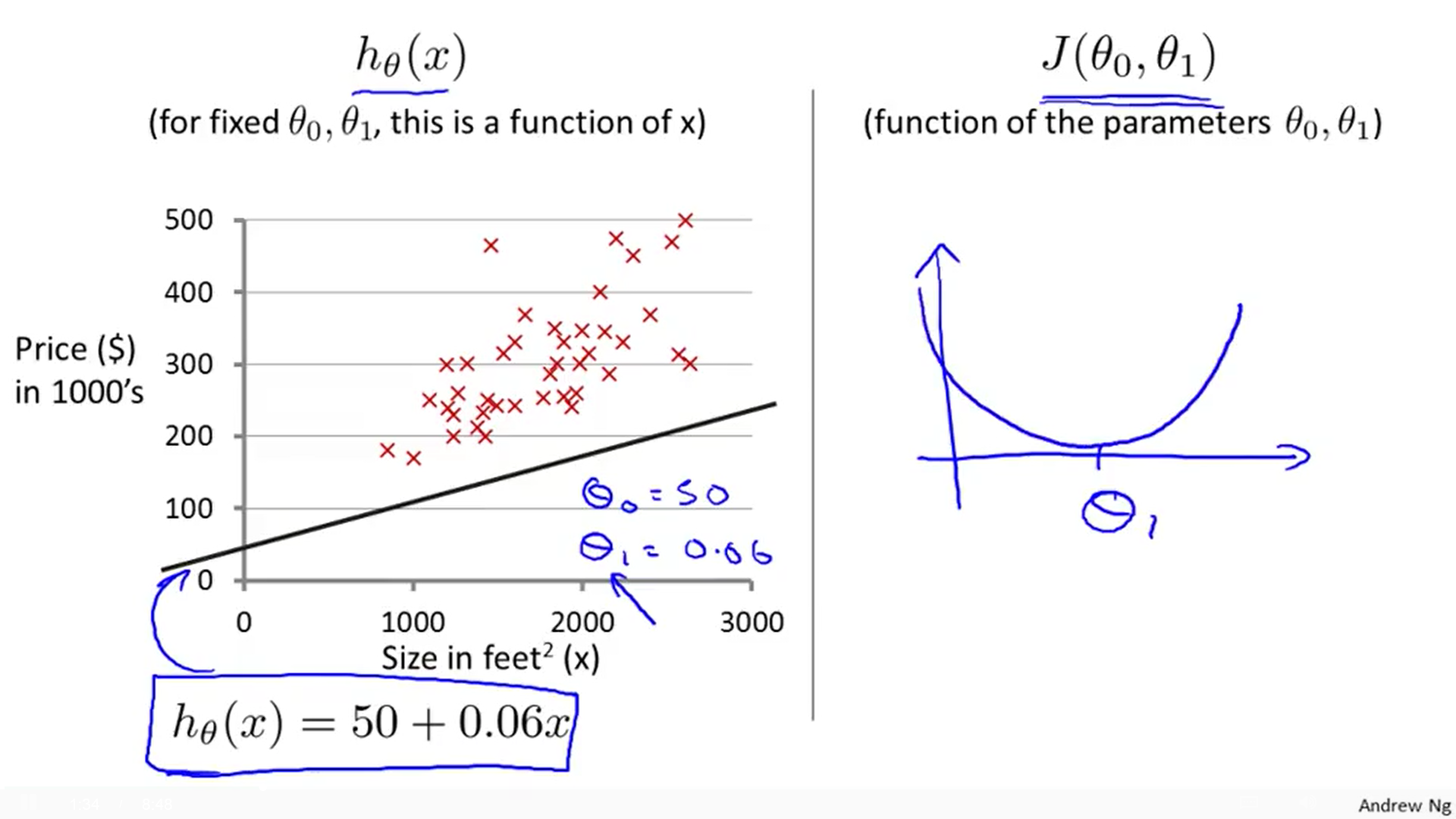
# Supervised learning

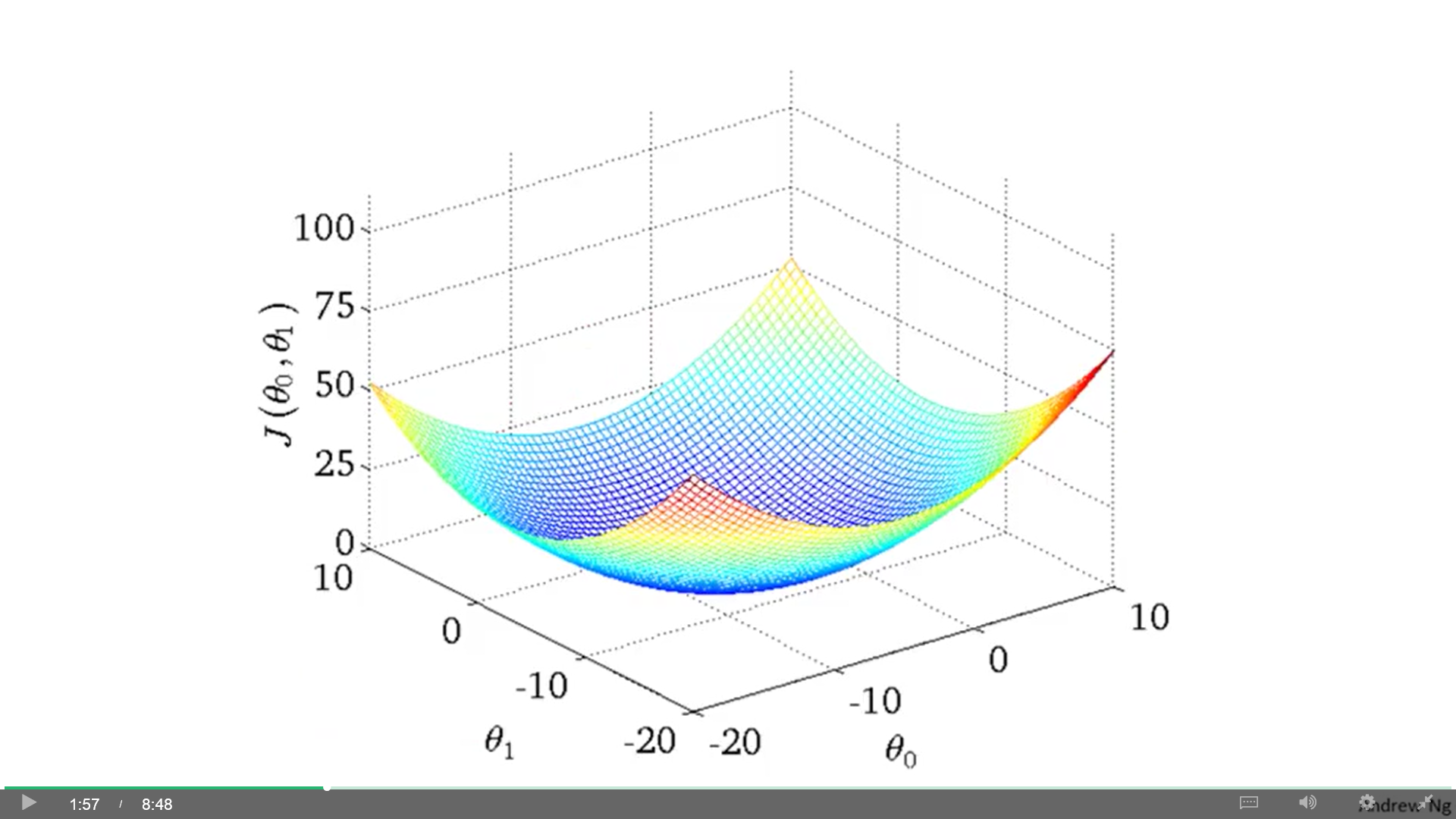
## Linear regression

Model a body (line, surface, volume …) one dimension less than the plot (dimensions: y, x1, x2 … xn) and adjust it to the data points. Cost function is convex (only has global minimum, no local minimums). Parameters θ are usually set to 0 at the beginning.

**Hypothesis:** hθ(x) = θ0x0 + θ1x1 + θ2x2 + … + θnxn (x0=1)

hθ(x(i)) = θT∙x(i)

**Cost function:**

**Batch Gradient descent:**

**Normalization:** (helps GD to run and converge faster) (don’t apply to x0=1)

* Feature scaling: Get every feature into approx. a range [-1, 1]. Divide by range (s = max-min) or Standard deviation (SD).
* Mean normalization: Make features have approx. zero mean (μ). Used with feature scaling. Subtract the mean.

Before making predictions, normalize x using the mean and SD previously applied to the training set.

* Learning rate (α): Plot J(θ) as function of number of iterations. Plot should decrease in each iteration (if not, decrease α). Check different values of α (recommended: try values on a log-scale at about 3 times the previous value: 0.1, 0.03, 0.01, 0.003, 0.001).
* Features edition: A combination of Length and Depth (Area) may be what really determines the output (house price).

**Polynomial regression:** Quadratic/cubic/etc. polynomial model, rather than linear, may fit the data better. Add additional features with any choice of feature you want (…). Even though we add polynomial terms, we are still solving a linear regression optimization problem.

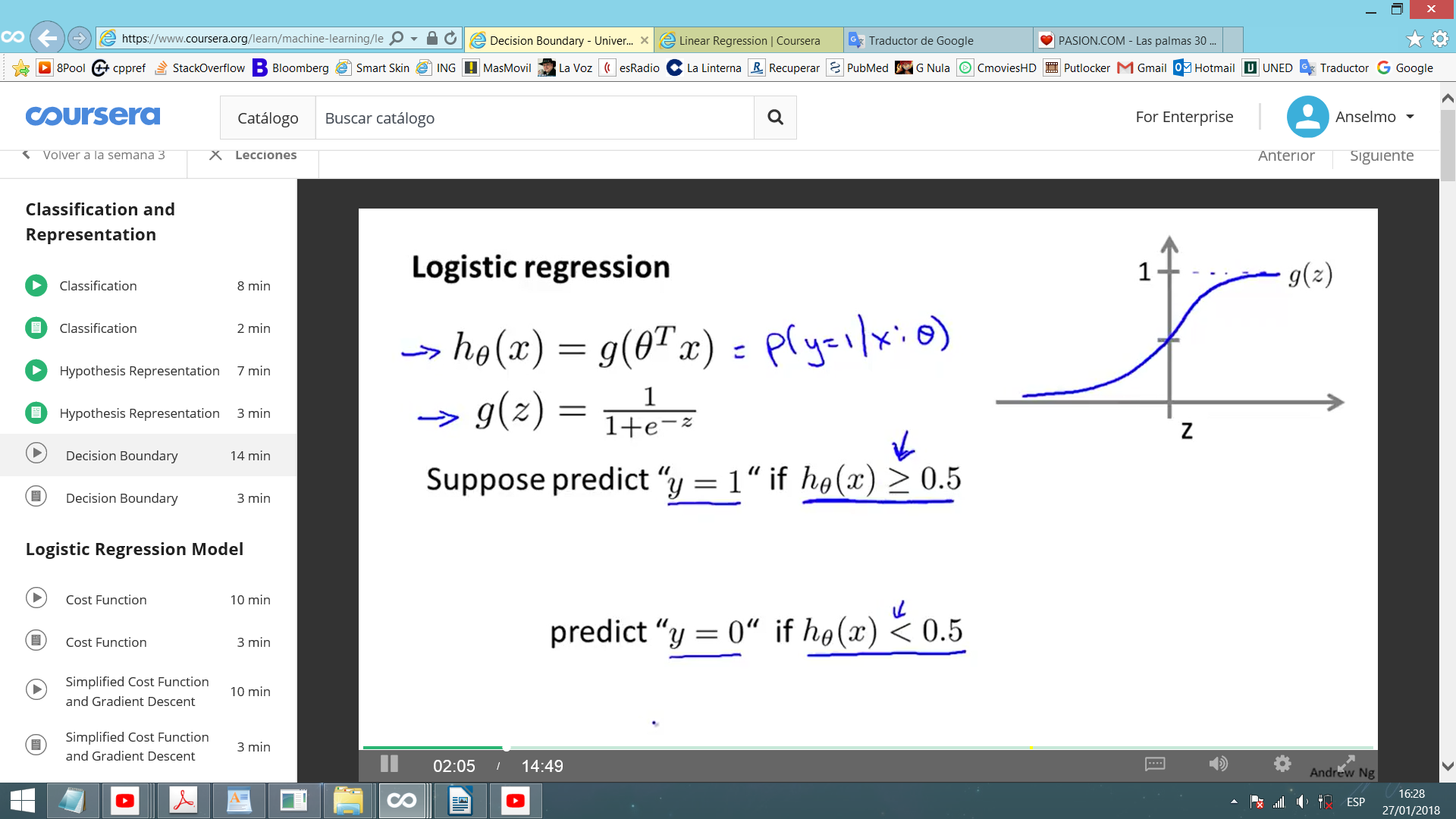
[ θ0 + θ1x + θ2x2 + θ3x3 ] > hθ(x) = θ0 + θ1x1 + θ2x2 + θ3x3 (x1 = size) (x2 = size2) (x3 = size3)

**Normal equation:** Solves θ analytically in linear regression problems. Take partial derivative of J(θ) with respect to every parameter θ and set these to 0 to get the θ values that minimize J.

**θ = (XT X)-1 XT y** Acceptable for about n ≤ 10.000 (GD is good for n ≥ 100.000).

(XT·X) may be non-invertible (maybe redundant features or m ≤ n [delete some features or use normalization]). Use inv() or pinv() (we always get an inverse, but it’s not clear that it gives a very good hypothesis). No need of feature scaling. Need to add x0 = 1.

## Logistic regression

Classification algorithm where h(x) satisfy 0 ≤ h(x) ≤ 1. (y is valued: y Î {0, 1})

**Hypothesis:**

(sigmoid/logistic function) The result is the probability of hθ(x) = 1.

Let's suppose we predict: y = 1 if hθ(x) ≥ 0.5 (only happens when θTx ≥ 0).

y = 0 if hθ(x) < 0.5, which only happens when θTx < 0.

All the features are points in an n-dimensional space and z draws valued regions (0, 1 …) delimited by a decision boundary (where h(x)=0.5) (property of h(x) under the parameters). We can make linear boundaries (simple polynomial) or non-linear boundaries (adding higher polynomial terms, like in linear regression). Include x0=1.

**Cost function:**

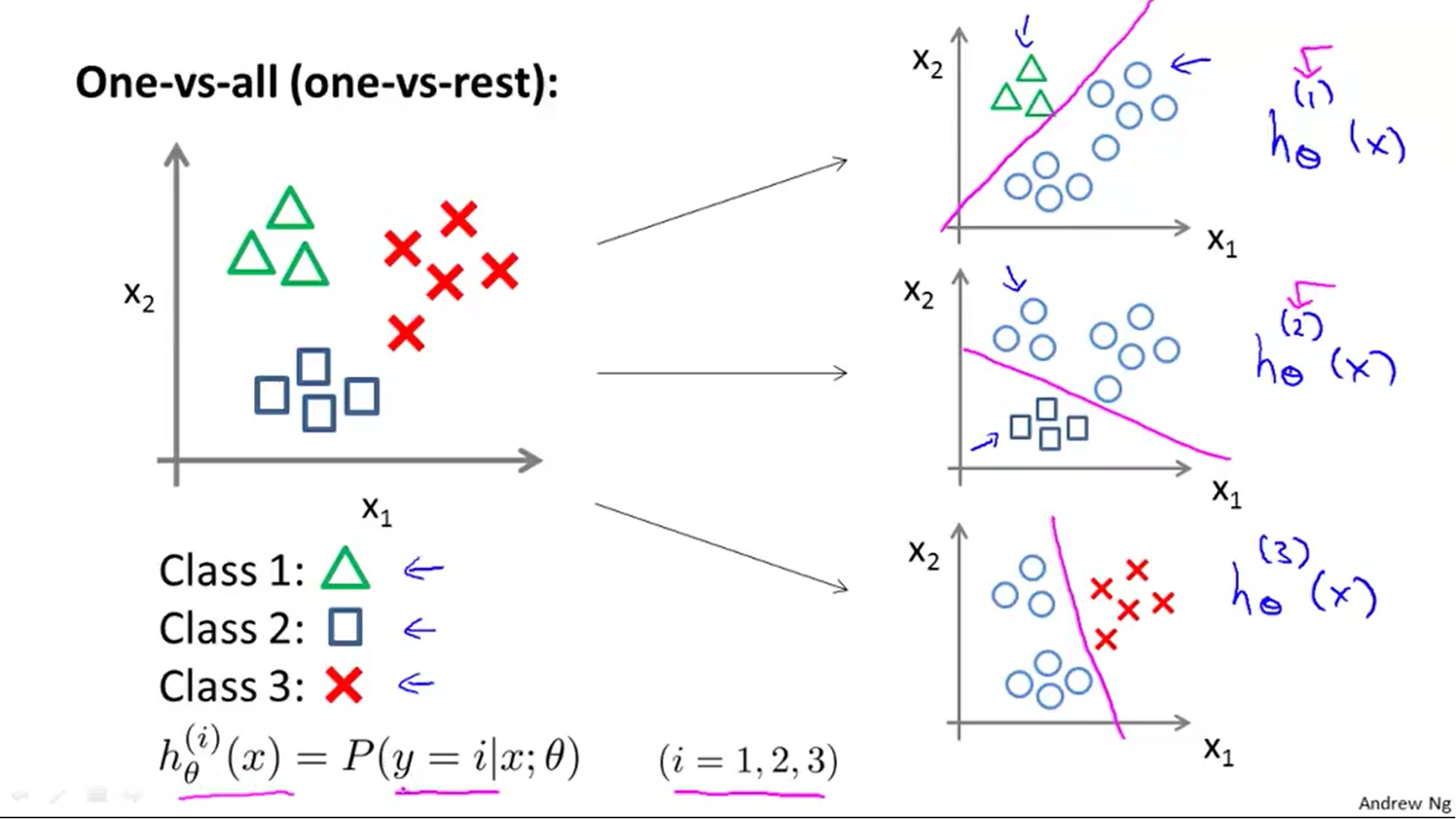
This J(θ) is convex. We use a modified version of the linear reg. cost function because that is not-convex in logistic reg. There are other options for cost function.

**Gradient descent:**

Identical to the one in linear regression, except that the definition of h(x) changes. Like in linear reg., you can plot J(θ) against number of iterations to see if it’s converging. Feature scaling helps GD to converge faster.

Other (better) optimization algorithms, given J(θ) and : **Conjugate gradient**, **BFGS**, **L-BFGS**… Faster, no need to pick α, but very complex. Use libraries, Octave and MATLAB have good ones. For linear and logistic reg., Octave has **fminunc** (an optimized version of GD) and **fmincg** (similar to fminunc but more efficient for dealing with large number of parameters).

**Multi-class classification: One vs all classification algorithm**

Problems with more than two classes y Î {0, 1, 2, 3 ...} (start from 0 or 1, it doesn’t matter).

We take the different classifiers and consider each one as a binary classification problem where that one classifier is opposed to the rest. We train a logistic reg. classifier for each class i to predict the probability that y = i. On a new input x, to make a prediction, run all the classifiers on x and pick the class i that maximizes .

## Regularization (linear and logistic regression)

Underfitting (high bias): Poor fit to data.

Overfitting (high variance): Too many features, too high order polynomial. Solutions:

* Reduce number of features: Manually or automatically (model selection algorithm)
* Regularization: Keep features, but reduce magnitude of parameters θ

Regularization: Adding an extra term to the end of J(θ) we reduce the values of parameters θ, which results in a simpler and smoother hypothesis (less prone to overfitting). By convention, we don’t regularize θ0 (makes little difference). Too big λ can result in underfitting (hθ(x) ≈ θ0).

**Linear regression:**

* Cost function:
* Gradient descent:

// for θ0

// for θj (j={1,2…n})

**Normal equation:**

(XT·X+λZ) is always invertible if λ > 0 (even when m ≤ n)

**Logistic regression:**

* Cost function:
* Gradient descent (identical to linear reg.):

// for θ0

// for θj (j={1,2…n})

## Functions

**Linear regression:**

* **Hypothesis (ML):**

hθ(x(i)) = θT∙x(i)

* **Cost function (ML):**
* **Cost function (DL):**

<<<

* **Batch gradient descent (ML):**

repeat {

// for θ0

// for θj (j={1,2,3..n})

}

Normal equation:

**Logistic regression:**

* **Hypothesis (ML):**
* **Cost function (ML):**
* **Cost function (DL):**

First term: Addition of every cost function of every output unit (k) for all examples (i).

Second term: Addition of every parameter Θ in the network (not including those that correspond to the bias: Θ0). Every layer (l), unit (j), parameter (i).

* **Batch gradient descent (ML):**

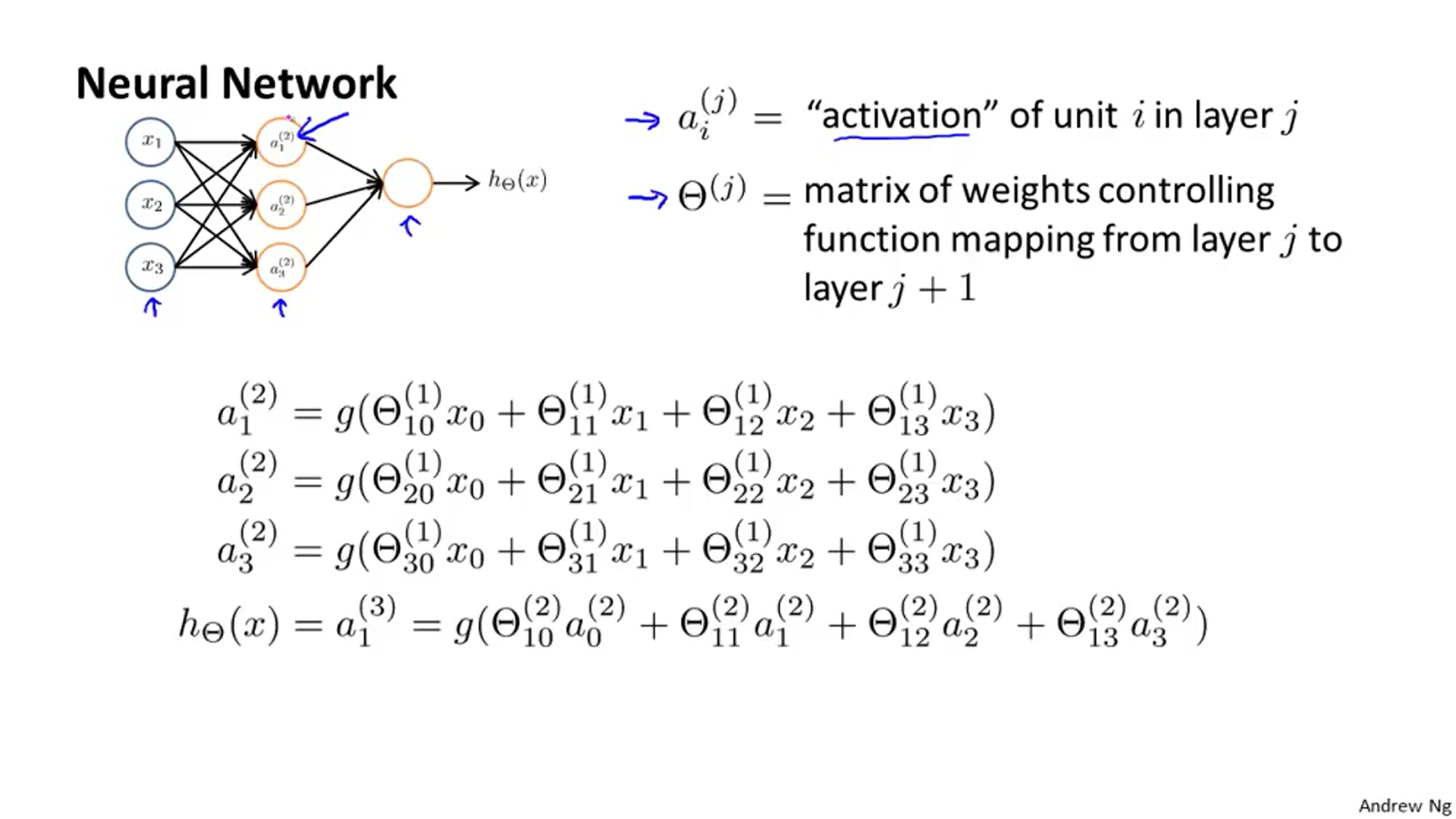
repeat {

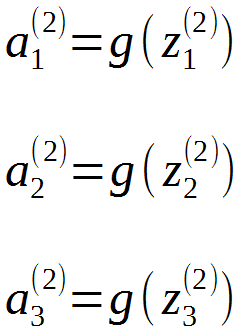
// for θ0

// for θj (j={1,2,3..n})

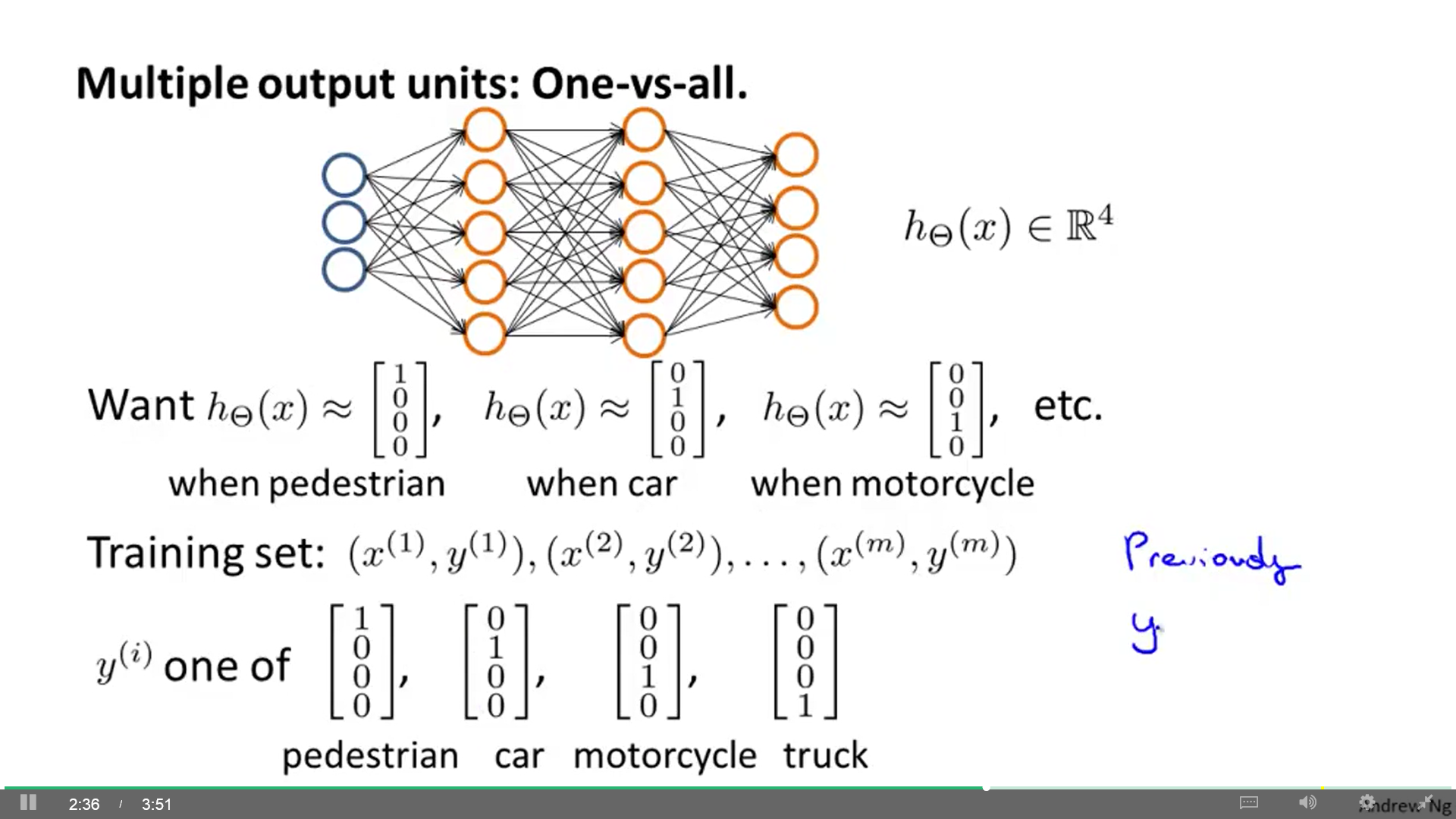
}

## Neural networks

When we apply logistic regression including many polynomial features in the hypothesis, we can end with a huge amount of polynomial terms, especially when n is large. Neural networks are much better to learn complex non-linear hypothesis, even when n is large. You can add more features (ex.: polynomial features) to log. reg. but that can be very expensive to train.



**Feed forward propagation:** Process of computing hΘ(x). We use Θ instead of θ. Very similar to logistic regression, except that the NN learns its own complex features from the original features. Deeper layers compute more complex features.

**Multiclass classification:** Similar to One Vs All method. Our NN will have a number of output units, where each one classify one category. We may use fmincg.

Training set:

**Cost function:**

* First term: Addition of every cost function of every output unit (k) for all examples (i).
* Second term: Addition of every parameter Θ in the network (not including those that correspond to the bias: Θ0). Every layer (l), unit (j), parameter (i).

**Backpropagation:** To minimize J(Θ) using an optimization algorithm, we need to supply J(Θ) and the partial derivative terms for every θ in the NN. Backpropagation is an algorithm for computing the derivatives. Errors (δ) of each activation value are back propagated through layers; then, they are used together with the activation values to compute the derivatives for each θ. For one example (direct order):

Error of the output layer:  (yk : kth element of vector y)

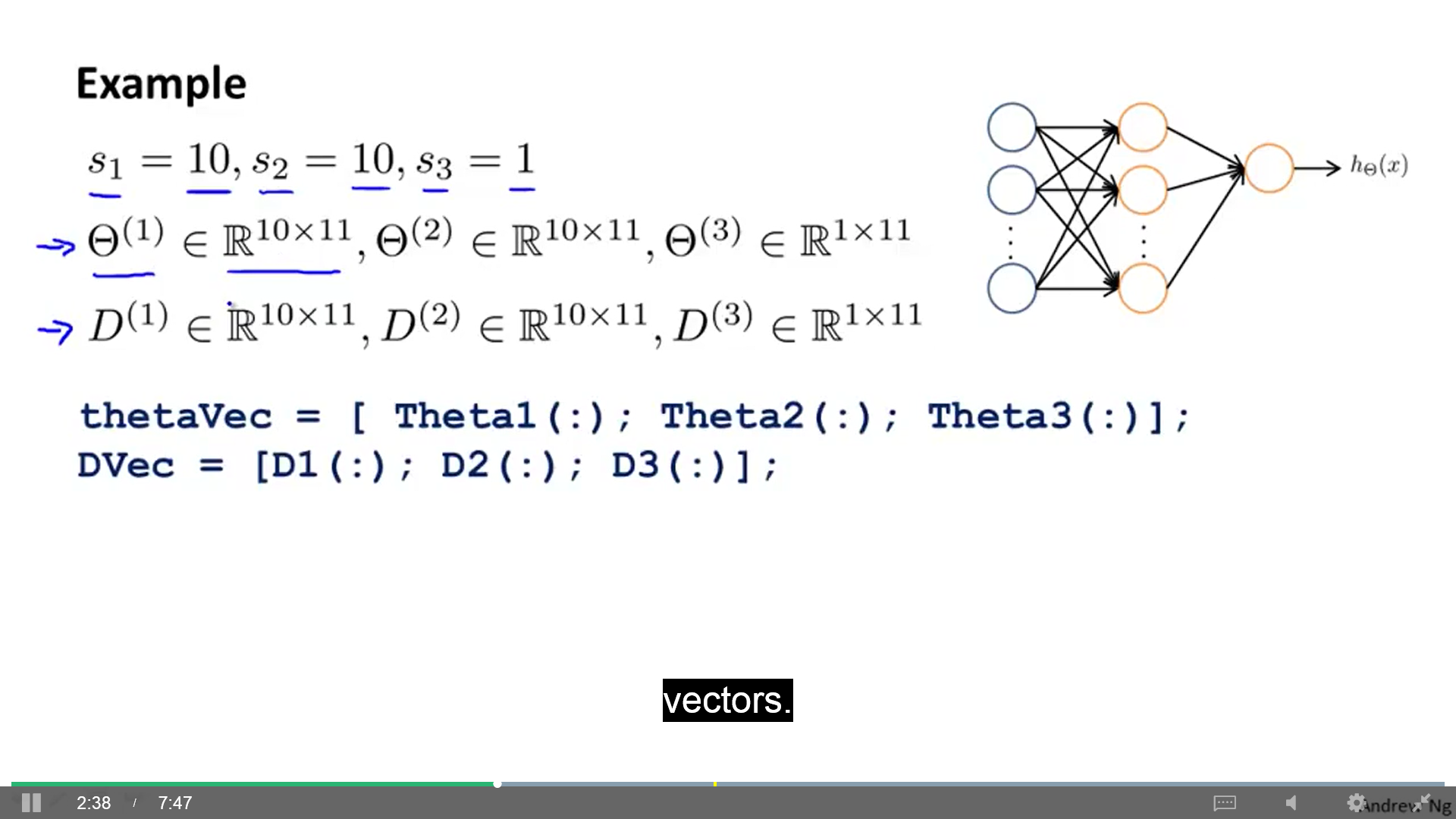
Error of earlier layers: Derivative:

Don't include θ0 in the previous computation of δ. Also, there is no δ(1) (input layer has no errors).

Partial derivative: supposing no regularization (λ=0)

**Synthesis:** For i to m: (x(i), y(i)) (a(1) = x(i))

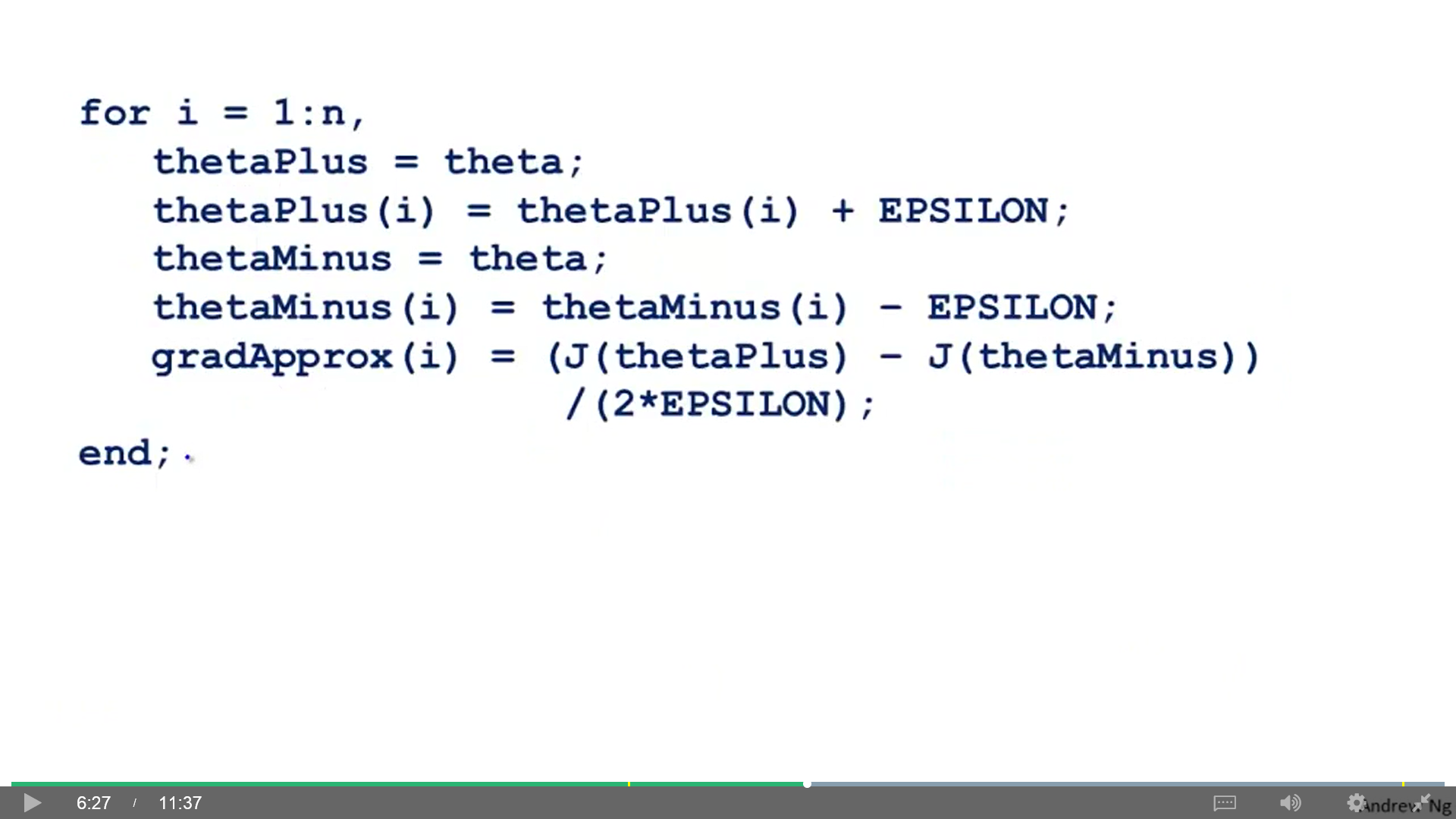
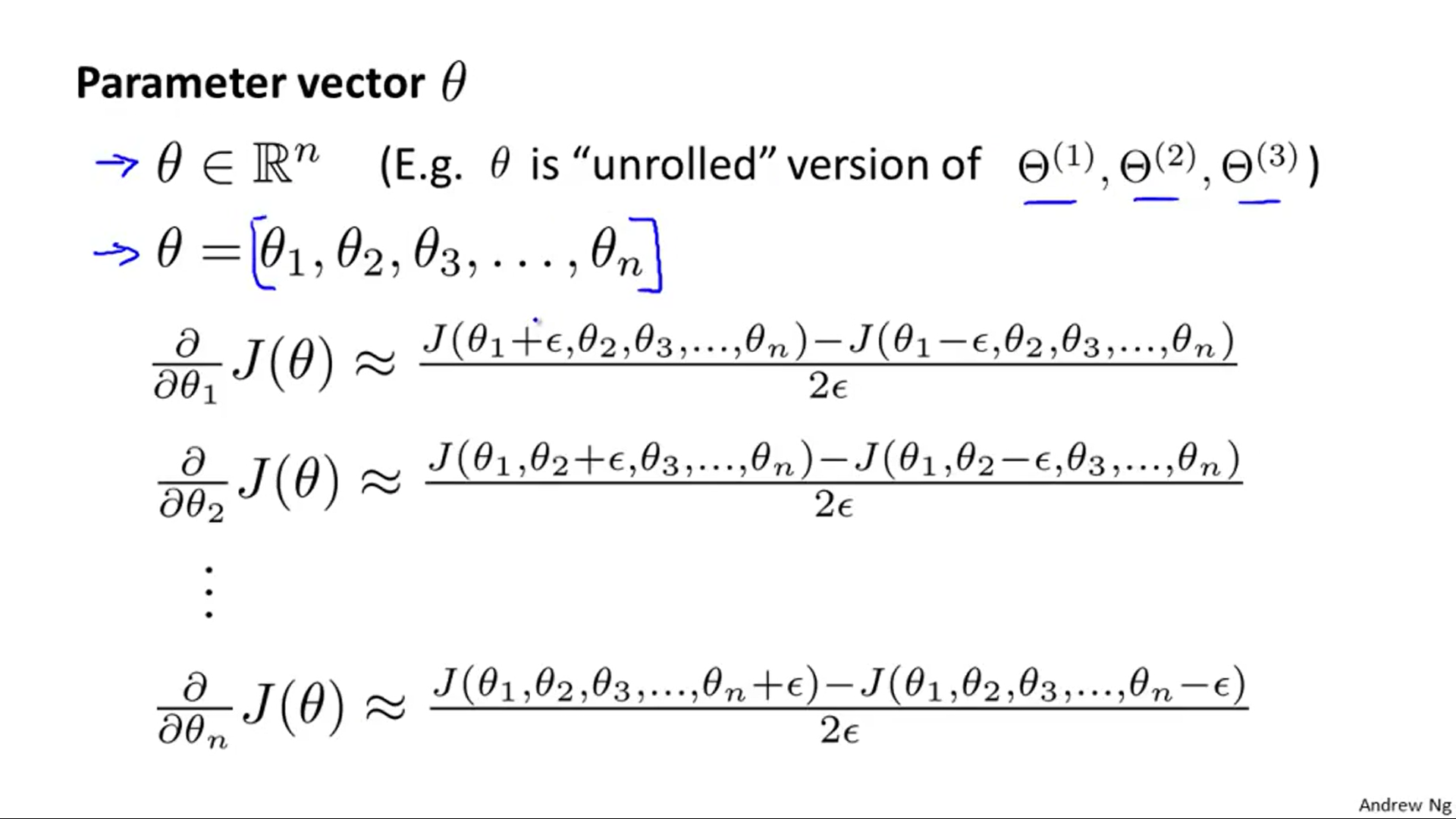
* Compute a(l) for l = 2, 3 … L (forward propagation)
* Compute J(Θ)
* Compute δ(L) = a(L) – y(i). (back propagation)
* Compute δ(L-1), δ(L-2) … δ(2).
* Compute partial derivatives for each θ and accumulate them in
* Compute final derivatives: if j = 0 if j ≠ 0
* Use GD or advanced optimization method to minimize J(Θ).

**Unrolling parameters from matrices into vectors:** Process needed in order to use advanced optimization routines (fminunc…). If L=4, we have 3 parameter matrices () and 3 derivative matrices ().

Unroll Θ to pass it to fminunc and to costFunction (δ). Then, reshape Θ. Get D and unroll it.

**Gradient checking:** BP may have bugs (worst performance) but still decrease the cost function. Gradient checking calculates an approximation of J(Θ) that can be compared with the gradients from BP or any similar GD algorithm (on NN or any other complex model). This works for any function where cost and gradient are computed. The parameters should be initialized with a fixed set of values.

ε: Really small number (like 10-4)



After computing D(1), D(2), ..., compute the approximate gradients and compare both. They should be similar, up to a few decimal places. Turn off gradient checking before using BP for serious training.

**Random initialization:** In NN, if we zero initialize Θ, a symmetry appears (after each iteration, parameters are identical). We use random initialization: Initialize each to a random value in [-ε, ε].

Theta1 = rand(10,11)\*(2\*INIT\_EPSILON) – INIT\_EPSILON;

Theta2 = rand(1,11)\*(2\*INIT\_EPSILON) – INIT\_EPSILON;

One effective strategy: sl , sl+1 : number of units in the layers adjacent to Θ(l)

**Training set accuracy:** mean(double(prediction == y)) \* 100

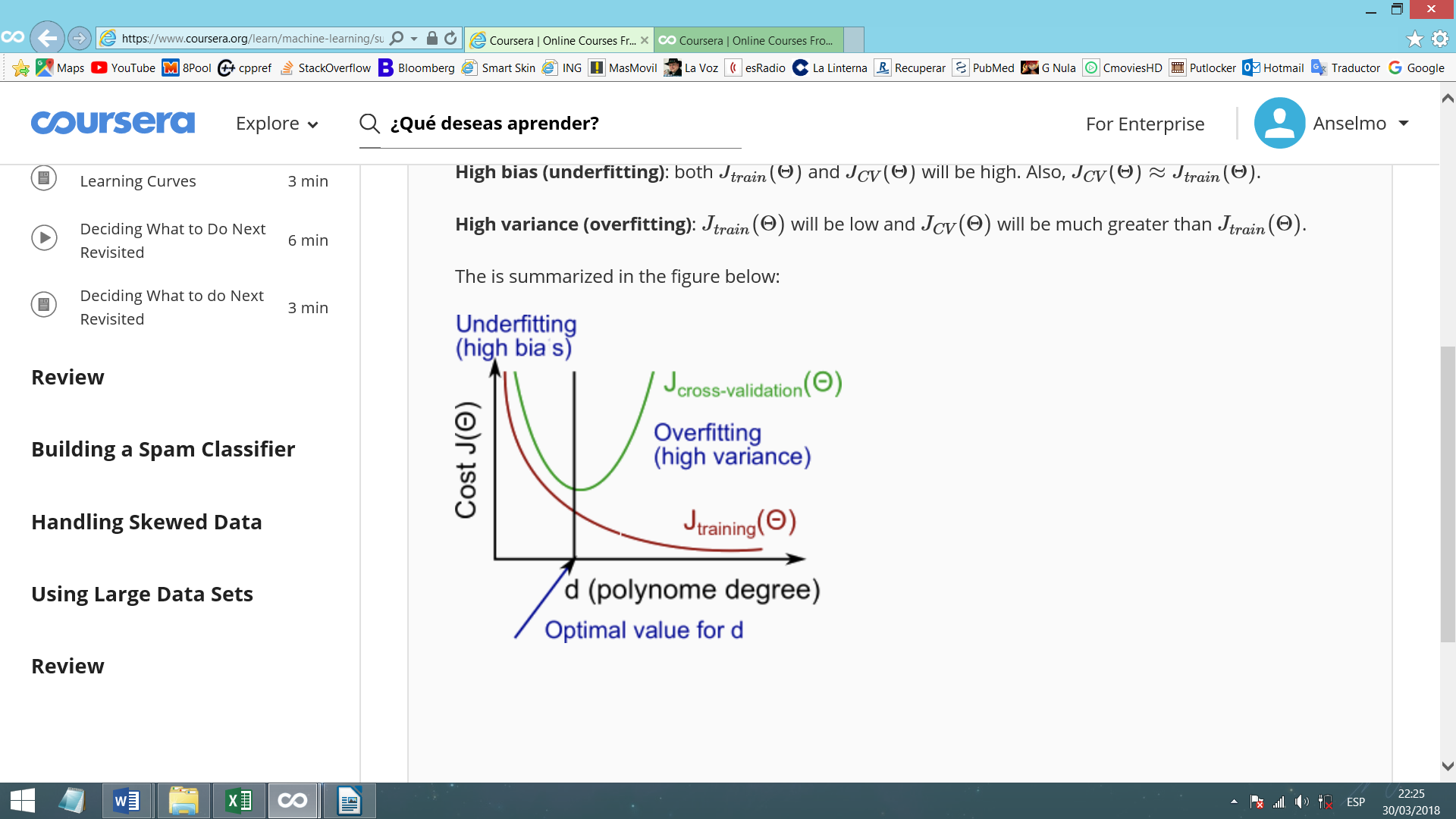
## Advice for machine learning

**Visualizing the hidden layer:** One way to understand what your NN is learning is to visualize the representations captured by the hidden units. Informally, given a particular hidden unit, one way to visualize what it computes is to find an input x that will cause it to activate (). We can display Θ(1) as images (hidden units corresponds roughly to detectors that look for strokes and other patterns.

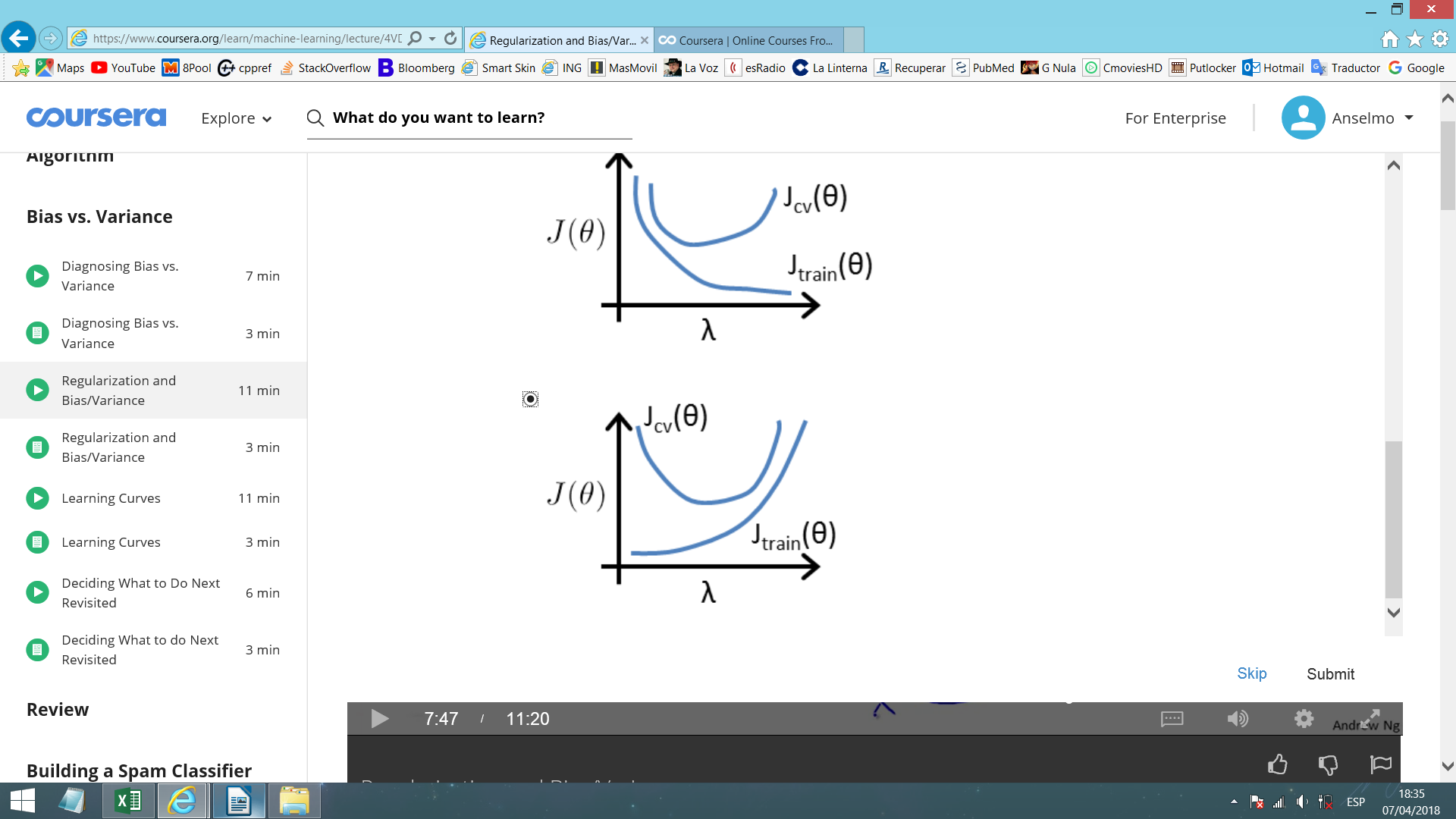
**Evaluating hypothesis:** Hypothesis may fit the parameters but fail to generalize to new examples (overfitting). Randomize the training set and split it in 2 parts: 70% training set, 30% test set.

* Learn parameter θ from training data
* Compute test error. Two options:
  + Compute Jtest(θ)
  + Compute 0/1 misclassification error (an example is right or wrong) (not applicable to linear regression):

**Model selection problem:** What degree polynomial should we choose to fit to data (linear, cuadratic, cubic…)? Let’s establish a 60% training set, 20% cross validation and 20% test set.

* Minimize J(θ) and get θ for every model.
* Compute Jtest(θ) to select a model.
* Compute Jcv(θ) to evaluate the chosen model.

**Diagnosing Bias (underfitting) vs Variance (overfitting):** In a bias problem, J(Θ) and Jcv(Θ) are high. In a variance problem, J(Θ) is low and Jcv(Θ) is high. Regularization (λ) can prevent overfitting. You can plot the graph.

**Automatic choosing λ:** Previously, when computing Jtrain(Θ), Jcv(Θ) and Jtest(Θ) we don’t use regularization (λ=0). For choosing λ, we select some values for λ (usually multiples of 2: 0, 0.01, 0.02, 0.04…). We minimize J(Θ) for each value of λ and pick the hΘ(x) with the lowest Jcv(Θ). To evaluate the chosen hypothesis, use Jtest(Θ). You can plot the graph.