Neural Networks

1. **Representation**

Origins: Algorithms that try to mimic the brain.

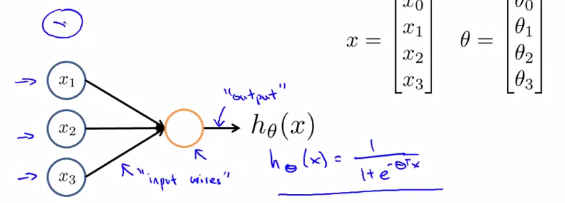
Was very widely used in 80s and early 90s; popularity diminised in late 90s

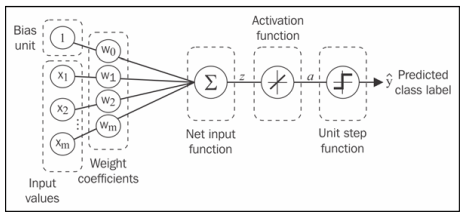
Recent resurgence: State-of-the-art technique for many applications.

Many more major breakthroughs resulted in what we now called deep learning algorithms, which can be used to create **feature detectors** from unlabeled data to pre-train deep neural networks – neural networks that are composed of many layers.

* 1. **Model representation(single-layer neural network):**

Neuron model : logistic unit, Sigmoid (logistic ) activation function.

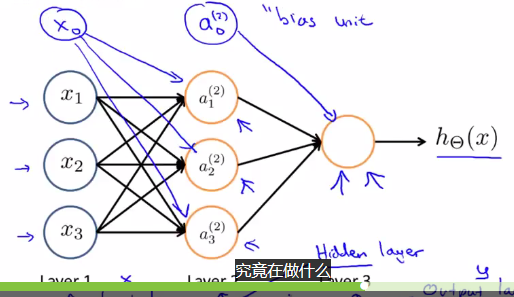




Another common choice for activation function is the hyperbolic tangent, or tanh,function:

C:\Users\phenix\AppData\Local\Temp\enhtmlclip\Image(1).png

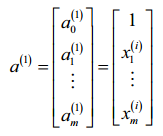
* 1. **Neural Network (Multi-layer perceptron (MLP))**

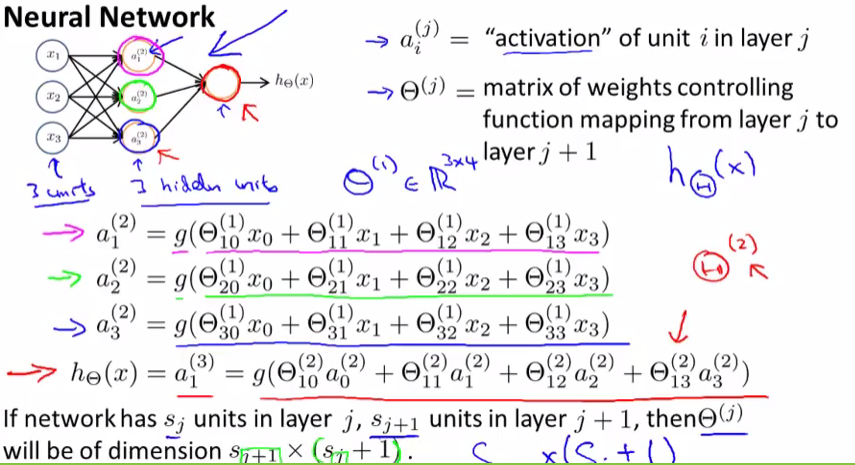


Practically, we can think of the number of layers and units in a neural network as additional hyperparameters that we want to optimize for a given problem task using the cross-validation.

However, the error gradients that we will calculate later via backpropagation would become increasingly small as more layers are added to a network. This vanishing gradient problem makes the model learning more chanllenging. tHerefore,special algorithms have been developed to pertain such deep neural network structures, which is called deep learning.

Activation units  and  are the bias units, respectively, which we set equal to 1.



Each unit in layer *l* is connected to all units in layer *l* +1 via a weight coeffcient. For example, the connection between the *k* th unit in layer *l* to the *j* th unit in layer *l* +1 would be written as.

Forward propagation:

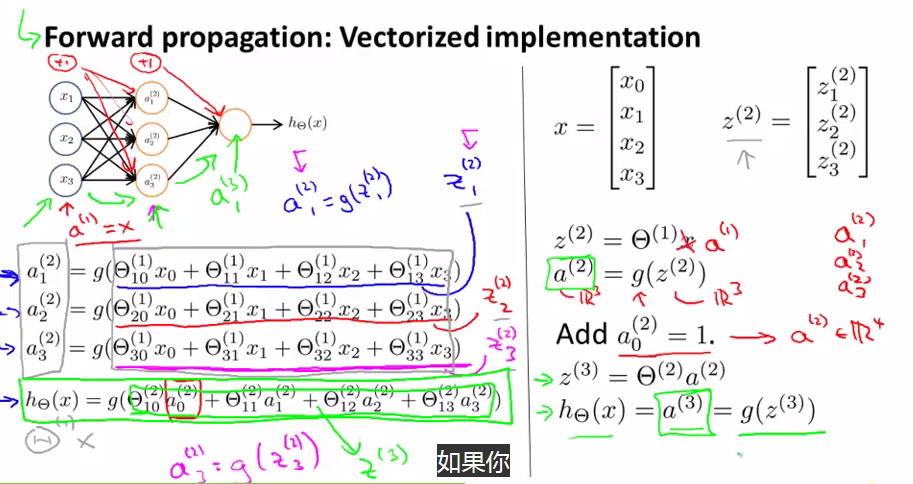
Three simple steps:

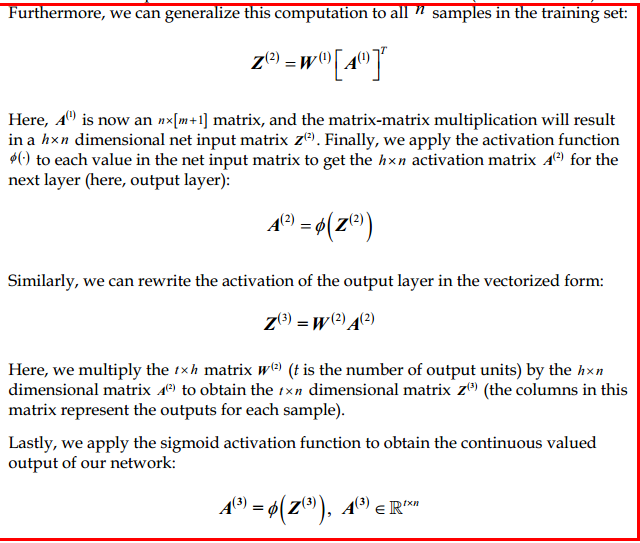
1. Starting at the input layer, we forward propagate the patterns of the training data through the network to generate an output
2. Based on the network’s output, we calculate the error that we want to minimize using a cost function
3. We backpropagate the error, find its derivative with respect to each weight in the network, and update the model.

Finally, after repeating the steps for multiple epochs and learning the weights of MLP , we use forward propagation to calculate the network output and apply a threshold function to obtain the predicted class labels in the one-hot representation.

**Feedforward** artificial neural network: refers to the fact that each layer serves as the input to the next layer without loops.

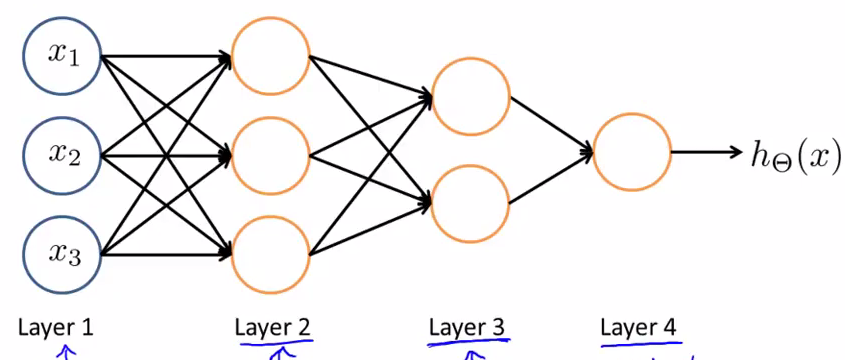
**Forward propagation : Vectorized implementaion**

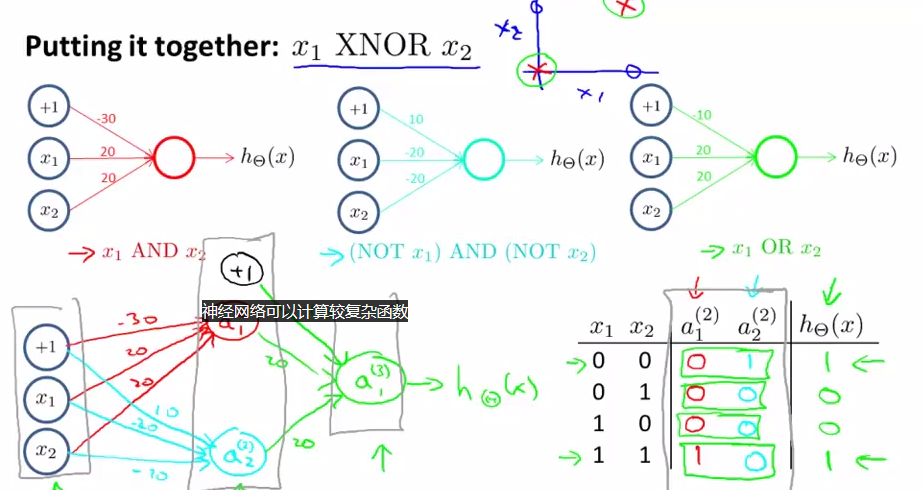




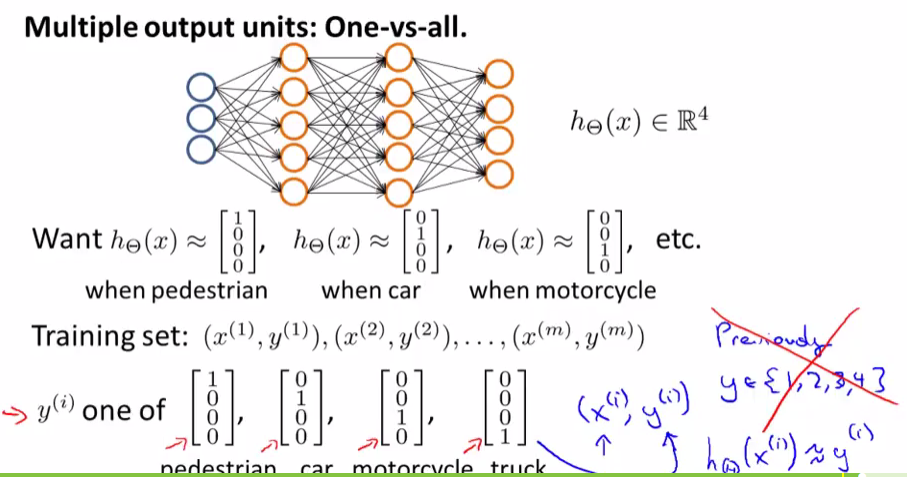
**Neural Network learning its own features**

Other network architectures

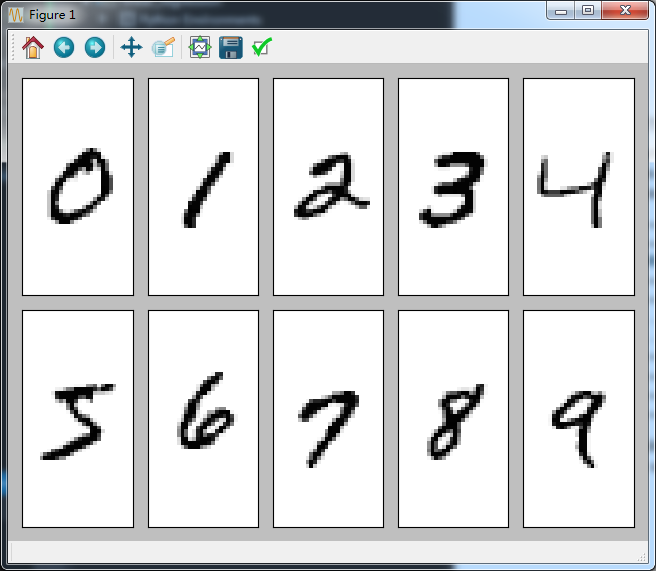


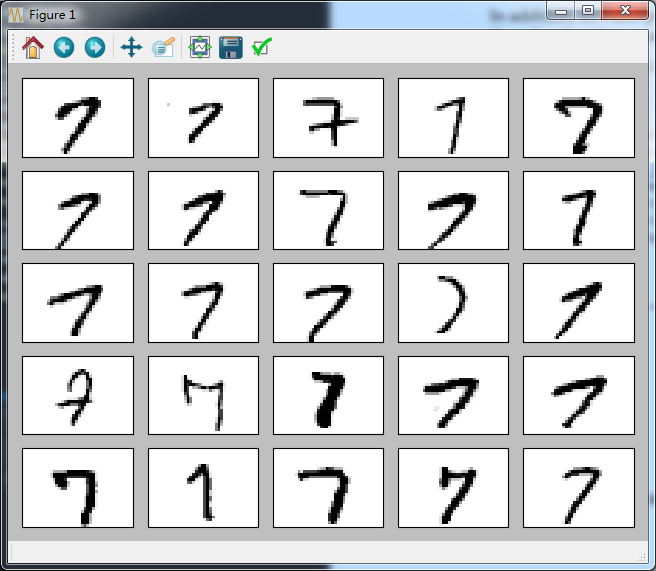


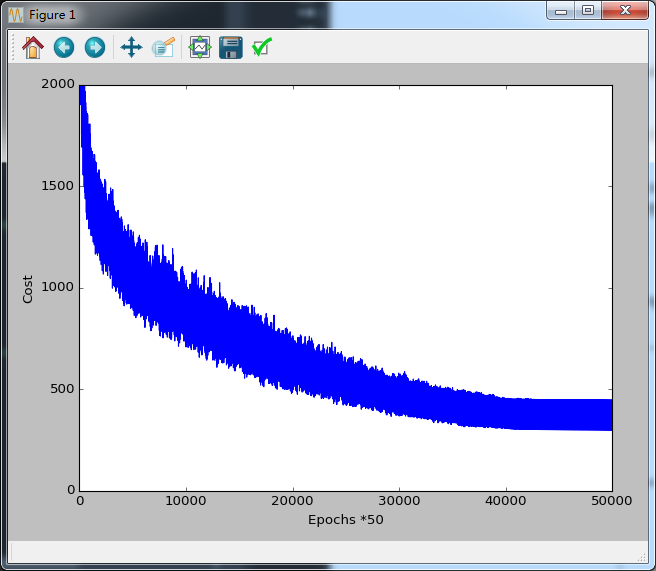
Multiclass Classification

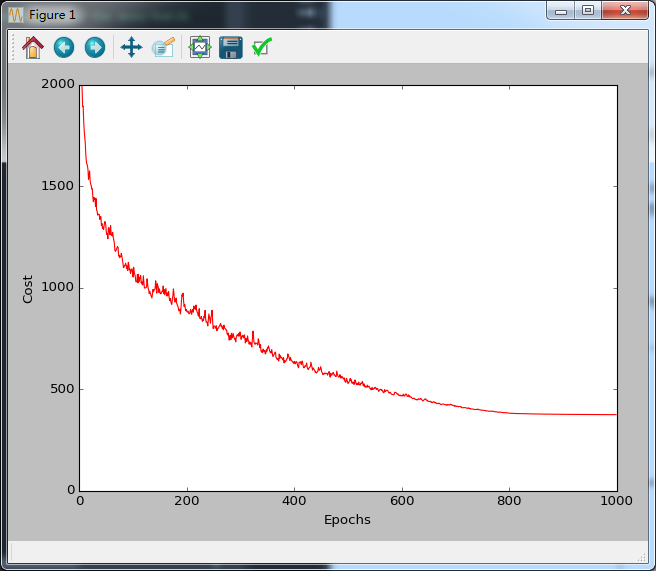


Example: classifying handwrite digits.

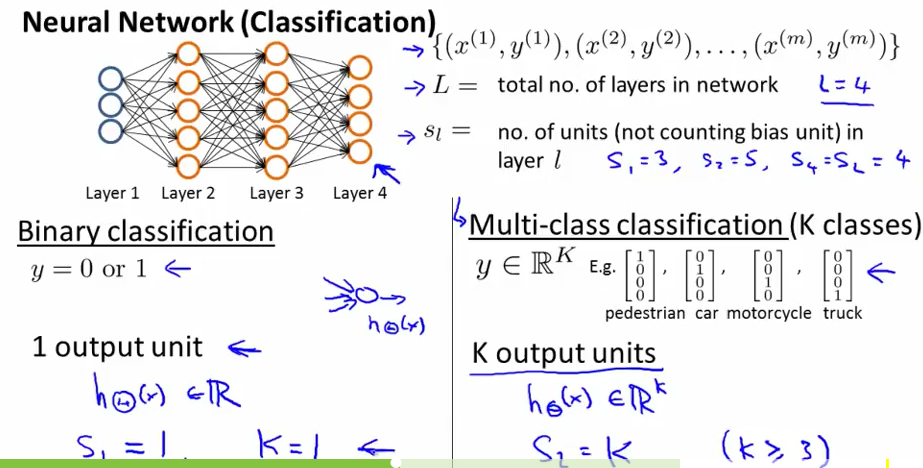


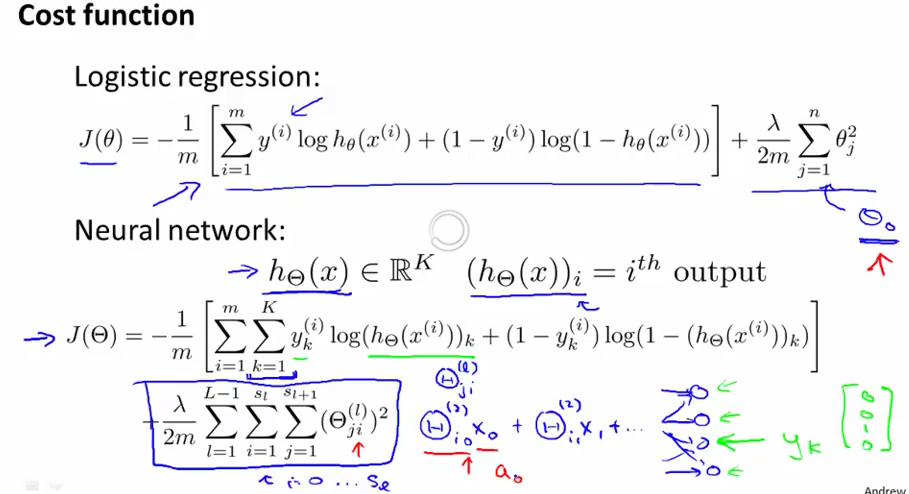




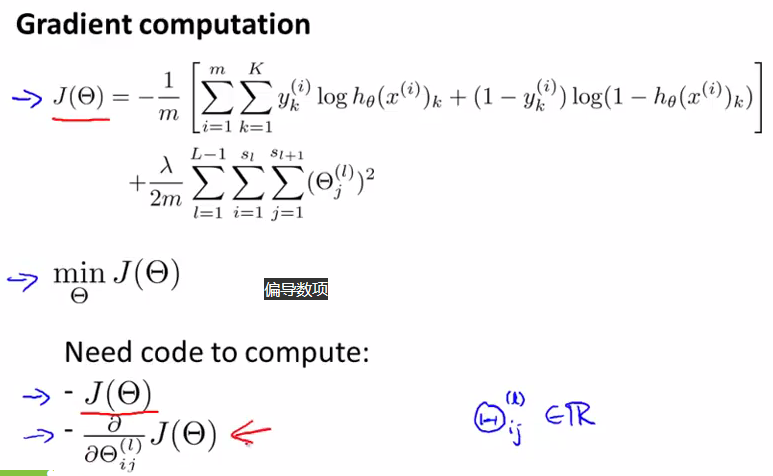


To further fine-tune the model, we could change the number of the hidden units, values of the regularization parameters,learning rate, values of the decrease constant, or the adaptive learning using the techniques that we discussed *Learning Best Practices for Model Evaluation and Hyperparameter Tuning.*

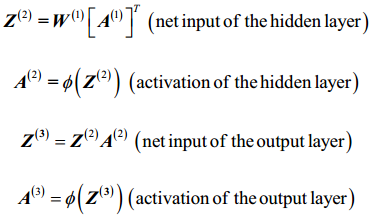
1. **Learning**



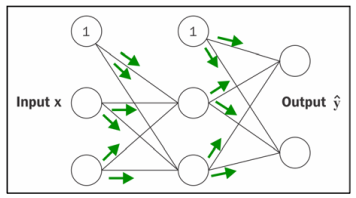
* 1. **Backpropagation Algorithm**



1. Apply forward propagation in order to obtain the activation of the output layer, which formulated as follows:



Concisely, we just forward propagate the input features through the connection in the network as shown here:



In backpropagation, we propagate the error from right to left. We start by calculating the error vector of the output layer:



Here, y is the vector of the true class labels.



Here,  is simply the derivative of the sigmoid activation function, which we implemented as 

Note that the asterisk symbol ( ) ∗ means element-wise multiplication in this context.

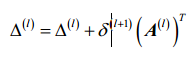
Eventually, after obtaining the  terms, we can now write the derivation of the cost function as follows:



Next ,we need to accumulate the partial derivative of every jth node in layer l and the ith error of the node in layer l+1:



Remember that we need to compute for every sample in the training set. Thus, it is easier to implement it as a vectorized version like in our preceding MLP code implementation:



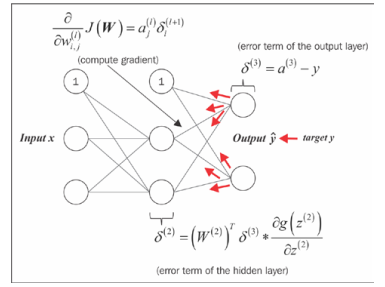
After we have accumulated the partial derivatives , we can add the regularization term as follows:

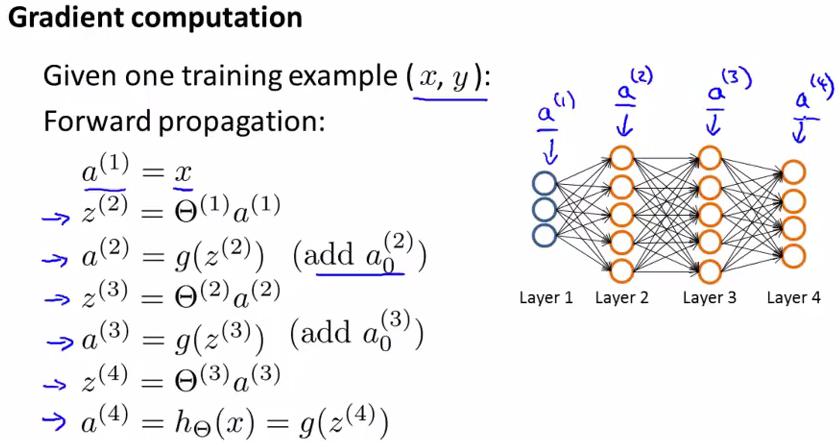


Lastly, after we have computed the gradients, we can now update the weights by taking the opposite step towards the gradient:

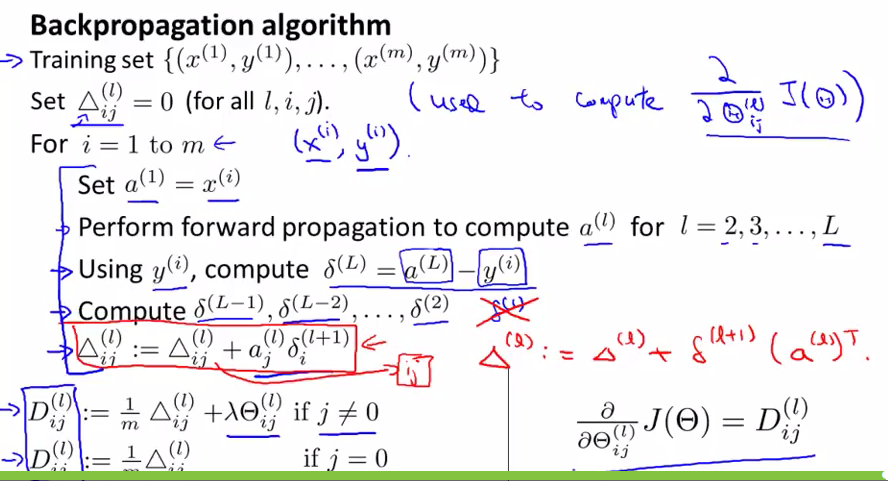


To bring everything together,





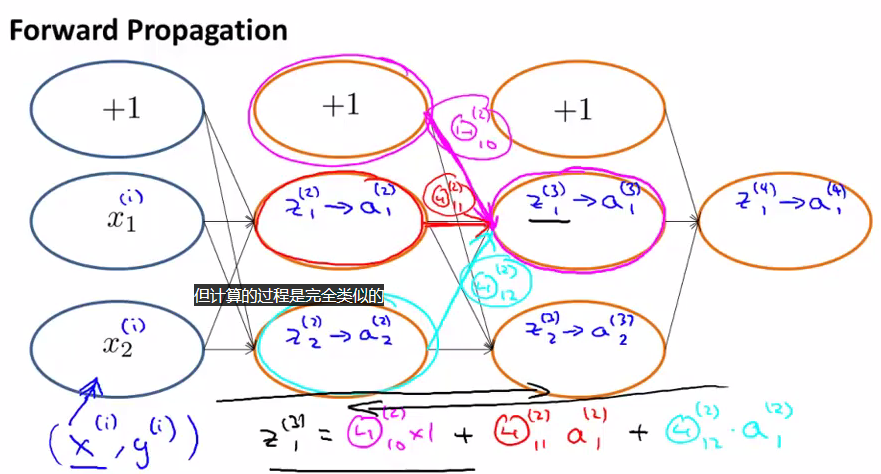


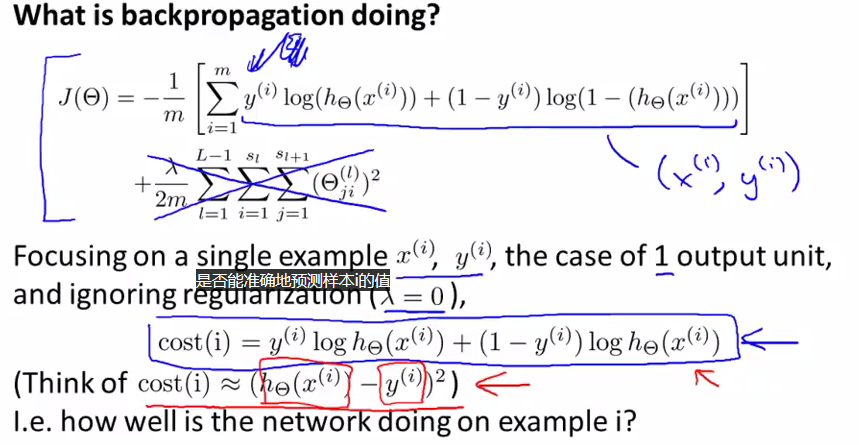


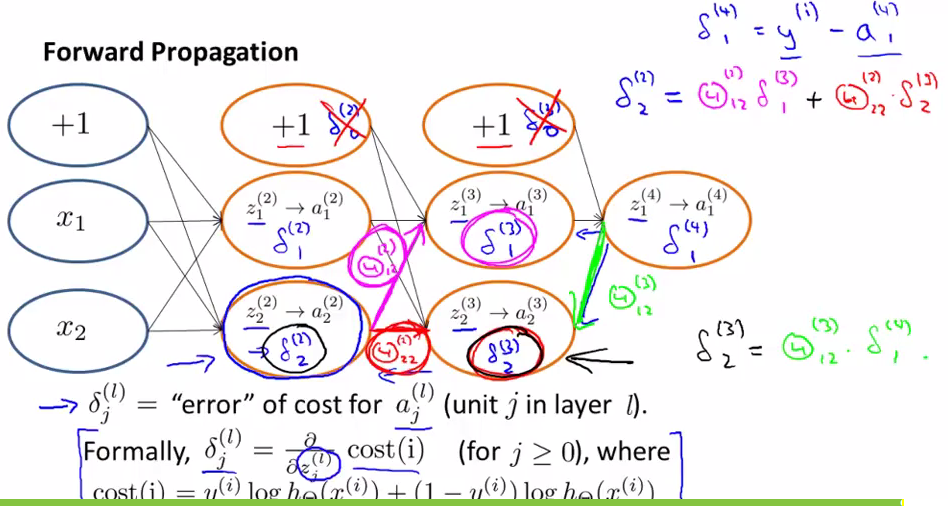
**Backpropagation Intuition**

The challenge in the parameterization of neural networks is that we are typically dealing with a very large number of weight coefficients in a high-dimensional feature space. In contrast to other cost functions that we have seen in previous chapters, the error surface of a neural network cost function is not convex or smooth. There are many bumps in this high-dimensional cost surface (local minima) that we have overcome in order to find the global minimum of the cost function.

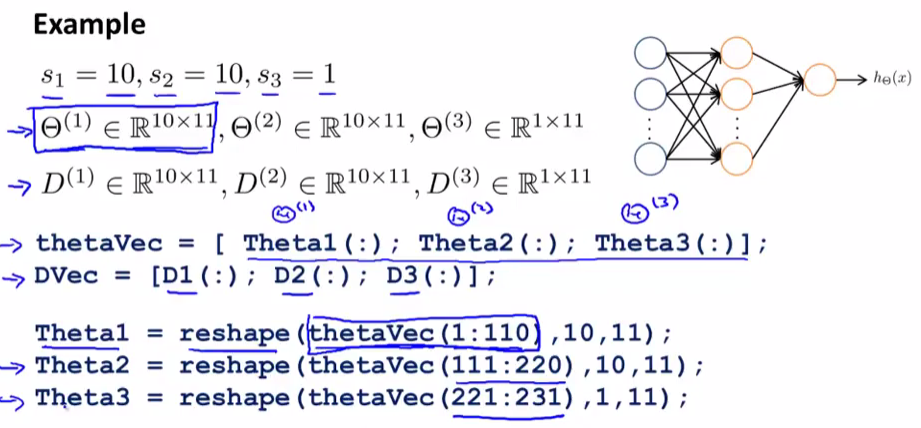
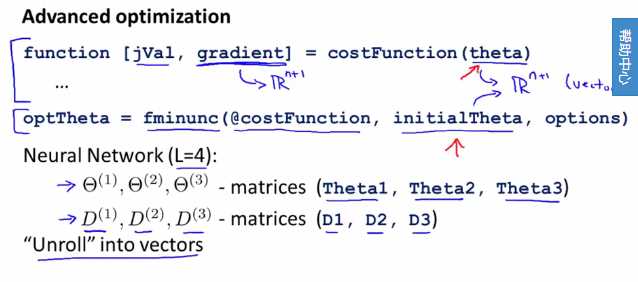
Automatic differentiation comes with two modes, the forward and the reverse mode, respectively. Backpropagation is simply just a special case of the reverse-mode automatic differentiation. The key point is that applying the chain rule in the forward mode can be quite expensive since we would have multiply large matrices for each layer (Jacobians) that we eventually multiply by a vector to obtain the output. The trick of the reverse mode is that we start from right to left: we multiply a matrix by a vector, which yields another vector that is multiplied by the next matrix and so on. Matrix-vector multiplication is computationally much cheaper that matrix-matrix multiplication, which is why backpropagation is one of the most popular algorithms used in neural network training.

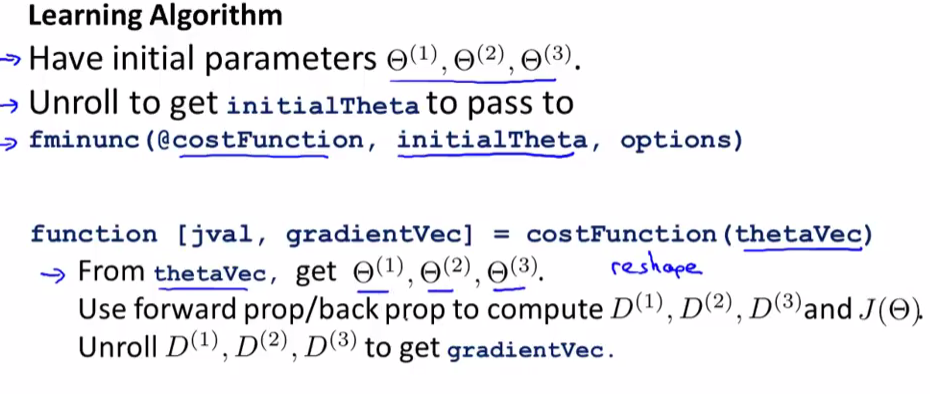




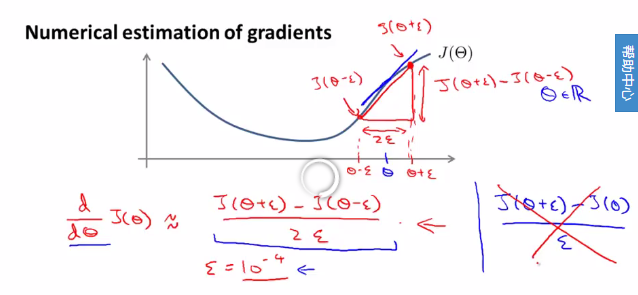


**2.2 Implementation Note : Unrolling Parameters**



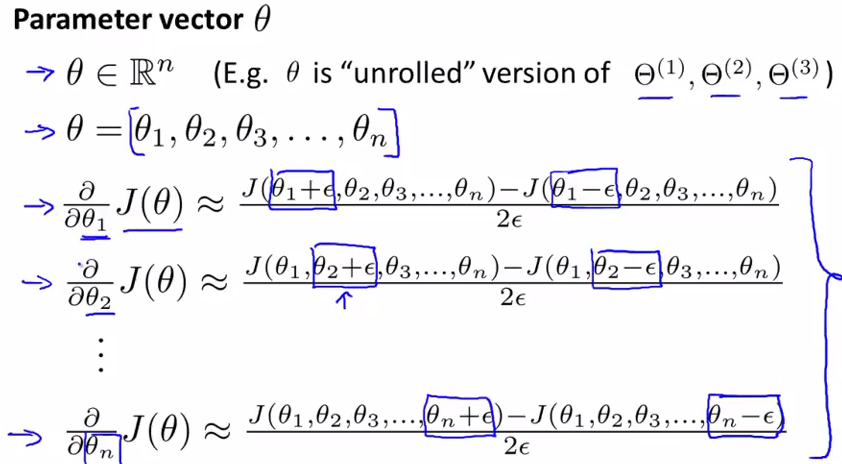


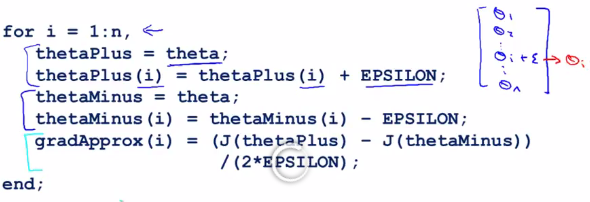
**2.3 Gradient Checking**



What is the acceptable error threshold to pass the gradient check? The relative error threshold for passing the gradient check depends on the complexity of the network architecture. As a rule of thumb, the more hidden layers we add, the larger the difference between the numerical and analytical gradient can become if backpropagation is implemented correctly.Since we have implemented a relatively simple neural network architecture in this chapter, we want to be rather strict about the threshold and define the following rules:

* Relative error <= 1e-7 means everything is okay!
* Relative error <=1e-4 means the condition is problematic, and we should look into it
* Relative error >1e-4 means there is probably something wrong it our code.





Check that grad Approx ~~Dvec

**2.4 Implementation Note:**

- Implementation backprop to compute DVec (Unrolled D(1),D(2),D(3))

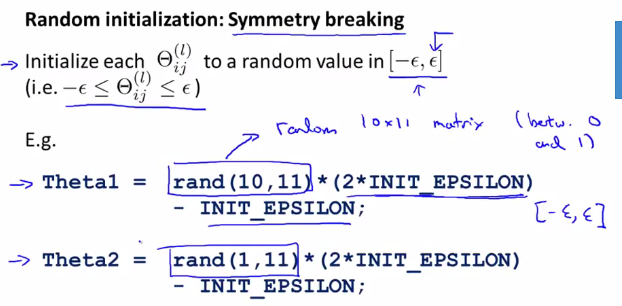
- Implement numerical gradient chekc to compute gradApprox

- Make sure they give similar values

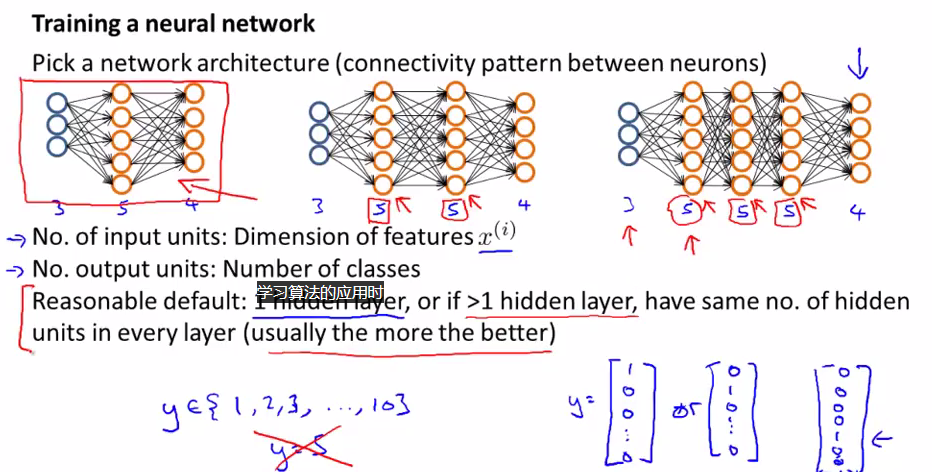
- Turn off gradient checking. Using backprop code for learning.

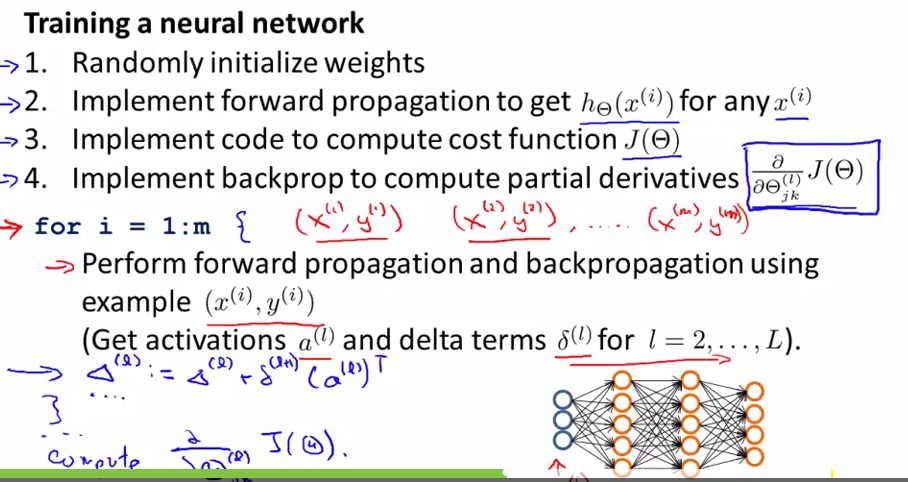
- Be sure to disable your gradient checking code before training your classifier. If you run numerical gradient computation on every iteration of gradient descent (or in the inner loop of costFunction(...)) your code will be very slow.

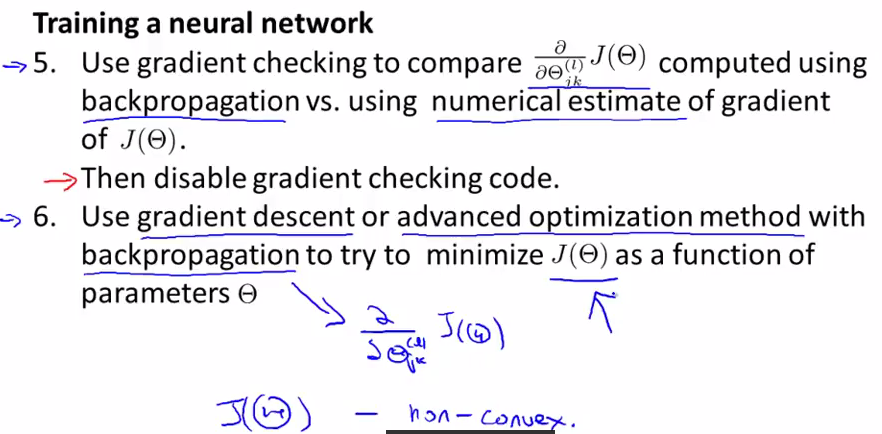
**2.5 Random Initialization**



Putting it Together

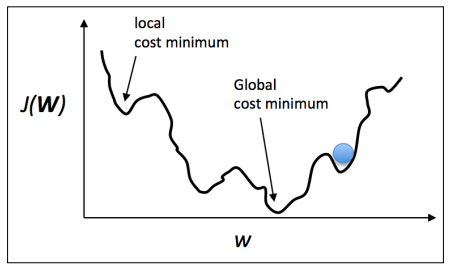






Convergence in neural networks

We add more tuning parameters such as the decrease constant and a parameter for an adaptive learning rate. The reason is that neural network are much harder to train than simper algorithms such as Adaline, logistic regression, or support vector machines. In multi-layer neural networks, we typically have hundreds, thousands, or even billions of weights that we need to optimize. Unfortunately, the output function has a rough surface and the optimization algorithm can easily become trapped in local minima, as shown in the following figure:



By increasing the learning rate, we can more readily escape such local minima. On the other hand, we also increase the chance of overshooting the global optimum if the learning rate is too large. Since we initialize the weights randomly, we start with a solution to the optimization problem that is typically hopelessly wrong. A decrease constant, which we defned earlier, can help us to climb down the cost surface faster in the beginning and the adaptive learning rate allows us to better anneal to the global minimum

Reference:

1. python machine learning
2. machine learning, Ng Video