weights. The only thing picture below does not explain is how we came up with 2 nodes in first hidden (hidden means non-input and non-utput) layer and one node in second layer. I am guessing 2 comes from the fact that we are looking at 2-dimensioanl space: red or blue. We will see…actually NOT. Section 31 video in DLNF Lesson 1 is a revelation



The actual data preparation for Gradient Descent Neural Network

* Need to normalize all inputs do that they have zero mean and a standard deviation of 1. This is because of nature of sigmoid function. It squashes really small and really large inputs. The gradient of really small and large inputs is zero, which means that the gradient descent step will go to zero too. See data\_prep.py
* Initial weight initialization requires interesting approach:  
  We want these to be small such that the input to the sigmoid is in the linear region near 0 and not squashed at the high and low ends. It's also important to initialize them randomly so that they all have different starting values and diverge, breaking symmetry. So, we'll initialize the weights from a normal distribution centered at 0. A good value for the scale is 1/√​n​​​ where n is the number of input units. This keeps the input to the sigmoid low for increasing numbers of input units.

weights = np.random.normal(scale=1/n\_features\*\*.5, size=n\_features)

* Also see my GDNeuralNework 1 – one layer.

**Backpropagation**

Important is to understand that we are trying to calculate delta\_w on each layer by propagating back the WEIGHTED error.

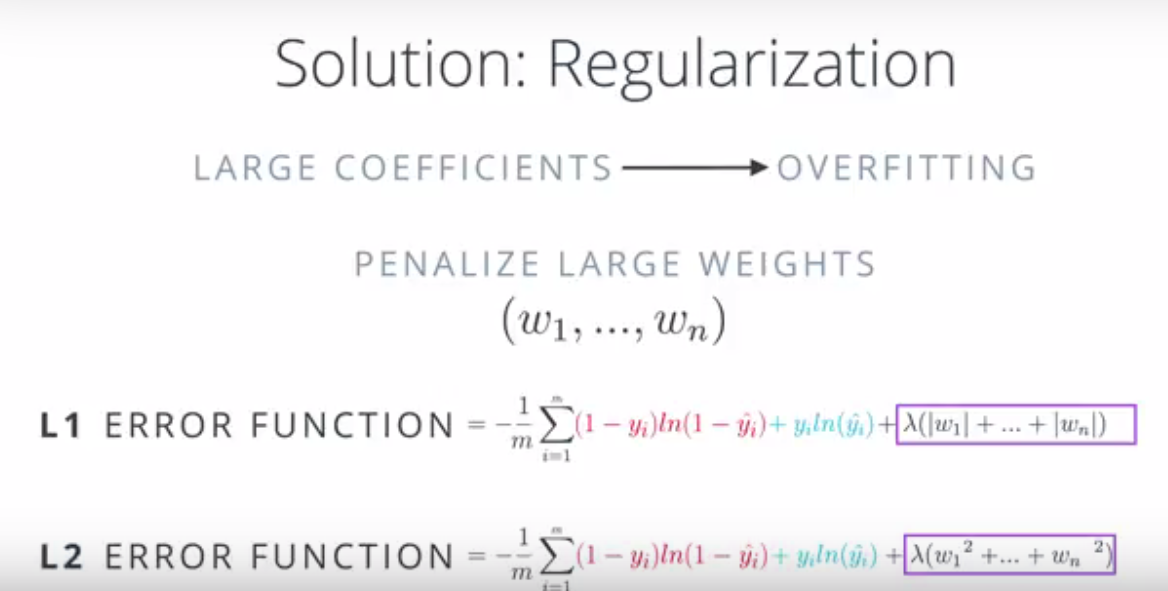
Lesson 2 backpropagation exercises and especial a description of 2 input -> one hidden -> One output network is priceless. Also my code for backpropStepForAdmissions is priceless with comments.

Testing - underfitting vs overfitting.

General idea is to overfit bu use techniques that roll back effects of overfitting. This is done by looking at accuracy of model on training set and on testing set and finding the Golden Spot where there is a good fit on training and testing together. Because the further in terms of number of epochs we go the worse we do on testing set because of overfitting. SO stop in the right epoch ☺



Another trick to prevent overfitting is Regularization. There are 2 kinds L1 and L2. Both just add a term to error function to make the large weights less “intrusive” – basically penalizing large weight. This is because large weights make the model to easily overfit.



L1 is good for feature selection because it moves some weights to 0 quickly thus getting rid of unimportant features. The L2 is good for training models.

Another cool method for training is turning OFF nodes with probability P form each EPOCH. Its called DROPOUT

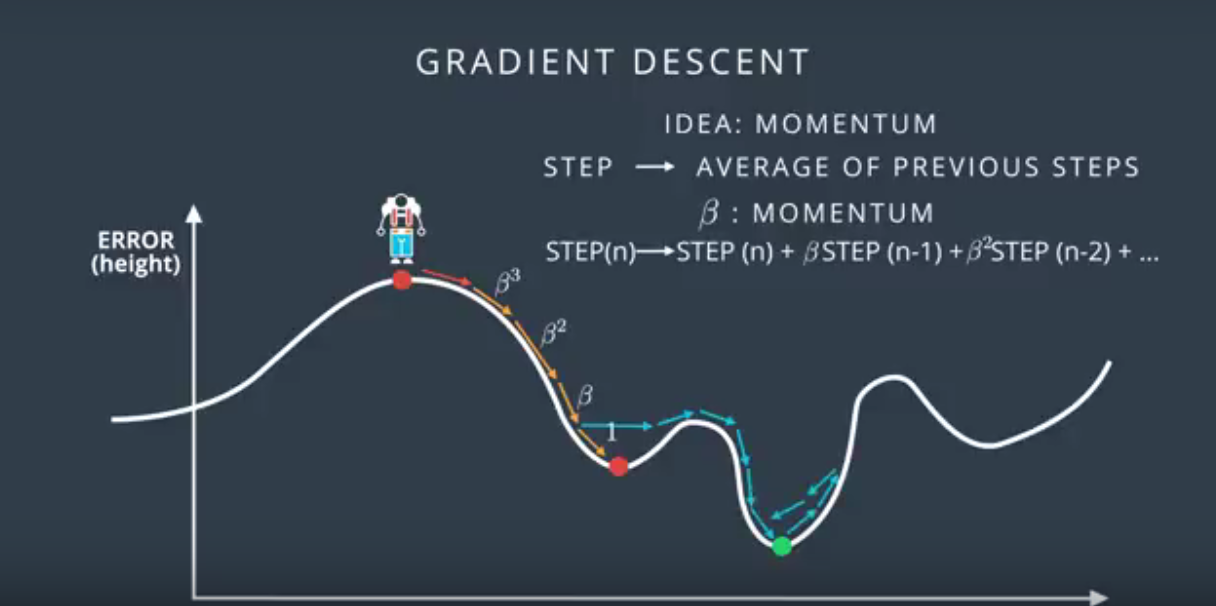
To solve the problem with Vanishing Gradient we can use tanh or ReLU. The problem with sigmoid is that derivatives go to 0 really quick and chain rules multiplies derivatives. SO tanh or even ReLU tha is linear solves this problem easy.

Stochastic Gradient Descent is doing the whole forward and back propagation N times with small batches of data of size Size/N rather than one time with whole data. This is because N small steps are computationally less intensive than 1 large step. MEMORY for Matrix Multiplication is an issue

Learning rate decay is a schema that allows to take large gradient descent steps when the error is alrge and smaller ones when it is small. This is very cool.

The way to deal with local minima is random restart with random initial weights!!!!! Just like your simulated annealing

Also concept of Momentum is somehtng that helps with local minima. Becaue the random restart is computationally expensive



# Evaluating How Good the Neural Network IS

1. Confusion Matrix: simply

[[TruePositive, FalseNegative],

[FalsePositive, TrueNegative]]

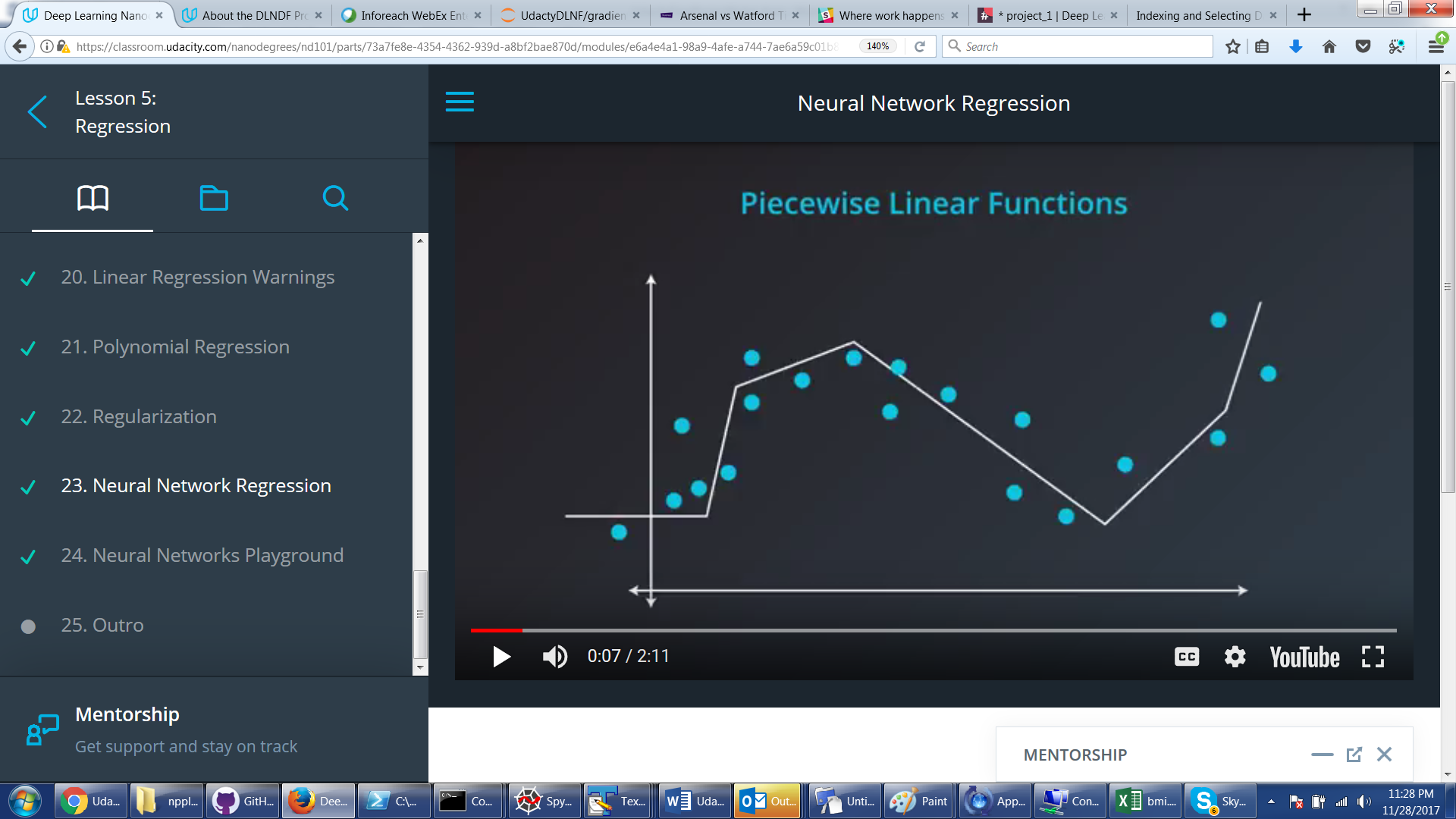
1. Accuracy: simply correct/total – not good when small number of things to identify then if you miss them all it still could be 99% accuracy
2. Precision = TruePositive / (TruePositive + FalsePositive) we want high precision if we want to Err on the side where we really do not want False Positives and much rather have more False Negative  
   Recall = TruePositive / (TruePositive + FalseNegative) we want highr ecall where we really do not want False Negatives and much rather have more False Positives

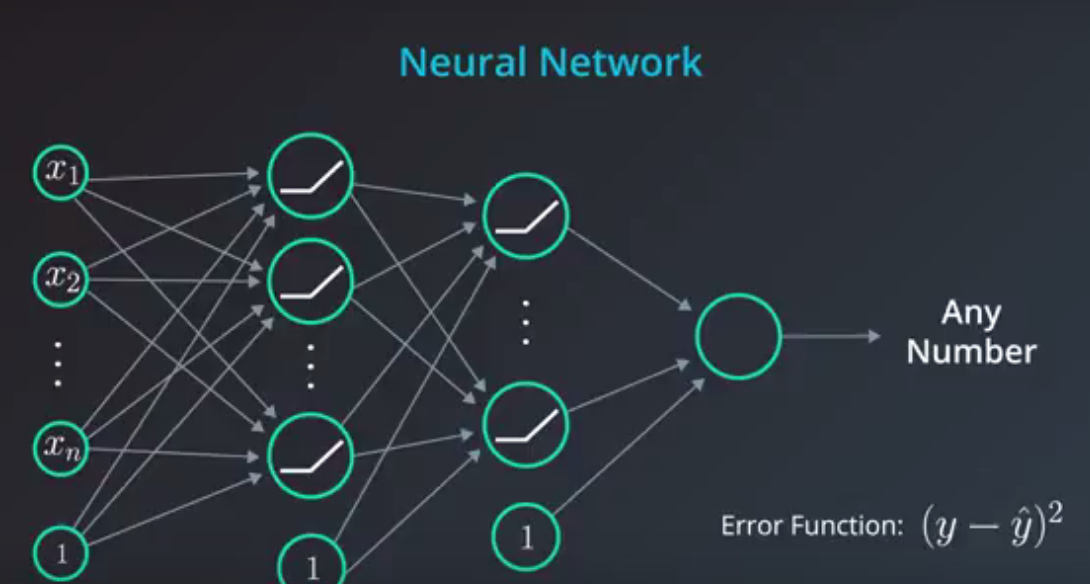
# Linear Regression

1. Build a line.
2. Minimize error by using gradient descent to minimize either absolute error 1/m(SUM|y – y\_h|) or by minimizing squares 1/2m(SUM((y – y\_h)\*\*2))
3. Squares are better because more tending to go to middle – with absolute the error does not change if liens just moves up or down.
4. Use sklearn to fit.
5. Multi-variable is still OK.
6. Polynomial is still OK but complex
7. Has CLosedForm solution where we could just compute weights minimizing the Error but it is very expensive.
8. Regularizaion!!! – is a trick that lets us chose simpler model instead of more complex. For example Linear instead of polynomial. This is done by penalizing –increasing error based on number of polynomial coefficients. So model pays price if it tries to be more complex. Tehre is L1 or L2 regularization – difference is adding abs(coefficient) or square(coefficient). Introducing coefficient LAMBDA les us control how much we penalize. For example small LAMDA means penalize complexity a little bit. And vice versa.



The coolest thing is that Neural Networks can be used to find weights for Linear Regression – each layer is a line segment form pic below. Only difference is that we ar removing sigmoid from the last node and replacing it with just the value and then trying to mimize linear regression error (absolute or square). And this just works with back propagation etc…!!!!!!!!!





Visual playground for using NN for Linear regression

http://jalammar.github.io/visual-interactive-guide-basics-neural-networks/