Neural Networks

Andrew Ng (video tutorial from Coursera's "Machine Learning" class)

Transcript written* by *José Soares Augusto*, June 2012 (V1.0)

1 Non-linear Hypothesis

I'd like to tell you about a learning algorithm called a **Neural Network** (NN)[1]. We're going to talk about the representation, and then talk about learning algorithms for it.

Neural networks is actually a pretty old idea, which has fallen out of favor for a while. But today it is the state of the art technique for many different machine learning problems.

Why do we need yet another learning algorithm? We already had linear regression and logistic regression, so why do we need neural networks? In order to motivate the discussion, I'll show a few examples of machine learning problems where we need to learn complex nonlinear hypothesis, and then show you that logistic regression with nonlinear features is not feasible.

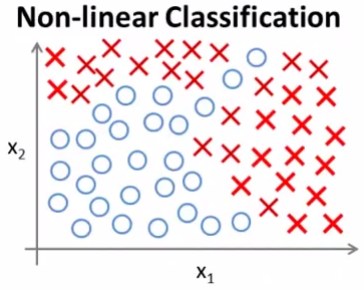


Figure 1: Nonlinear classification problem with two features, x_1 and x_2.

Consider the supervised learning classification problem in Fig. 1. If you want to apply logistic regression to this problem, one thing you could do is to use a lot of nonlinear features like in

$$g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1 x_2 + \theta_4 x_1^2 x_2 + \theta_5 x_1^3 x_2 + \theta_6 x_1 x_2^2 + \cdots)$$

where $g()$ is the sigmoid function, and we can include there lots of polynomial terms.

If you include enough polynomial terms, then maybe you can get a hypothesis that separates the positive and negative examples. This particular method works well when you have only two features, x_1 and x_2. But in practice many interesting machine learning problems would have a lot more features than just two.

Suppose you have a housing classification problem. You have different features of a house and you want to predict what are the odds that your house will be sold within the next six months. We can come up with quite a lot of features, maybe a hundred different features of several different houses

*With help from the English subtitles which are available in the "Machine Learning" site, `https://class.coursera.org/ml/lecture/`.

[1]In this document we will use the acronyms **NN** for Neural Network, **BP** for backpropagation, **BPA** for backpropagation algorithm, and **FP** for forward propagation (algorithm).

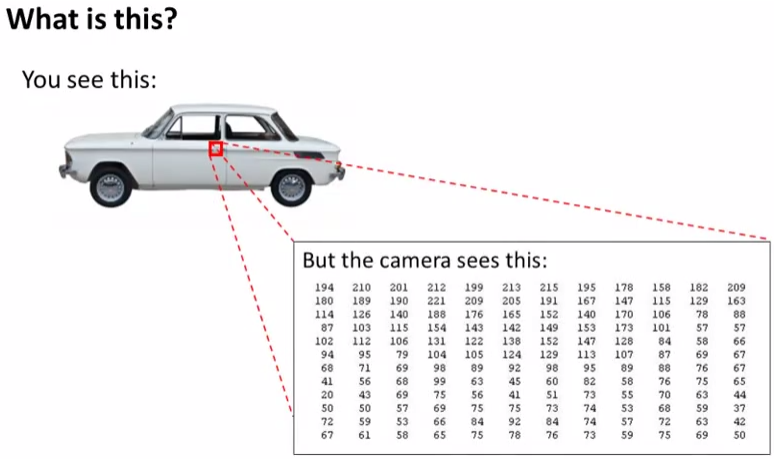


Figure 2: The image of a car and zoom of a spot enclosing the door handle.

x_1 = size
x_2 = # bedrooms
x_3 = # floors
x_4 = age
$\cdots$
x_{100}

For a problem like this, if you were to include all the quadratic terms there would be a lot of them. There would be terms like x_1^2, x_1x_2, x_1x_3, x_1x_4 up to x_1x_{100}, and then you would have x_2^2 squared, x_2x_3, and so on.

And if you include just the second order terms, for $n = 100$, you end up with about 5000 features. Asymptotically, the number of quadratic features grows roughly with $\mathcal{O}(n^2)$, where n is the number of original features $x_1, \cdots, x_{100}$. And it is actually close to $n^2/2$.

So, including all the quadratic features doesn't seem a good idea because that is a lot of features and you might overfit the training set, and it can also be computationally expensive to work with that many features.

One thing you could do is include only a subset of quadratic features, such as x_1^2, x_2^2, ... up to x_{100}^2, and then the number of quadratic features is much smaller: 100. But this is not enough features, and certainly it won't let you fit well enough the data set shown in Fig. 1.

In fact, if you include only quadratic features together with the original x_1, ..., x_{100} features, then you can actually fit interesting hypothesis, such as contours which are ellipses, but you certainly cannot fit complex data sets like those in Fig. 1.

So, 5000 features seems like a lot; but if you were to include cubic terms like $x_1x_2x_3$, $x_1^2x_2$, $x_{10}x_{11}x_{17}$, and so on, you can imagine there are a lot of these features. In fact, their number grows as $\mathcal{O}(n^3)$ and if $n = 100$ you end up with about 170000 such cubic features. So, including these higher order polynomial features when n is large dramatically blows up the feature space, and this doesn't seem like a good way to build classifiers when n is large.

For many machine learning problems, n will be pretty large. Let's consider the problem of computer vision where you want to use machine learning to train a classifier to examine an image and tell us whether or not the image is a car.

Many people wonder why computer vision could be difficult: I mean, when you and I look at the picture in Fig. 2, it is obvious what it is. You wonder how a learning algorithm could fail to know what the picture is.

To understand why computer vision is hard, let's zoom into a small part of the image where the little red rectangle is. It turns out that where you and I see a car, the computer sees a matrix, or grid, of pixel intensity values that tells the brightness of each pixel in the image. So, the computer vision problem is to look at the matrix of pixel intensity values and tell that those numbers represent the door handle of a car.

Concretely, when we use machine learning to build a car detector we come up with a label training set with a few label examples of cars and a few label examples of things that are not cars (Fig. 3). Then we give our training

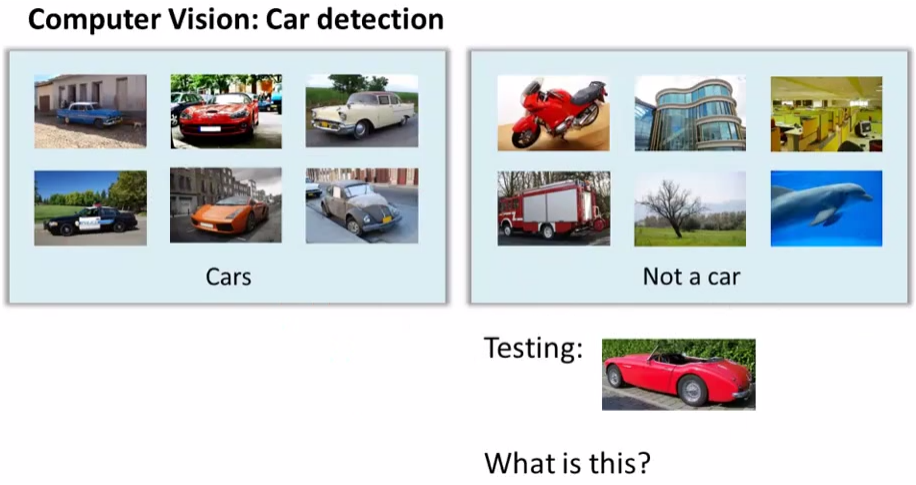


Figure 3: Training set for a classifier of car images. Some labels are cars, others are not cars.

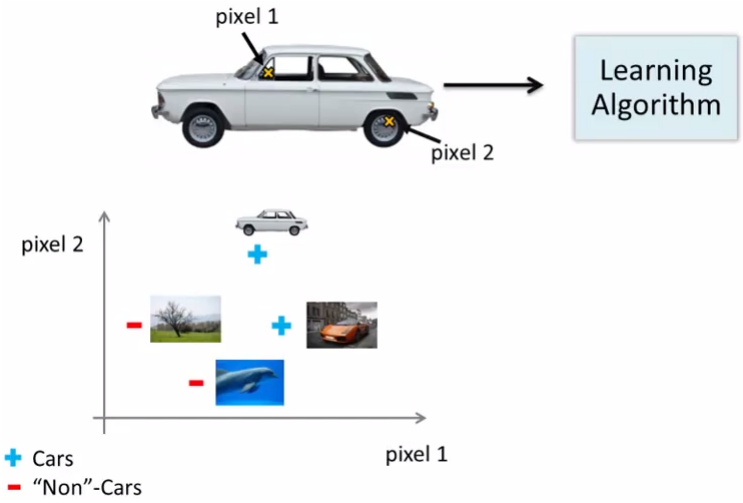


Figure 4: Choosing intensities of two pixels in the image as features of the classification problem.

set to the learning algorithm that trains the classifier and then, after the training is done, we may test the classifier by showing it a new image and asking "What is this new thing?".

And hopefully it will recognize that this new image is a car.

To understand why we need nonlinear hypothesis, let's take a look at some of the images of cars and maybe non-cars that we might feed to our learning algorithm.

Let's pick a couple of pixel locations in our images, and let's plot this car at a certain point, depending on the intensities of pixel 1 and pixel 2 (Fig. 4). And let's do this with a few other images. So let's take a different example of the car, look at the same two pixel locations and that image has a different intensity for pixel 1 and a different intensity for pixel 2. So, it ends up at a different location on the graphics picture.

And then let's plot some negative examples as well (dolphin, landscape in Fig. 4). If we do this for more and more examples using the (+) to denote cars and (-) to denote non-cars, the cars and non-cars end up lying in different regions of the space, and what we need is some sort of non-linear hypothesis to separate out the two classes (Fig. 5).

What is the dimension of the feature space? Suppose we use just 50×50 pixel images. Then we would have 2500

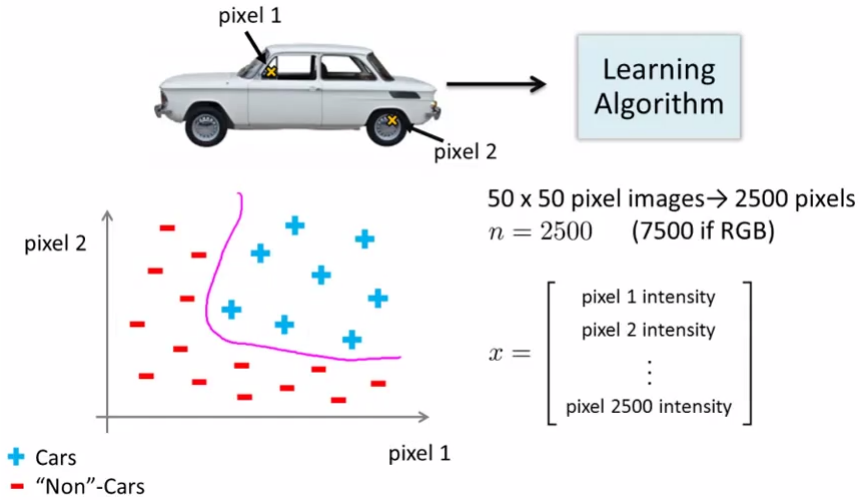


Figure 5: Small 50×50 image results in 2500 pixels, or features.

pixels, and so the dimension of our feature size will be $n = 2500$, not taking into consideration quadratic or cubic features. Our feature vector X is a list of all the pixel brightnesses, or intensities

$$X = \begin{bmatrix} \text{pixel 1 intensity} \\ \text{pixel 2 intensity} \\ \vdots \\ \text{pixel 2500 intensity} \end{bmatrix}$$

In a typical computer representation, each of these values may be between 0 and 255 if they are grayscale values. If we were using RGB images with separate red, green and blue values, what is equivalent to 3 grayscales per pixel, we would have $n = 7500$. If we try to learn a nonlinear hypothesis by including all the quadratic features $x_i \times x_j$, with $i, j \in [1, ..., 2500]$, we would end up with about 3 million features. And that's just too large to be reasonable; the computation to find and to represent all 3 million features per training example would be very expensive.

So, in conclusion, simple **logistic regression with quadratic or cubic features is not a good way to learn complex nonlinear hypotheses when n is large**, because there are too many features.

In the next videos I will present Neural Networks, which turns out to be a much better way to learn complex nonlinear hypothesis even when the input feature space n is large. And along the way I'll also show you a couple of fun videos of historically important applications of Neural Networks.

2 Neurons and the Brain

Neural Networks are a pretty old algorithm that was originally motivated by the goal of having machines that can mimic the brain.

In this class I'm teaching NNs because **they work really well for different machine learning problems**. In this video, I'd like to give you some of the background on NNs so that we can get a sense of what we can expect them to do, both in the sense of applying them to modern day machine learning problems, as well as, for those of you that might be interested in the big AI (Artificial Intelligence) dream of, someday, building truly intelligent machines, to show you how NNs might pertain to that.

The origins of NNs was as algorithms that try to mimic the brain in a sense that if we want to build learning systems, why not mimic the most amazing learning machine we know about, which is the brain?

NNs were very widely used throughout the 1980's and 1990's and, for various reasons, their popularity diminished in the late 90's. But more recently NNs have had a major resurgence. One of the reasons for this resurgence is that

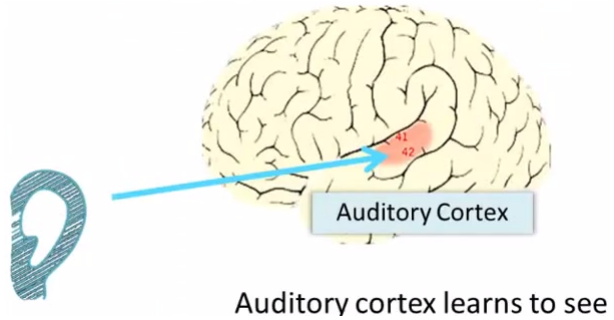


Figure 6: The "one learning algorithm" hypothesis.

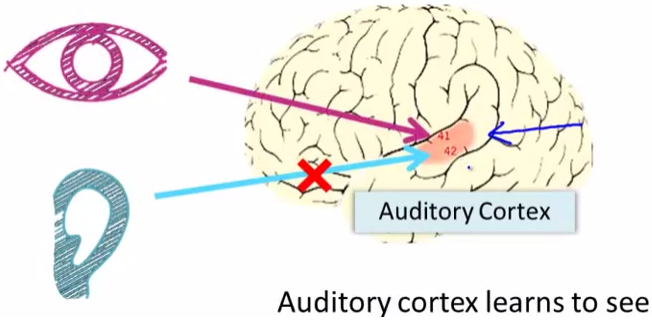


Figure 7: Auditory cortex learns to see: by rewiring vision signals to the auditory cortex it learns to see!

NNs are somewhat computationally expensive algorithms and so it was only recently that computers became fast enough to run really large scale NNs. Because of that, as well as because of a few other technical reasons which we'll talk about later, modern NNs are today the state of the art technique for many applications.

So, when you think about mimicking the brain, you realize that the human brain does amazing things, right? The brain can learn to see and process images, learn to hear, learn to process our sense of touch. We can learn to do math, learn to do calculus. And the brain does so many different and amazing things that it seems like if you want to mimic the brain you have to write lots of different pieces of software to mimic all of these different, fascinating, amazing things that the brain does.

But there is this fascinating hypothesis that *the way the brain does all of these different things is not worth like a thousand different programs, but instead, the way the brain does it is worth just a single learning algorithm.* This is just a hypothesis, but let me share with you some of the evidence for this.

That little red part of the brain in Fig. 6 is your auditory cortex. The way you're understanding my voice now is because your ear is taking the sound signal and routing that signal to your auditory cortex, and that's what's allowing you to understand my words.

Neuroscientists have done the following fascinating experiment where you cut the wire from the ears to the auditory cortex in an animal's brain and rewire so that the signal going from the eyes to the optic nerve eventually gets routed to the auditory cortex (Fig. 7). If you do this, it turns out the auditory cortex will learn to see – in every single sense of the word see, as we know it. The animals can then perform visual discrimination tasks; they can look at images and make appropriate decisions based on the images and they're doing it with that piece of brain tissue which is the auditory cortex, originally trained to hear.

Here's another example. That red piece of brain tissue in Fig. 8 is the somatosensory cortex, which process your sense of touch. If you do a similar re-wiring process, then the somatosensory cortex will learn to see (Fig. 9). Because of these and of other similar neuro-rewiring experiments, there's this sense that if the same piece of physical brain tissue can process sight, or sound or touch, then maybe there is one single learning algorithm that can process sight, or sound, or touch. And instead of needing to implement a thousand different programs or algorithms to do

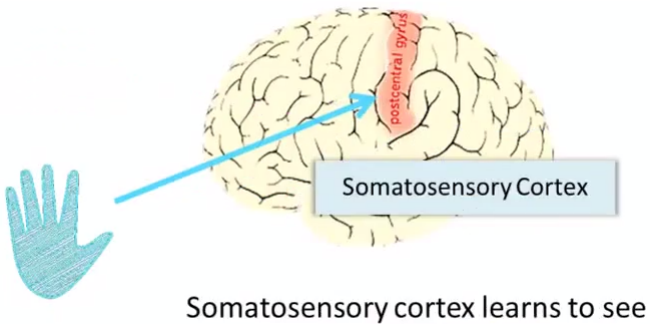


Figure 8: Somatosensory cortex learns to see.

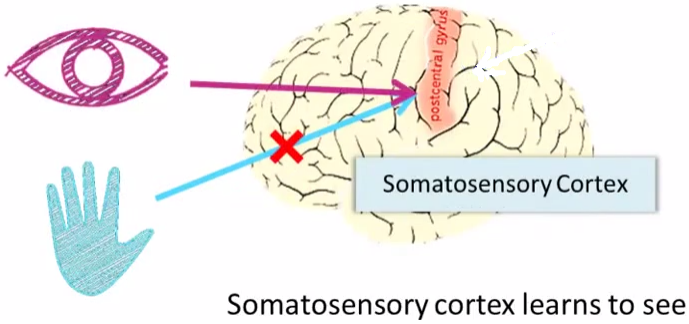


Figure 9: Rewiring vision to somatosensory cortex: it also learns to see.

the thousand wonderful things that the brain does, maybe what we need to do is just to figure out and **implement some approximation to whatever the brain's one and unique learning algorithm** is, and then just let the brain (or algorithm) learn by itself how to process these different types of data.

To a surprisingly large extent it seems that we can plug in almost any sensor to almost any part of the brain and so, by some reason, the brain will learn to deal with it.

Here are a few more examples.

On the upper left of Fig. 10 is an example of learning to see with your tongue. This is actually a system called *BrainPort* undergoing FDA trials now to help blind people see. The way it works is you strap a grayscale camera to your forehead, facing forward, that takes the low resolution grayscale image of what's in front of you and you then run a wire to an array of electrodes that you place on your tongue so that each pixel gets mapped to a "pixel" location on your tongue where, maybe, a high voltage corresponds to a dark pixel and a low voltage corresponds to a bright pixel. And, even as it stands today, with this sort of system you and I will be able to learn to see with our tongues in tens of minutes.

Here's a second example of human echo location, or human sonar, in the upper right of Fig. 10. So, there are two ways you can do this: you can either snap your fingers, or click your tongue (I can't do that very well). But there are blind people today that are actually being trained in schools to do this and learn to interpret the pattern of sounds bouncing off your environment - that's the principle of the sonar.

So, if you search on YouTube, there are actually videos of an amazing kid who, tragically, because of cancer, had his eyeballs removed. But by snapping his fingers he can walk around and never hit anything, he can ride a skateboard, he can shoot a basket ball into a hoop – and this is a kid with no eyeballs.

The third example is the *Haptic Belt* (lower left of Fig. 10) where you have a strap around your waist with a ring of buzzers and always have the northmost one buzzing. You can give a human a direction sense similar to that of birds which can sense where the north is.

And now a somewhat bizarre example: if you plug a third eye into a frog, the frog will learn to use that third eye as well (lower right of Fig. 10).

Figure 10: Sensor representation in the brain.

So, it's pretty amazing to what extent you can plug in almost any sensor to the brain and the brain's learning algorithm will just figure out how to learn from and deal with that data.

There's a sense that if we can figure out what the brain's learning algorithm is, and implement it or implement some approximation to it on a computer, maybe that would be our best shot at making real progress towards the Artificial Intelligence dream of someday building truly intelligent machines.

Of course I'm not teaching NNs just because they might give us a window into this far-off AI dream, even though that's one of the things that I personally work on in my research life. The main reason I'm teaching NNs in this class is because it's actually a very effective state of the art technique for modern day machine learning applications.

So, in the next few videos we'll start diving into the technical details of NNs so that you can apply them to modern-day machine learning applications and get them to work well on problems. But one of the reasons that excite me is that maybe NNs give us a window into what we might do if we're also thinking of what algorithms might someday be able to learn in a manner similar to humankind.

3 Model Representation I

I want to start telling you about how we represent Neural Networks (NNs), i.e., how we represent our hypotheses or our model when using NNs.

NNs were developed by simulating neurons, or networks of neurons, in the brain. So, to explain the hypotheses representation, let's start by looking at what a single neuron in the brain looks like (Fig. 11).

Your brain and mine is jam-packed full of **neurons**, which are cells in the brain. Two important things to draw attention to are: (i) the neuron has a *cell body* and has a number of input wires, called the ***dendrites***, which work as *input wires* and receive inputs from other locations. The neuron also has an *output wire* called the ***axon***, which sends signals or messages to other neurons.

So, at a simplistic level, a neuron is a computational unit that gets a number of inputs through its input wires, does some computation, and then sends outputs via its axon to other neurons in the brain.

Here's an illustration of a group of neurons (Fig. 12).

The way that neurons communicate with each other is with little pulses of electricity, also called spikes. So, when *Neuron 1* wants to send a message, it sends a little pulse of electricity via its *Axon 1* to some different neuron. The axon connects to the input wire, or dendrite, of this second neuron, *Neuron 2*, which then accepts this incoming message, does some computation, and may in turn decide to send out its own messages, through its own axon, to other neurons (green arrow).

And this is the process by which all human thought happens, neurons doing computations and passing messages to other neurons as a result of the inputs they've got.

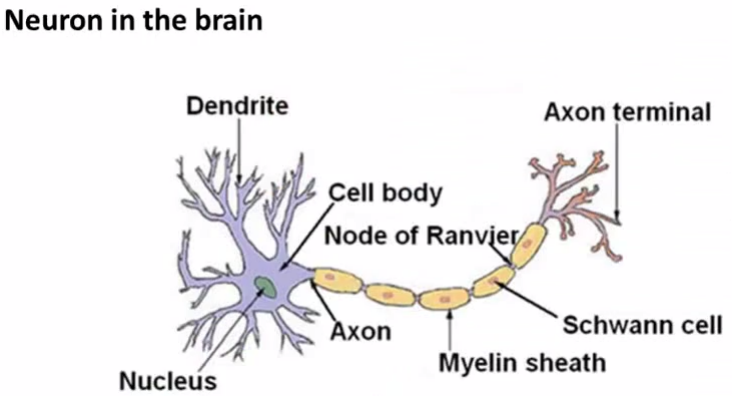


Figure 11: A single neuron.

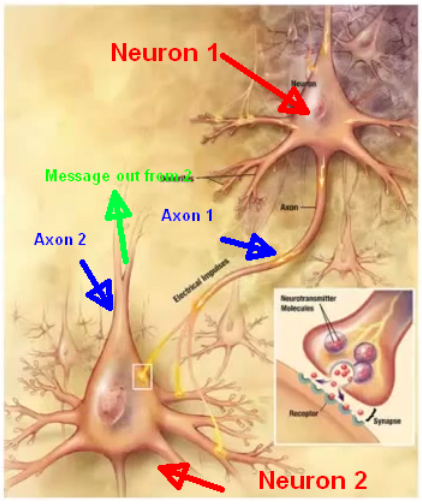


Figure 12: Net of neurons in the brain.

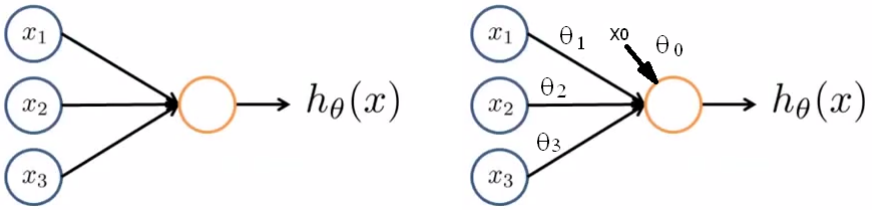


Figure 13: Neuron model: logistic unit. The **bias** input x_0, and the weights θ_i, are explicitely shown in the right.

And, by the way, this is how our senses and our muscles work as well. If you want to move one of your muscles, a neuron send pulses of electricity to your muscle and that causes your muscles to contract. And your eye – if some sensor like your eye wants to send a message to your brain, – what it does is it sends its pulses of electricity to a neuron in your brain.

In an artificial neural network implemented in a computer, we use a very simple model of a neuron: a **logistic unit** (Fig. 13).

The yellow circle in Fig. 13 plays a role analogous to the body of a neuron. When we feed inputs to the neuron via its dendrites, or input wires, the neuron does some computation and outputs some value on the output wire, a sort of a biological axon.

Each arrow linking an input x_i to the neuron, has attached a parameter, or weight, θ_i.

The diagram represents the **computation of a sigmoid fed with the dot product of** x **and** Θ that is

$$h_\Theta(x) = \frac{1}{1 + e^{-\Theta^T x}} \tag{1}$$

where, as usual, x and Θ are our parameter vectors

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad \Theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$$

So, this is a vastly oversimplified model of the computation that the biological neuron does, where it gets a certain number of inputs, x_1, x_2, x_3, and it outputs some value $h_\theta(x)$ computed like (1).

When I draw a neural network, usually I draw only the inputs x_1, x_2, x_3; sometimes, I draw an extra node for x_0. This x_0 node is called the **bias unit** or the **bias neuron** and $x_0 = 1$, always. Sometimes I draw it, sometimes I won't, just depending on whether it's more notationally convenient for the example.

Finally, one last bit of terminology in neural networks. The **sigmoid**

$$g(z) = \frac{1}{1 + e^{-z}}$$

is also called a (logistic) **activation function**.

I've been calling θ the **parameters**, but in the neural networks literature sometimes they are called **weights** of a model; it just means exactly the same thing as parameters of the model.

The diagram in Fig. 13 represents a single neuron. A **Neural Network** is just a group of different neurons strung together.

Concretely, in Fig. 14 we have the input units x_1, x_2 and x_3 and, once again, sometimes we draw the extra (bias) node x_0 linked to all the three neurons in layer 2.

And we have three neurons called $a_1^{(2)}$, $a_2^{(2)}$ and $a_3^{(2)}$ (we'll discuss the indices later) and, once again, we can add $a_0^{(2)}$ as an extra bias to layer 2, which is feeding the output node in layer 3, and $a_0^{(2)}$ always outputs the value of 1.

Then, at last, we have the final layer 3, which is the fourth node in our model that outputs the value of the hypotheses that $h_\Theta(x)$ computes.

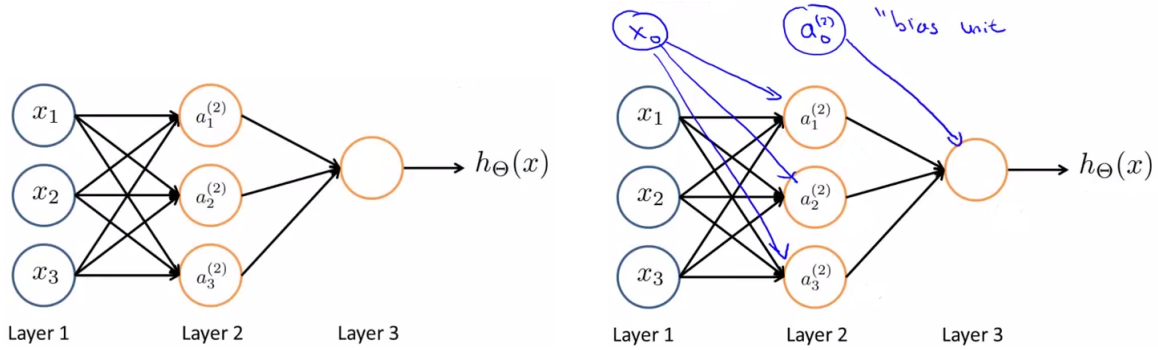


Figure 14: Neural Network with 3 layers. In the right picture the bias units are shown.

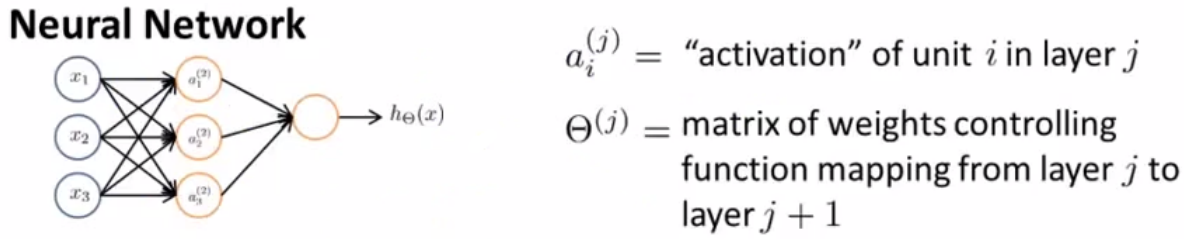


Figure 15: Notation used in NNs.

To introduce a bit more NN terminology, the first layer (layer 1) is also called the ***input layer*** because there we input our features, x_1, x_2 and x_3. The final layer (layer 3) is also called the ***output layer*** because it has the neuron that outputs the final value computed by a hypothesis $h_\Theta(x)$. The layer in between (layer 2) is called the ***hidden layer***; it isn't a great terminology, but the intuition is that in supervised learning the hidden layer values aren't observed in the training set.

If a layer it's not x and it's not y, we call those the hidden layers. And later on we'll see NNs with more than one hidden layer. Basically, anything that isn't an input layer or a output layer is called a hidden layer.

So, to be clear about what this neural network is doing, let's step through the computational steps embodied in this diagram. To explain the specific computations represented by a neural network, here's a little bit more notation.

I'm going to use $a_i^{(j)}$ to denote the activation of neuron (or unit) i in layer (j). So, concretely, $a_1^{(2)}$ does the activation of the first unit in layer 2, our hidden layer. And by activation, I just mean the value that is computed by a specific sigmoid.

In addition, our NN is parameterized by the matrices $\Theta^{(j)}$; each $\Theta^{(j)}$ is a matrix of weights controlling the function mapping from layer j to the next layer, layer $j+1$.

So, here are the computations represented by the diagram in Figs. 14 and 15.

$$
\begin{aligned}
a_1^{(2)} &= g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3) \\
a_2^{(2)} &= g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3) \\
a_3^{(2)} &= g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3) \\
h_\Theta(x) = a_1^{(3)} &= g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)})
\end{aligned}
$$

This first hidden unit, $a_1^{(2)}$, has its value computed as

$$
a_1^{(2)} = g\left(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3\right)
$$

where $g(\cdots)$ is the sigmoid or logistic activation function, applied to a linear combination of its inputs. And so on, for the other $a_i^{(2)}$ activation units.

So, we have 3 input units and 3 hidden units. And so $\Theta^{(1)}$, the matrix of parameters governing our mapping from the 3 input units, x_1, x_2 and x_3, plus the bias x_0, in layer 1, to the 3 hidden units in layer 2, is going to be a matrix with dimensions 3×4:

$$\Theta^{(1)} = \begin{bmatrix} \Theta_{10}^{(1)} & \Theta_{11}^{(1)} & \Theta_{12}^{(1)} & \Theta_{13}^{(1)} \\ \Theta_{20}^{(1)} & \Theta_{21}^{(1)} & \Theta_{22}^{(1)} & \Theta_{23}^{(1)} \\ \Theta_{30}^{(1)} & \Theta_{31}^{(1)} & \Theta_{32}^{(1)} & \Theta_{33}^{(1)} \end{bmatrix} \tag{2}$$

More generally, if a network has s_j units in layer j and s_{j+1} units in layer $j+1$ then the matrix $\Theta^{(j)}$ which governs the function mapping from layer j to layer $j+1$ will be $s_{j+1} \times (s_j + 1)$.

For $\Theta^{(1)}$ we have $s_1 = 3$ and $s_2 = 3$, thus getting a 3×4 matrix, which is $\Theta^{(1)}$.

So, we talked about the computations in the 3 hidden units. Finally, in the final layer (layer 3) we have one more unit, $h_\theta(x)$, which can also be written as $a_1^{(3)}$, calculated as

$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)})$$

where $\Theta^{(2)}$ is the 1×4 matrix of parameters, or weights, that maps from the 3 hidden layer units to the single unit in layer 3 , that is, the output unit:

$$\Theta^{(2)} = \begin{bmatrix} \Theta_{10}^{(2)} & \Theta_{11}^{(2)} & \Theta_{12}^{(2)} & \Theta_{13}^{(2)} \end{bmatrix}$$

To summarize, we've shown how a picture (Fig. 14) represents an artificial NN which defines a function $h_\Theta(x)$ mapping the input values x to some space of predictions, y. And the hypothesis $h()$ are parametrized by weights in $\Theta^{(l)}$, so that by varying the parameters $\Theta^{(l)}$ we get different hypothesis.

So that was a mathematical definition of how to represent the hypothesis in the NN. In the next few videos I give you more intuition about what these hypothesis representations do, I go through a few examples and talk about how to do efficient computations.

4 Model Representation II

In the last video we gave a mathematical definition of how to compute the hypothesis used by NNs. Now I show you how to actually carry out that computation efficiently using a vectorized implementation. And second, more importantly, I want to start giving you intuition about why these NN representation might be a good idea and how they can help us to learn complex nonlinear hypothesis.

Consider the neural network in Fig. 16. The sequence of steps that we need in order to compute the output of a hypothesis are the equations given in that figure, where we compute the activation values of the three hidden units $a_i^{(2)}$, which are used to compute the final output of our hypothesis, $h_\Theta(x)$.

I'm going to define a few extra terms. So,

$$z_1^{(2)} = \Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3$$

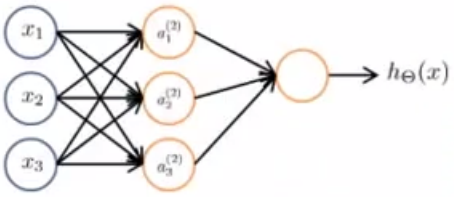
such that

$$a_1^{(2)} = g\left(z_1^{(2)}\right)$$

The superscript (2) means that these are values associated with layer 2, which is the hidden layer in our NN.

In the same way we could define $z_2^{(2)}$ and $z_3^{(2)}$ as the arguments of $g()$ in the definitions of the $a_2^{(2)}$ and $a_3^{(2)}$ activation values in Fig. 16

$$\begin{aligned} z_2^{(2)} &= \Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3 \\ z_3^{(2)} &= \Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3 \end{aligned}$$



$$a_1^{(2)} = g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3)$$
$$a_2^{(2)} = g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3)$$
$$a_3^{(2)} = g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3)$$
$$h_\Theta(x) = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)})$$

Figure 16: Forward propagation: vectorized implementation.

and accordingly

$$\begin{aligned} a_2^{(2)} &= g\left(z_2^{(2)}\right) \\ a_3^{(2)} &= g\left(z_3^{(2)}\right) \end{aligned}$$

So these $z_i^{(2)}$ values are just a weighted linear combination of the input values x_0, x_1, x_2 and x_3 that go into a particular neuron.

Now you may notice that the calculation of the $z_i^{(2)}$ corresponds to the matrix-vector multiplication

$$z^{(2)} = \Theta^{(1)}\, x$$

with

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad z^{(2)} = \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{bmatrix}$$

where x_0 as usual is always equal to 1, and $\Theta^{(1)}$ is given in (2).

Using this observation we're going to be able to vectorize this computation of the NN. Notice that $z^{(2)}$ is a 3-D vector $\in \mathbb{R}^3$. We can now vectorize the computation of $a^{(2)}$ in two steps

$$\begin{aligned} z^{(2)} &= \Theta^{(1)}\, x \\ a^{(2)} &= g\left(z^{(2)}\right) \end{aligned}$$

Both $a^{(2)}$ and $z^{(2)}$ are 3-D vectors and the activation $g()$ applies the sigmoid function element-wise to each of the $z_i^{(2)}$ elements of $z^{(2)}$. To make our notation more consistent with what we'll do later, we can say that in the input layer

$$a^{(1)} = x$$

now we can rewrite the former expressions of $a^{(2)}$ and $z^{(2)}$:

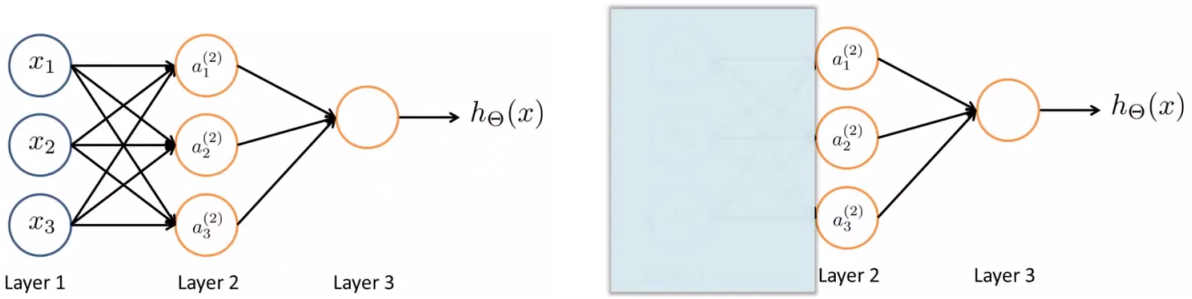


Figure 17: Neural Network learning its own features. In the right the input layer is covered, and that looks like logistic regression.

$$\begin{aligned} z^{(2)} &= \Theta^{(1)} a^{(1)} \\ a^{(2)} &= g\left(z^{(2)}\right) \end{aligned}$$

In $a^{(2)}$ we need one more value, $a_0^{(2)}$, that corresponds to a bias unit in the hidden layer. Of course, there was also a bias unit in the input, $x_0 = 1$, which I just didn't draw. So, after adding an extra equation $a_0^{(2)} = 1$, then $a^{(2)}$ is going to be a 4-D feature vector.

Finally, to compute the actual output value of our hypothesis, we then simply need to compute $z^{(3)}$, the activation of the one and only unit in the output layer. Recall that

$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)})$$

and using the definition of the $z_i^{(j)}$ and the new vectorized notation we just introduced, we can write

$$\begin{aligned} z^{(3)} &= \Theta^{(2)} a^{(2)} \\ h_\Theta(x) = a^{(3)} &= g\left(z^{(3)}\right) \end{aligned}$$

So, $h_\Theta(x) = a^{(3)} \equiv a_1^{(3)}$ is just a real number here.

This process of computing $h_\Theta(x)$ is also called **forward propagation**, because we start with the activations of the input units and then we sort of forward-propagate them to the hidden layer, and so on and so forth we compute the activations of the hidden layer, and then we forward-propagate that and compute the activations of the output layer. If there were more hidden layers in the NN, they also would have been included in the cascaded forward propagation activity.

We just worked out a vectorized implementation of forward propagation, which gives us an efficient way of computing $h_\Theta(x)$.

Forward propagation also helps us to understand why NNs might help us to learn interesting nonlinear hypothesis.

Consider the NN in Fig. 17 and let's say I cover up the left part corresponding to the input layer. If you look at what's left, it looks a lot like logistic regression, where we are just using the output node as a logistic regression unit to make a prediction $h_\Theta(x)$ such that

$$h_\Theta(x) = g(\Theta_0 a_0 + \Theta_1 a_1 + \Theta_2 a_2 + \Theta_3 a_3) \tag{3}$$

where $a_0 = 1$ is the bias input in layer 2.

Now, to be consistent to the early notation, we need to fill in these superscripts (2) into the $a_i^{(2)}$ and $\Theta_{ij}^{(2)}$

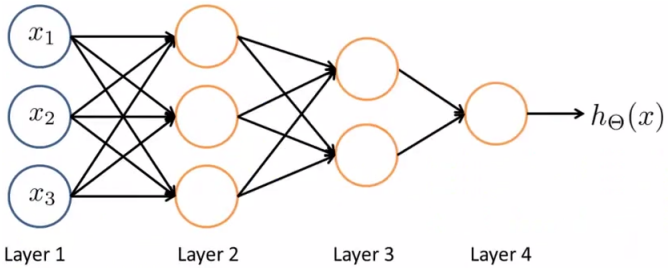


Figure 18: Another neural network architecture.

$$h_\Theta(x) = g\left(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)}\right)$$

but if you focus on the notation given in (3) it looks awfully like the standard logistic regression model, except that we now have Θ instead of θ.

And what this NN is doing is just logistic regression, where the features fed into it are these values computed by the hidden layer, the $a_i^{(2)}$, rather than using the original features x_1, x_2, x_3.

And the cool thing about this, is that the features $a_1^{(2)}$, $a_2^{(2)}$ and $a_3^{(2)}$ they themselves are learned as functions of the input (see Fig. 17, left): the function mapping from layer 1 to layer 2 is determined by some other set of parameters, engraved in the matrix $\Theta^{(1)}$.

So, it's as if the NN instead of being constrained to feed the features x_1, x_2, x_3 to logistic regression, it has first to learn its own features $a_1^{(2)}$, $a_2^{(2)}$ and $a_3^{(2)}$, to then feed them into the logistic regression, and as you can imagine that depending on what parameters it chooses for $\Theta^{(1)}$.

You can learn some pretty interesting and complex features and, therefore, you can end up with a better hypothesis than if you were constrained to use simple logistic regression with the raw features x_1, x_2, x_3 or even with the polynomial terms x_1x_2, x_2x_3, ... and so on. But instead this algorithm has the flexibility to try to learn several features at once, using the $a_1^{(2)}$, $a_2^{(2)}$ and $a_3^{(2)}$, in order to feed the last unit that's essentially a logistic regression calculating $h_\Theta(x)$.

I realize the example I described has a somewhat high level, and so I'm not sure if this intuition of the NN having the potential to cope with more complex features will quite make sense to you. But if it doesn't, in the next videos I'm going to go through a specific example of how a neural network can use the hidden layer (layer 2) to compute more complex features to feed into the final output layer (layer 3) and how that process can learn more complex hypothesis.

So, in case what I'm saying here doesn't quite make sense, stick with me for the next two videos and hopefully there, working through those examples, this explanation will make a little bit more sense.

You can have neural networks with other types of diagrams as well, and the way that neural networks are connected is called the architecture. So, the term **architecture** refers to how the different neurons are connected to each other. In Fig. 18 is an example of a different neural network architecture: you may be able to get the intuition of how layer 2 has 3 units that are computing maybe some complex function of the input layer; and then layer 3 can take the second layer's features and compute even more complex features, so that by the time you get to the output layer, layer 4, you can have even more complex features of what you are able to compute in layer 3 and so you can end up with very interesting nonlinear hypothesis.

By the way, in the NN shown in Fig. 18 layer 1 is called an input layer, layer 4 is still the output layer, and this NN has two hidden layers – layers 2 and 3. So, anything that's not an input layer or an output layer is called a hidden layer, as we've said before.

So, hopefully from this video you've got a sense of how the feed-forward propagation step in a NN works; you start from the activations of the input layer and forward propagate those to the first hidden layer, and then to the second hidden layer, and then finally the output layer. And you also saw how we can vectorize the computations associated to feed forward propagation.

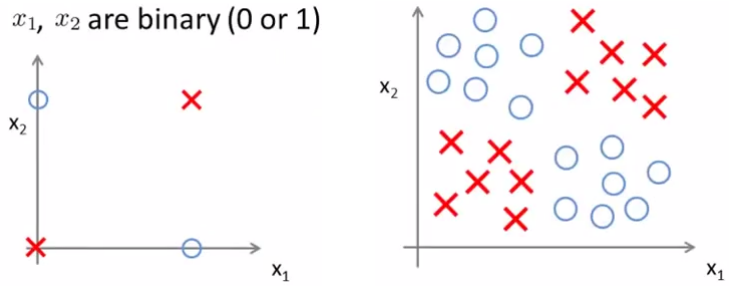


Figure 19: Non-linear classification example: logical function XNOR. At right, a generalization of outputs for a XNOR classification problem is presented.

I realized that some of the intuitions conveyed in this video, of how certain layers are computing complex features of the early layers, may be still slightly abstract and kind of a high level. And so what I would like to do in the next two videos is work through a detailed example of how a neural network can be used to compute nonlinear functions of the input, and hope that it will give you a good sense of the sorts of complex nonlinear hypothesis we can get out of NNs.

5 Examples and Intuitions I

In this and in the next video, I want to work through a detailed example, showing how a neural network can compute a complex nonlinear function of the input and, hopefully, this will give you a good sense of why NNs can be used to learn complex nonlinear hypothesis.

Consider the following problem (Fig. 19) where we have two binary input features, x_1 and x_2, so their values are either 0 or 1. So x_1 and x_2 can each take on only one of two possible values, 0 or 1.

I've drawn only two positive examples and two negative examples in Fig. 19 left, but you can think of it as a simplified version of a more complex learning problem where we may have a bunch of positive examples in the upper right and the lower left of the 2-dimensional x_1 vs. x_2 diagram, and a bunch of negative examples notified by the circles (see Fig. 19 right).

What we'd like to do is learn a nonlinear decision boundary that separates the positive and the negative examples.

So how can a NN do this? And rather than use the example on the right of Fig. 19, I'm going to use the maybe easier to examine example on the left, which concretely is really computing the target label

$$\begin{aligned} y &= x_1 \text{ XNOR } x_2 \\ &= \text{NOT } (x_1 \text{ XOR } x_2) \end{aligned}$$

where XNOR is the alternative notation for NOT (x_1 XOR x_2). So, (x_1 XOR x_2) is true only if *exactly one* of x_1 or x_2 is equal to 1; thus, the XNOR example is true only if $x_1 = x_2$, that is, we're going to have positive examples if either both are true (i.e. equal to 1) or both are false (i.e. equal to 0). And we want to figure out if we can get a NN to fit to this sort of training set.

In order to build up to a network that fits the XNOR example, we're going to start with a slightly simpler one and show a network that fits the AND function (Fig. 20). The inputs x_1 and x_2 are again binary, so they are either 0 or 1. And our target label is $y = x_1$ AND x_2, the logical AND function.

Can we get a one-unit NN to compute this logical AND function? I'm going to actually draw in the bias unit, the +1 unit (which means I have a $x_0 = 1$), and assign values to the weights, or parameters, of the network (Fig. 20 right). So, concretely, this is saying that my hypothesis is

$$y = h_\Theta(x) = g\left(-30 + 20x_1 + 20x_2\right) \tag{4}$$

Sometimes is convenient to draw the weights in the diagram of the neural network: but their meaning in terms of

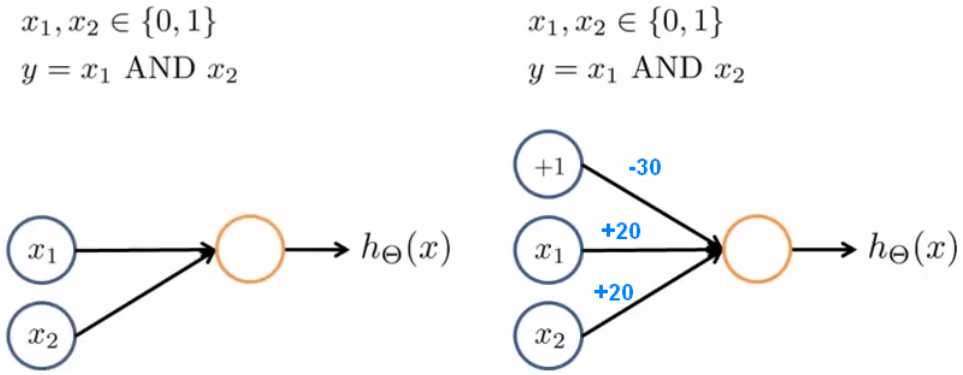


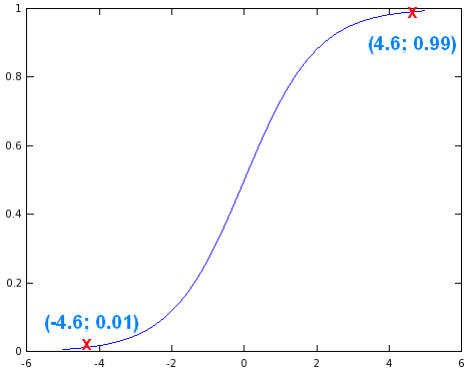
Figure 20: Simple network which implements the AND function. The bias unit and the weights are included in the complete NN shown in the right.

the $\Theta^{(l)}$ matrices of parameters is just

$$\Theta^{(1)} = \begin{bmatrix} \Theta_{10}^{(1)} & \Theta_{11}^{(1)} & \Theta_{12}^{(1)} \end{bmatrix} = \begin{bmatrix} -30 & +20 & +20 \end{bmatrix}$$

Let's look at what this little single neuron network in Fig. 20 will compute. Just to remind you, the sigmoid activation function expression and graphics are

$$g(z) = \frac{1}{1 + e^{-z}}$$



And to give you some landmarks, if $z = 4.6$ then $g(z) = 0.99$, very close to 1.0, and kind of symmetrically if $z = -4.6$, then $g(z) = 0.01$, which is very close to 0.

Let's look at the four possible input values for the pair (x_1,x_2) and how the hypothesis (4) will be calculate in those cases.

If $x_1 = x_2 = 0$, then the hypotheses is $y = g(-30)$, very far to the left in the diagram of $g(z)$ and thus $g(-30)$ is very close to 0.

If $x_1 = 0$ and $x_2 = 1$, then $y = g(-10)$ and that's again very close to 0. It is also $y = g(-10) \approx 0$ when $x_1 = 1$ and $x_2 = 0$.

And, finally, if $x_1 = x_2 = 1$ then you have $y = g(-30 + 20 + 20)$, so that's $g(+10)$, which is therefore very close to 1.0. Thus we get the following table:

x_1	x_2	$h_\Theta(x)$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(+10) \approx 1$

And if you look in the right column of the table, it is exactly the logical AND function, that is

$$h_\Theta(x) \approx x_1 \text{ AND } x_2$$

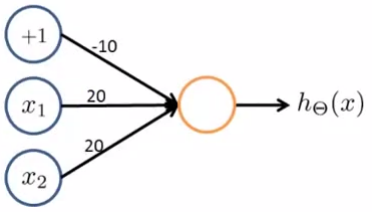
In other words, $h_\Theta(x)$ outputs 1 if and only if x_1 and x_2 are both equal to 1.

So, by writing out our truth table we manage to figure out what's the logical function that our NN computes.

The network in Fig. 21 computes the OR function. The hypothesis is

$$y = h_\Theta(x) = g\left(-10 + 20x_1 + 20x_2\right)$$

and so, if you fill in the truth table (shown also in figure 21) you find that the numbers in the right column are essentially the x_1 OR x_2 logical function.



x_1	x_2	$h_\Theta(x)$
0	0	$g(-10) \approx 0$
0	1	$g(+10) \approx 1$
1	0	$g(+10) \approx 1$
1	1	$g(+40) \approx 1$

Figure 21: Neural network which implements the OR function and its truth table.

So, hopefully you now understand how single neurons in a NN can be used to compute logical functions like AND, OR, and so on. In the next video we'll continue building on these examples and work through a more complex example. We'll show you how a NN with multiple layers of neuron units can be used to compute more complex functions like the XOR function or the XNOR function.

6 Examples and Intuitions II

In this video I'd like to show how a neural network can compute complex nonlinear hypothesis. In the last video we saw how a neural network can be used to compute the functions "x_1 AND x_2" and the function "x_1 OR x_2" when x_1 and x_2 are binary, that is, when they take on values of 0 or 1.

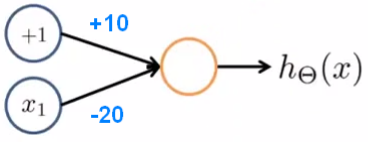
We can also have a network to compute negation (Fig. 22), that is, to compute the function "NOT x_1".

Let me just write down the weights associated with this network. We have only one input feature, x_1, and the bias unit with value +1, and if I associate the weights +10 with the bias unit and -20 with x_1 then my hypothesis is

$$h_\Theta(x) = g\left(10 - 20x_1\right)$$

and so, when $x_1 = 0$ my hypothesis will be computing $g(10) \approx 1$; and when $x_1 = 1$ my hypothesis will be computing $g(-10) \approx 0$. That's essentially the "NOT x_1" function.

So, to include negations the general idea is to put a large negative weight, -20, in front of the variable x_1 you want to negate.



x_1	$h_\Theta(x)$
0	g(10) ~ 1
1	g(-10) ~ 0

Figure 22: Neural network for negation of x_1 (or NOT x_1) and truth table of the NOT function.

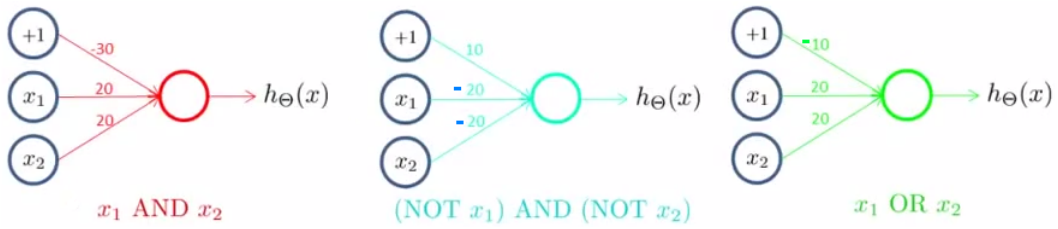
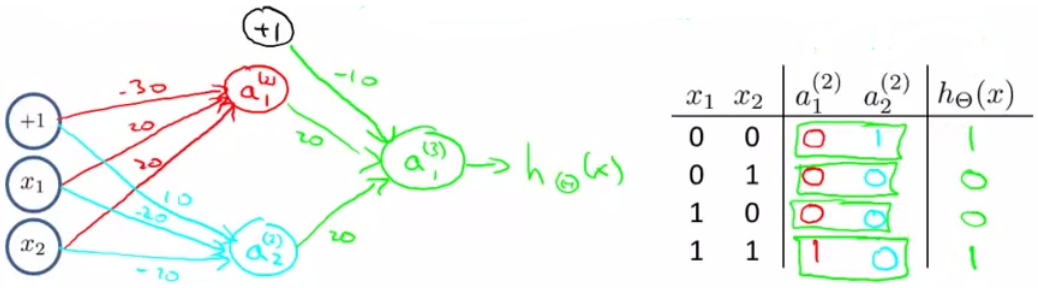


Figure 23: Three NNs for the AND, NOT-AND-NOT and OR logical functions.



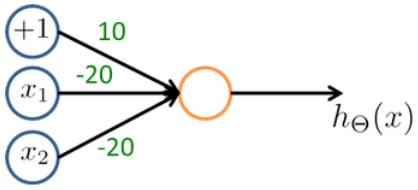
x_1	x_2	$a_1^{(2)}$	$a_2^{(2)}$	$h_\Theta(x)$
0	0	0	1	1
0	1	0	0	0
1	0	0	0	0
1	1	1	0	1

Figure 24: NN for implementation of the XNOR function.

In an example that I hope you will figure out yourself, if you want to compute "(NOT x_1) AND (NOT x_2)" probably you would be putting large negative weights in front of x_1 and x_2, and it should be feasible to get a neural network with just one output unit to compute this.

So, "(NOT x_1) AND (NOT x_2)" is going to be equal to 1 iff (i.e., if and only if) $x_1 = x_2 = 0$. Both operands of the AND must be 1, thus x_1 and x_2 both have to be 0.

Hopefully you should be able to figure out how to make a small neural network to compute this logical function: it is represented below.



Now, taking the three NNs for implementing logical functions that we have put together – the NNs for computing "x_1 AND x_2", "(NOT x_1) AND (NOT x_2)" and "x_1 OR x_2", – shown in Fig. 23, we should be able to put these three pieces together to compute the "x_1 XNOR x_2" function.

The "x_1 XNOR x_2" classification diagram was shown in Fig. 19. Clearly we'll need a nonlinear decision boundary in order to separate the positive and negative examples.

Let's draw the network (Fig. 24).

The first hidden unit, $a_1^{(2)}$, implements the AND function (in red). The second hidden unit, $a_2^{(2)}$, in cyan, implements the "(NOT x_1) AND (NOT x_2)" function. Thus we can fill the columns $a_1^{(2)}$ and $a_2^{(2)}$ in the truth table.

Finally, my output unit, that is $a_1^{(3)} = h_\Theta(x)$, uses the OR network (in green). I insert a +1 bias unit and copy the weights from the green NN. We know that this computes the OR function. So, let's go on to fill the truth table entries below $h_\Theta(x)$, which is "$a_1^{(2)}$ OR $a_2^{(2)}$", giving 1, 0, 0, and 1 and this is indeed our definition of the XNOR function.

And concretely, with this neural network, which has an input layer, one hidden layer and one output layer, we end up with a nonlinear decision boundary that computes this XNOR function.

And the more general intuition about the NN is that in the input layer we just have our raw inputs, then the

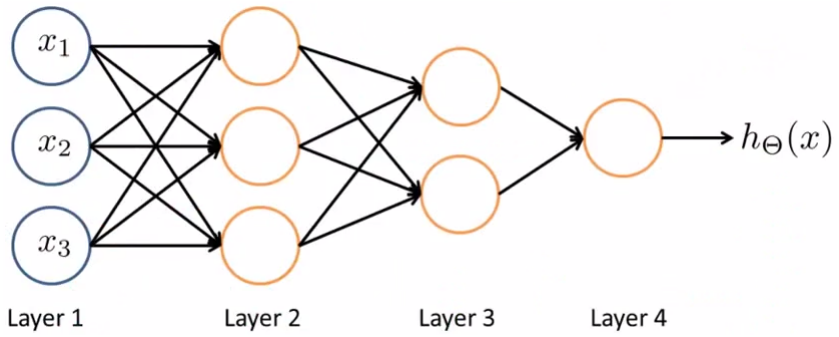


Figure 25: Neural network intuition.

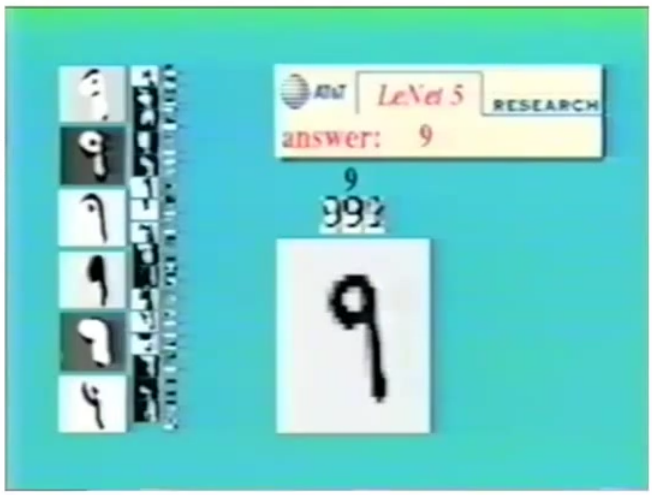


Figure 26: Film showing a NN performing very nice recognition of handwritten digits.

hidden layer computes some slightly more complex functions of the inputs, and then by adding yet another layer, the output layer, we end up with an even more complex nonlinear function.

And this is the sort of intuition about why NNs can compute pretty complicated functions (Fig. 25). When you have multiple layers, you start with relatively simple function of the inputs in the second layer, but the third layer can build on that to compute even more complex functions, and then the layer after that can compute even more complex functions and so on and so forth.

To wrap up this video, I want to show you a fun example of an application of a NN that captures this intuition of the deeper layers computing more complex features. I want to show you a video that I got from a good friend of mine, Yann LeCun, a professor at New York University, and he was one of the early pioneers of NN research and a sort of a legend in the field now. His ideas are used in all sorts of products and applications throughout the world.

So, I want to show you a video from some of his early work in which he was using a NN to do handwritten digit recognition. You might remember that at the start of this class I said that one of the early successes of NNs was trying to read postal zip codes, to help us send mail.

So, this is one of the algorithms used to try to address that problem. The lower center white box shows the handwritten character input to the network. The left column shows a visualization of the features computed by hidden layers of the network: it shows different features, different edges and lines, and so on, detected in the input digit. The visualization of the second hidden layer it's kind of harder to see, harder to understand deeper hidden layers. You probably have a hard time seeing what's going on, you know, much beyond the first hidden layer.

But then finally, all of the learned features get fed to the output layer and shown on top is the final or predicted

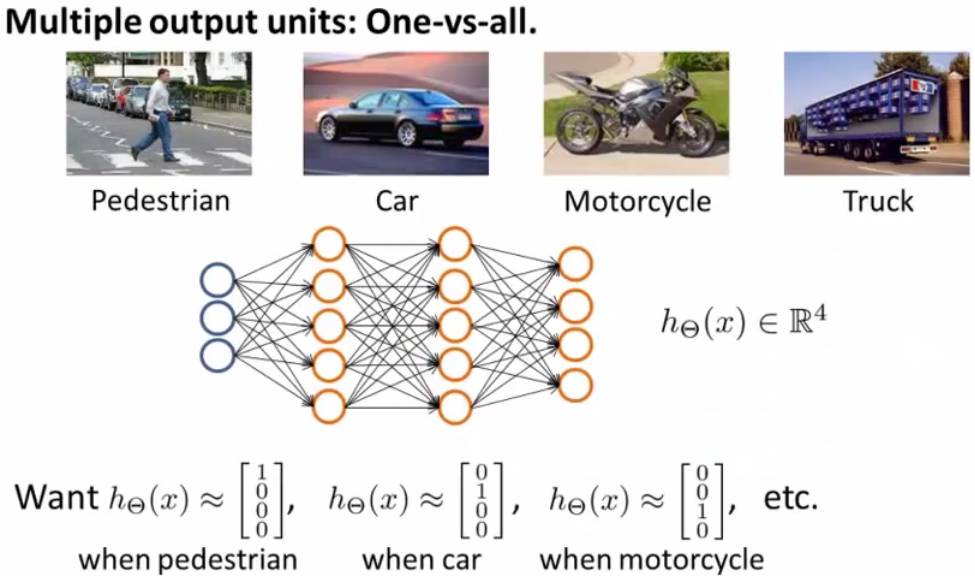


Figure 27: Multiclass classification problem solved with NNs.

answer for what handwritten digit the neural network thinks that is being shown.

So, let's take a look at the video. (Funny Medieval Music!)

So, I hope you enjoyed the video and that this hopefully gave you some intuition about the sorts of pretty complicated functions that NNs can learn. Here it takes as input the image, just takes as input the raw pixels, and the first hidden layer computes some set of features, the next hidden layer computes even more complex features, and these features can then be used by, essentially, the final layer of logistic regression classifiers to make accurate predictions about what are the numbers that the neural network sees in its inputs.

7 Multiclass Classification

In this video, I want to tell you about how to use neural networks to do multiclass classification. We have more than one category that we're trying to distinguish amongst. In the last video, we saw a movie about the handwritten digit recognition problem, that was actually a multiclass classification problem because there were ten possible categories for recognizing the digits from 0 through 9. And so, now I want to fill on the details of how to do that.

The way we do multiclass classification in a neural network is essentially an extension of the one versus all method.

So, let's say that we have a computer vision example, where instead of just trying to recognize cars, as in the original example that I started off with, we're trying to recognize four categories of objects. Given an image we want to decide if it is a pedestrian, a car, a motorcycle or a truck (Fig. 27).

What we do is build a NN with four output units, so that our NN now outputs a vector of four numbers.

And what we're going to try to do is to get the first output unit to classify if the image is a pedestrian, yes or no. The second unit to classify if the image is a car, yes or no. The third unit to classify if the image is a motorcycle, yes or no. And, finally, the fourth unit would classify if the image is a truck, yes or no.

And thus, when the image is of a pedestrian, we would ideally want the network to output

$$h_\Theta(x) \approx \begin{bmatrix}1\\0\\0\\0\end{bmatrix}$$

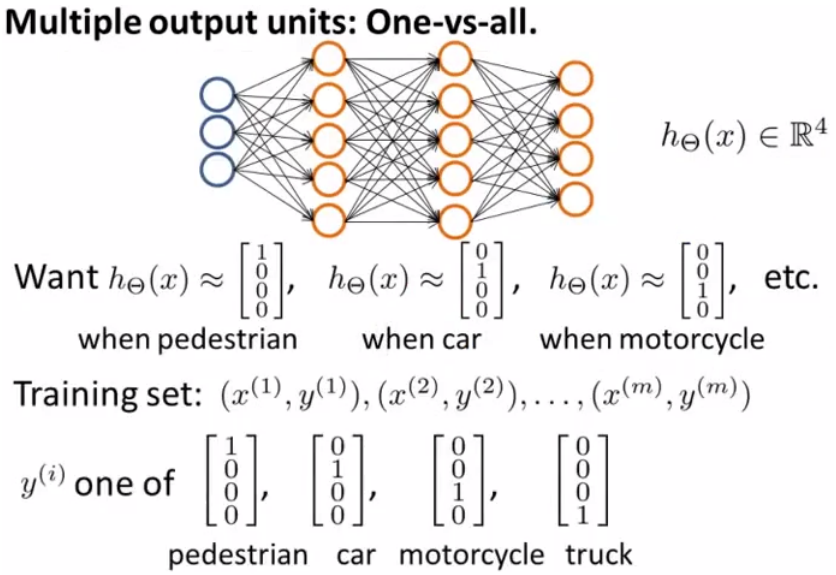


Figure 28: Multiclass classification: presenting the training set.

and when it is a car we want it to output $[0, 1, 0, 0]^T$, and when it is a motorcycle we want it to output $[0, 0, 1, 0]^T$, and so on.

So, this is just like the "one versus all" method that we talked about when we were describing logistic regression, and here we have essentially four logistic regression classifiers, each of which is trying to recognize one of the four classes that we want to distinguish amongst.

So, in Fig. 28 is our NN with four output units, and those vectors are what we want $h_\Theta(x)$ to be when we have the different images.

The way we're going to represent the training set in these settings is as follows.

When we have a training set with different images of pedestrians, cars, motorcycles and trucks, whereas previously we had written out the labels as $y^{(i)}$ being an integer in the set $\{1, 2, 3, 4\}$, now we're going to instead represent $y^{(i)}$ as one element of the set of four vectors

$$y^{(i)} \in \left\{ \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix} \right\}$$

depending on what the corresponding image $x^{(i)}$ is, or to what class it belongs.

And so one training example will be one pair $(x^{(i)}, y^{(i)})$, where $x^{(i)}$ is an image with one of the four objects and $y^{(i)}$ will be one of those four vectors.

And, hopefully, we can find a way to get our NN to output some values such that

$$h_\Theta(x^{(i)}) \approx y^{(i)}$$

both $h_\Theta(x^{(i)})$ and $y^{(i)}$ are going to be 4-D vectors, i.e. $\in \mathbb{R}^4$ in our example, because we have four classes.

So, that's how you get a NN to do multiclass classification. This wraps up our discussion on how to represent NNs, that is, on our hypothesis representation. In the next set of videos we'll start to talk about how to take a training set and how to automatically learn the parameters of the NN.

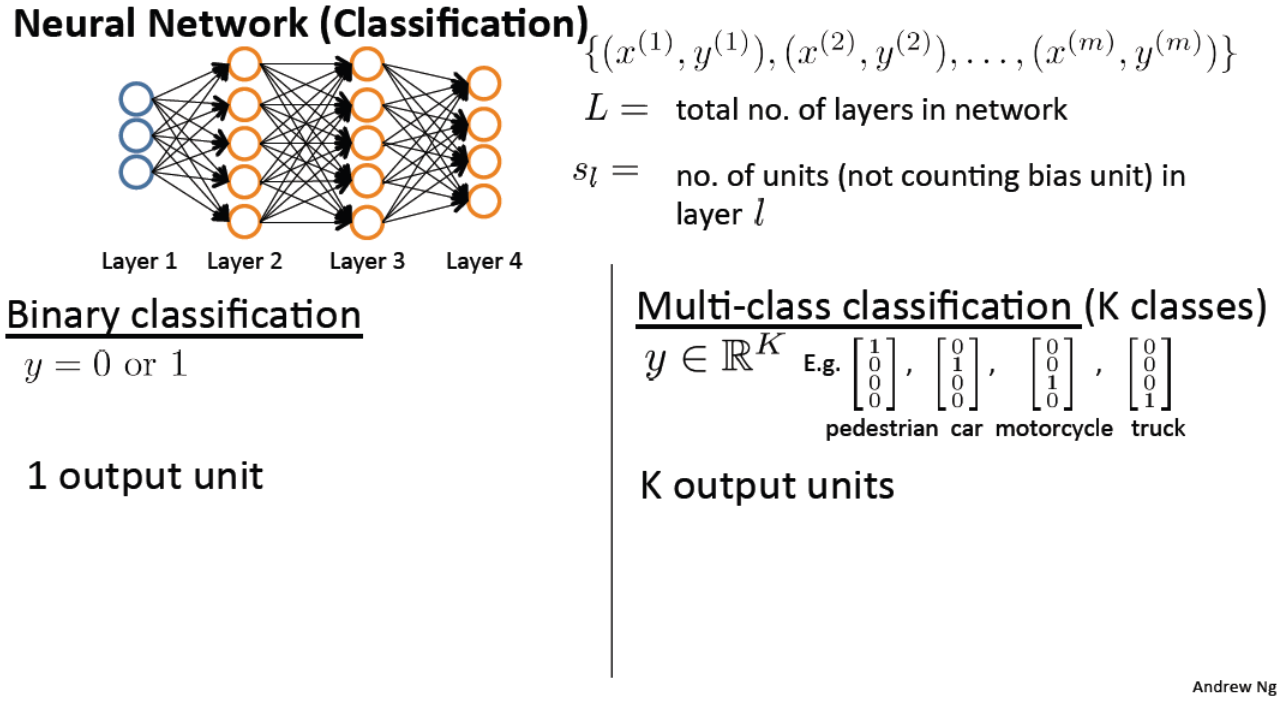


Figure 29: Neural Network and notation for defining the Cost Function.

8 Cost Function

Neural Networks are one of the most powerful learning algorithms that we have today. In this and in the next few videos, I'll start talking about a learning algorithm for fitting the parameters of the NN, given the training set. As for the discussion of most of the learning algorithms, we start by talking about the cost function for fitting the parameters of the NN. I'm going to focus on the application of NNs to classification problems.

So, suppose we have a network like that in Fig. 29 and suppose we have a training set

$$\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots, (x^{(m)}, y^{(m)})\}$$

consisting of m pairs of training examples $(x^{(a)}, y^{(a)})$.

L denotes the total number of layers in the network: so, in this example $L = 4$.

And s_l will denote the number of units, that is the number of neurons, not counting the bias unit, which are found in layer l of the network. So, for example, we would have $s_1 = 3$ for the input layer; also, $s_2 = 5$ in the example because the hidden layer 2 has 5 neuron units in Fig. 29; and, for the output layer, $s_4 = s_L = 5$ because the output layer in the example has four output units.

We're going to consider two types of classification problems. The first is *binary classification*, where the labels y are either 0 or 1. In this case, we would have only one output unit. So, this NN in Fig. 29 has four output units, but if we had binary classification we would have only one output unit that would compute $h_\Theta(x)$, which then would be going to be a real number. In this case of binary classification $s_L = 1$, where L is the number of layers we have in the network and also the index of the output layer.

We use K to denote the number of classes in the classification problem, and that is also equal to the number of output units in the NN. In the binary classification case we have certainly $K = 1$.

The second type of classification problem we'll consider will be the *multiclass classification problem* where we may have K distinct classes. So, in our early example of spotting car images I had this representation for y, which

is one of the 4 possible vectors

$$y \in \left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \right\}$$

with $K = 4$ classes and $K = 4$ output units, and our hypotheses $h_\Theta(x)$ will output vectors that are K-dimensional. So, we have in general $h_\Theta(x) \in \mathbb{R}^K$ and the number of neurons in the output layer is $s_L = K$.

And usually we will have $K \geq 3$, because if we have only two classes ($K = 2$) we don't need to use the one versus all method: for two classes, we will need to use only one output unit.

Now, let's define the cost function for the Neural Network.

The cost function we use for the NN is going to be a generalization of the one that we use for logistic regression. For logistic regression, we used to minimize the cost function

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log h_\theta\left(x^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - h_\theta\left(x^{(i)}\right)\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n} \theta_j^2 \tag{5}$$

where the summation in the θ_j is the regularization term. Note that *we did not regularize the bias term* θ_0 (the summation in j starts at 1).

For a neural network our cost function is going to be a generalization of that one, where instead of having just one logistic regression output unit we may instead have K of them. So here's our **cost function for the Neural Network**

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log\left(h_\Theta\left(x^{(i)}\right)\right)_k + \left(1 - y_k^{(i)}\right) \log\left(1 - \left(h_\Theta\left(x^{(i)}\right)\right)_k\right)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}\left(\Theta_{ji}^{(l)}\right)^2 \tag{6}$$

The NN now outputs vectors $h_\Theta(x) \in \mathbb{R}^K$, where K might be equal to 1 if we had the binary classification problem. I'm going to use $\left(h_\Theta\left(x\right)\right)_k$ to denote the k-th output, because $h_\Theta(x)$ is in general a K-dimensional vector. And so, the subscript k just selects out the k-th element of the vector that is output by my NN.

The NN cost function, $J(\Theta)$, as shown in (6), is a sum of terms similar to what we have in logistic regression (5), except that in the NN we have a summation going from $k = 1$ upto K, running over the K output units. So, if the final layer of my NN has 4 output units as in the example, then k goes from 1 upto 4. Basically, the NN cost function is that of the logistic regression algorithm (5), but adds to that the summing of terms over each of the $K = 4$ output units in turn.

And, finally, the second term in (6) is the regularization term similar to what we had for logistic regression (5). This summation term for regularization

$$\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}\left(\Theta_{ji}^{(l)}\right)^2$$

looks really complicated, but in fact it only is summing all the weights which are elements of the $\Theta^{(l)}$ matrices, except that we don't sum over the terms corresponding to the bias values which are those terms where $i = 0$. So, that is because when we are computing the activation of the neuron we have terms like $\Theta_{j0}^{(l)}x_0 + \Theta_{j1}^{(l)}x_1 + \cdots$ (where the bias would be $x_0 = 1$) and so on, and so the values with $i = 0$ that correspond to something that multiplies x_0 or a_0 and so, are like bias units and we won't sum over those terms in our regularization term.

But not summing those $\Theta_{j0}^{(l)}$ weights, which ponder the bias units, is just one possible convention, and even if you were to sum over them, starting the summation at $i = 0$, the algorithm will work about the same and it doesn't make a big difference. But maybe this convention of not regularizing the bias term is just slightly more common.

So, that was the cost function we're going to use in our NN. In the next video we'll start to talk about an algorithm for optimizing the cost function $J(\Theta)$.

Given one training example (x, y):

Forward propagation:

$$
\begin{aligned}
a^{(1)} &= x \\
z^{(2)} &= \Theta^{(1)} a^{(1)} \\
a^{(2)} &= g(z^{(2)}) \quad (\text{add } a_0^{(2)}) \\
z^{(3)} &= \Theta^{(2)} a^{(2)} \\
a^{(3)} &= g(z^{(3)}) \quad (\text{add } a_0^{(3)}) \\
z^{(4)} &= \Theta^{(3)} a^{(3)} \\
a^{(4)} &= h_\Theta(x) = g(z^{(4)})
\end{aligned}
$$

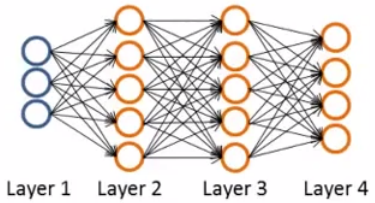


Figure 30: Gradient computation.

9 Backpropagation Algorithm

In the previous video we talked about a cost function for the neural network. In this video we start to talk about an algorithm for minimizing the cost function. In particular, we'll talk about the backpropagation algorithm (BPA).

Here's the NN cost function $J(\Theta)$ that we wrote down in the previous video

$$
J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log\left(h_\Theta\left(x^{(i)}\right)\right)_k + \left(1 - y_k^{(i)}\right) \log\left(1 - \left(h_\Theta\left(x^{(i)}\right)\right)_k\right)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}\left(\Theta_{ji}^{(l)}\right)^2 \quad (7)
$$

and what we'd like to do is to find parameters Θ to try to minimize $J(\Theta)$.

In order to use either gradient descent or one of the more advanced optimization algorithms, we need to write code that takes the parameters Θ as input and computes $J(\Theta)$ and the partial derivatives

$$
\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)
$$

Remember that the parameters in the the NN are the real numbers $\Theta_{ij}^{(l)} \in \mathbb{R}$, and so the partial derivative terms we need to compute are in respect to them.

In order to compute the cost function $J(\Theta)$ we just use the formula (7) and so what I want to do for the most of this video is focus on talking about how we can compute the partial derivatives $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$.

Let's start by talking about the case of when we have only one training example (Fig. 30), which is a single pair (x, y). Let's step through the sequence of calculations we would do with this one and single training example.

The **first thing we do is to apply forward propagation** in order to compute what a hypothesis actually outputs given the input x. Concretely, the vector $a^{(1)}$ are the activation values of the first layer, that is the input layer. So, in fact $a^{(1)} = x$ and then we're going to compute $z^{(2)} = \Theta^{(1)} a^{(1)}$ and $a^{(2)} = g(z^{(2)})$, that is, the sigmoid activation function is applied to $z^{(2)}$, and these first steps would give us our activations for the first hidden layer, that is for layer 2 of the NN, and we also add those bias terms $a_0^{(2)}$, $a_0^{(3)}$, etc... (see the equations in Fig. 30).

Next we apply two more steps of this forward propagation to compute $a^{(3)}$ and $a^{(4)}$ – which is also our hypothesis, $h_\Theta(x)$.

The complete sequence of equations is shown in Fig. 30.

So, this was our vectorized implementation of forward propagation and it allows us to compute the activation values for all of the layers in our NN.

Next, in order to compute the derivatives, we're going to **use an algorithm called backpropagation**. The intuition of the BPA algorithm is that for each node we're going to compute the term

$$\delta_j^{(l)}$$

that's going to, somehow, represent the "error" of node j in the layer l.

So, recall that $a_j^{(l)}$ does the activation of the j of unit in layer l, and so this $\delta_j^{(l)}$ term is, in some sense, going to capture our error in the activation of that neural unit. So, now we might wish that the activation of that node is slightly different. Concretely, taking the example NN in Fig. 30 which has four layers, so $L = 4$, for each output unit we're going to compute

$$\delta_j^{(4)} = a_j^{(4)} - y_j \tag{8}$$

So, $\delta_j^{(4)}$ in the fourth layer is equal to the activation of that unit minus the actual value of y_j in our training example. But $a_j^{(4)} = \left(h_\Theta(x)\right)_j$ and so $\delta_j^{(4)}$ is just the difference between the hypothesis output and the j-th element of the vector y in our labeled training set.

And, by the way, if you think about δ, a and y as vectors, then you can come up with a vectorized implementation of (8)

$$\delta^{(4)} = a^{(4)} - y$$

where each of these $\delta^{(4)}$, $a^{(4)}$ and y is a vector whose dimension is equal to the number of output units in our neural network, a number which we call K.

So, we've now computed the output vector error term $\delta^{(4)}$ for our network. What we do next is to compute the $\delta^{(l)}$ terms for the earlier layers in our network, starting with

$$\delta^{(3)} \quad = \quad \left(\Theta^{(3)}\right)^T \delta^{(4)} \,.*\, g'(z^{(3)})$$

where $.*$ is the element-wise multiplication operation that we know from Octave.

So, $\left(\Theta^{(3)}\right)^T \delta^{(4)}$ is a vector, $g'(z^{(3)})$ is also a vector and so $.*$ is the element-wise multiplication between these two vectors. The term $g'(z^{(3)})$ formally is the derivative of the activation function $g()$ evaluated at the input value given by $z^{(3)}$. If you know calculus, you can check that

$$g'(z^{(3)}) = a^{(3)} \,.*\, (1 - a^{(3)})$$

Next, you apply a similar formula to compute

$$\delta^{(2)} = \left(\Theta^{(2)}\right)^T \delta^{(3)} \,.*\, g'(z^{(2)})$$

where again

$$g'(z^{(2)}) = a^{(2)} \,.*\, (1 - a^{(2)})$$

can be computed using a similar formula.

And finally, notice that there is no $\delta^{(1)}$ term, because the first layer corresponds to the input layer and that's just the features we observed in our training sets, so there isn't any error associated with that.

And so we have $\delta^{(l)}$ terms only for layers 2, 3 and 4 in this example.

The name *backpropagation* comes from the fact that we start by computing $\delta^{(4)}$ for the output layer, and then we go back a layer and compute $\delta^{(3)}$ for the third hidden layer, and then we go back another step to compute $\delta^{(2)}$, and so on, and so we're **sort of backpropagating the errors** from the output layer, to layer 3 and to layer 2 and hence the name backpropagation (BP).

Finally, although the derivation is surprisingly complicated and involved, it happens that if you just do these few steps of computation (calculating the $\delta^{(l)}$ in back-to-front sense, i.e., from output to input) and **if you ignore regularization**, then the partial derivative terms are given by

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)} \tag{9}$$

Backpropagation algorithm

Training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$

Set $\triangle_{ij}^{(l)} = 0$ (for all l, i, j).

For $i = 1$ to m

Set $a^{(1)} = x^{(i)}$

Perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, \ldots, L$

Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$

Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$

$\triangle_{ij}^{(l)} := \triangle_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

Figure 31: Backpropagation algorithm.

again recall that this is when ignoring λ or, alternatively, with the regularization term $\lambda = 0$. We'll fix this detail about the regularization term later.

In conclusion, by performing BP and computing the $\delta^{(l)}$ terms you can pretty quickly compute the partial derivative terms (9) for all of your parameters.

So this is a lot of detail. Let's take everything and put it all together to talk about how to implement BP to compute the derivatives (9) for the case when we have a large training set, not just a training set of one example (Fig. 31).

Suppose we have a training set of m examples

$$\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots, (x^{(m)}, y^{(m)})\}$$

The first thing we're going to do is to create a new set of variables, which are initializaed to zero

$$\Delta_{ij}^{(l)} = 0, \quad \forall_{l,i,j}$$

The symbol we used here is the *capital delta*, Δ, from the Greek alphabet; the symbol we had before, on the previous slide, was the lower case δ. Take care, because Δ and δ have different meanings in BP as we'll see below.

Eventually $\Delta_{ij}^{(l)}$ will be used to compute $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$.

So, as we'll see in a second, the $\Delta_{ij}^{(l)}$ are going *to be used as accumulators* that will slowly add things in order to compute the partial derivatives.

Next (see Fig. 31) we're going to loop through our training set, iterating i from 1 to m, and so for the i-th iteration we're going to work with the training example $(x^{(i)}, y^{(i)})$.

So, the first thing we're going to do inside the loop is to input our example and set the activations in the input layer equal to our inputs

$$a^{(1)} = x^{(i)}$$

and then we're going to perform forward propagation to compute the activations for layers 2, 3 and so on, up to the final layer, the output layer, layer L.

Next, we're going to use the output label $y^{(i)}$, from the specific example we're looking at, to compute the error term for the output layer

$$\delta^{(L)} = a^{(L)} - y^{(i)}$$

So, recalling that $a^{(L)} = h_\Theta(x^{(i)})$, then we see that $\delta^{(L)} = a^{(L)} - y^{(i)}$ is our hypothesis output minus what the target label $y^{(i)}$ is.

And then we're going to use the BPA to compute $\delta^{(L-1)}$, $\delta^{(L-2)}$, and so on, down to $\delta^{(2)}$ and, once again, recall that there is no $\delta^{(1)}$ because we don't associate any error with the input layer.

And finally, we're going to use the $\Delta_{ij}^{(l)}$ to accumulate the partial derivative terms $a_j^{(l)}\delta_i^{(l+1)}$, that is

$$\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)}\delta_i^{(l+1)} \tag{10}$$

This accumulation is along the `for` loop in Fig. 31, which spans all the m training examples[2].

It's possible to vectorize (10) too. Concretely, if you think of $\Delta_{ij}^{(l)}$ as a matrix indexed by subscript ij, then we can rewrite this as

$$\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)}\left(a^{(l)}\right)^T$$

and that's a vectorized implementation that automatically does an update for all values of i and j.

After executing all the iterations of the `for` loop, we afterwards compute the following terms

$$\begin{aligned} D_{ij}^{(l)} &:= \frac{1}{m}\Delta_{ij}^{(l)} + \lambda\Theta_{ij}^{(l)} \qquad \text{if } j \neq 0 \\ D_{ij}^{(l)} &:= \frac{1}{m}\Delta_{ij}^{(l)} \qquad\qquad\quad \text{if } j = 0 \end{aligned}$$

and notice we have two separate cases for $j \neq 0$ and $j = 0$. The case of $j = 0$ corresponds to the bias term so that's why we're missing an extra regularization term, $\lambda\Theta_{ij}^{(l)}$.

Finally, while the formal proof is pretty complicated, we can show that

$$\frac{\partial}{\partial\Theta_{ij}^{(l)}}J(\Theta) = D_{ij}^{(l)}$$

that is, once you've computed the $D_{ij}^{(l)}$ terms, *they're exactly the partial derivatives of the cost function with respect to each of your parameters or weigths*, and so you can use those in either gradient descent or in one of the more advanced optimization algorithms.

So that's the backpropagation algorithm and how you compute derivatives of your cost function for a NN. I know this looks like a lot of details and this was a lot of steps strung together. But both in the programming assignments, and later in this video, we'll give you a summary of this so that you can have all the pieces of the algorithm together and know exactly what you need to implement if you want to implement BPA to compute the derivatives of your NNs cost function $J(\Theta)$ with respect to those parameters $\Theta_{ij}^{(l)}$.

10 Backpropagation Intuition

In the previous video we talked about the backpropagation algorithm. To a lot of people seeing it for the first time, the first impression is often that it is a very complicated algorithm with a lot of different steps, and they're not quite sure how the steps fit together and its like kind of a black box with many complicated steps.

In case that's how you are feeling about BP, it's actually okay. Backpropagation may be unfortunately a less mathematically clean and simple algorithm compared to linear regression or to logistic regression. I've actually used back propagation pretty successfully for many years, and even today, I still don't sometimes feel like I have a very good sense of just what it's doing or I have intuition about what BP is doing.

For those of you that are doing the programming exercises, that will at least mechanically step you through the different steps of how to implement backpropagation so you will be able to get it to work for yourself. And what I want to do in this video is look a little bit more at the mechanical steps of BP and offer you a little more intuition about what the mechanical steps of BP is doing to, hopefully, convince you that it is a reasonable algorithm to apply in NN training.

In case even after this video backpropagation still seems very black box, with too many complicated steps, and seems a little bit magical to you, that's actually okay. Even though I have used BP for many years, sometimes it's a difficult algorithm to understand. But, hopefully, this video will help a little bit.

[2]N.T: in Fig. 31, don't mix the index i of the `for` loop, which goes from 1 upto m and is used in $x^{(i)}$ and in $y^{(i)}$, with the subscript i found in $\Delta_{ij}^{(l)}$ and in $\delta_i^{(l+1)}$, which refers to the i-th neuron in layer $l+1$.

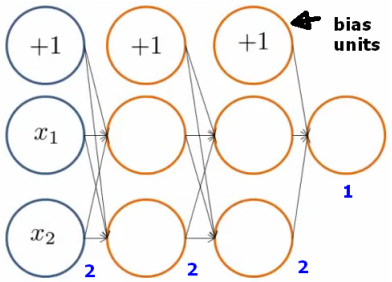


Figure 32: Revisiting forward propagation.

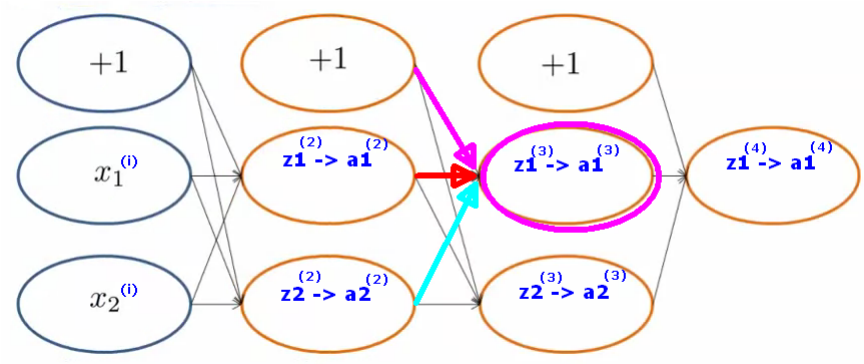


Figure 33: Details of FP.

In order to better understand backpropagation, let's take another closer look at what forward propagation is doing. In Fig. 32 is shown a NN with two input units (not counting the bias unit) labeled x_1 and x_2, two hidden layers with two units each and then, finally, one output unit. The counts 2, 2, 2, 1 of number of units in the NN are not including the bias units.

In order to illustrate forward propagation, the NN is redrawn with the nodes being very fat ellipses, in order to be possible to write text inside them (Fig. 33).

When performing forward propagation, we might have some particular example $(x^{(i)}, y^{(i)})$ in the training set and $x^{(i)} = [x_1^{(i)} \quad x_2^{(i)}]$ it what we feed into the input layer, such that $x_1 = x_1^{(i)}$ and $x_2 = x_2^{(i)}$.

When we forward propagate $x^{(i)}$ to the first hidden layer, what we do is compute $z_1^{(2)}$ and $z_2^{(2)}$, the weighted sum of the input units (by using the elements in $\Theta^{(1)}$ as weights) and then we apply the sigmoid activation function $g()$ to the z's to get the activation values $a_1^{(2)} = g(z_1^{(2)})$ and $a_2^{(2)} = g(z_2^{(2)})$. And then we forward propagate again to get $z_1^{(3)}$ and $z_2^{(3)}$, we apply the sigmoid activation function again to get $a_1^{(3)} = g(z_1^{(3)})$ and $a_2^{(3)} = g(z_2^{(3)})$, until we get $z_1^{(4)}$ and $a_1^{(4)} = g(z_1^{(4)})$ which is the final output value of the network.

To look at what this computation really is doing, we focus the hidden unit corresponding to $a_1^{(3)}$ (marked with the magenta ellipse in Fig. 33). The arrows pointing to it are drawn in magenta, red and cyan and the weights corresponding to those arrows are $\Theta_{10}^{(2)}$, $\Theta_{11}^{(2)}$ and $\Theta_{12}^{(2)}$ respectivelly. So we can compute

$$z_1^{(3)} = \Theta_{10}^{(2)} \times 1 + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)}$$

And so that's forward propagation.

It turns out that, as we will see later on in this video, BP is doing a process very similar to this except that, instead of the computations flowing from the left to the right of this network, the computations in BP flow from the right to the left of the network, but they are a very similar computation as that we've done in forward propagation with the example where $z_1^{(3)}$ is being calculated.

To better understand what BP is doing, let's look at the cost function when we have only one output unit

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log\left(h_\Theta\left(x^{(i)}\right)\right) + \left(1-y^{(i)}\right)\log\left(1-\left(h_\Theta\left(x^{(i)}\right)\right)\right)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}\left(\Theta_{ji}^{(l)}\right)^2$$

If only one output unit is present, then $K = 1$ and we don't have a summation over the output units through the index k, as we had in (6) or (7) (these two equations are really the same cost function). And so, we do forward propagation and backpropagation on one example at a time.

So, let's just focus on the single example $(x^{(i)}, y^{(i)})$ and focus on the case of having one output unit ($K = 1$) so that $y^{(i)}$ is just a real number, and let's ignore regularization, i.e. $\lambda = 0$ and the final regularization term goes away. With these simplifications, the cost function is

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \log\left(h_\Theta\left(x^{(i)}\right)\right) + \left(1-y^{(i)}\right)\log\left(1-\left(h_\Theta\left(x^{(i)}\right)\right)\right)\right]$$

If you look to the expression inside this summation, you find that it is the cost associated with training example $(x^{(i)}, y^{(i)})$; the cost of training example i is just written as follows

$$\text{cost}(i) = y^{(i)} \log h_\Theta\left(x^{(i)}\right) + \left(1-y^{(i)}\right)\log\left(1-h_\Theta\left(x^{(i)}\right)\right) \tag{11}$$

And what this cost function does, is it plays a role similar to the square error. So, rather than looking at this complicated expression, you can think of the cost as being approximately the square of the difference between the NN outputs and the actual training value, $y^{(i)}$

$$\text{cost}(i) \approx \left(h_\Theta\left(x^{(i)}\right) - y^{(i)}\right)^2$$

Just as in logistic regression, we actually prefer to use this slightly more complicated cost function using the logarithm but, for the purpose of intuition, feel free to think of the cost function as being sort of the squared error cost function.

And so this cost(i) measures how well is the network doing on correctly predicting example i, how close is the NN output to the actually observed label $y^{(i)}$.

Now let's look at what BP is doing (Fig. 34). One useful intuition is that BP is computing the $\delta_j^{(l)}$ terms, and we can think of these as the "error" of the activation value $a_j^{(l)}$ that we got for unit j in layer l. More formally, what the $\delta_j^{(l)}$ terms actually are is this: they're the partial derivatives of the cost function (11) with respect to $z_j^{(l)}$, that is with respect to the weighted sum of inputs that we use for computing the $z_j^{(l)}$ terms:

$$\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}}\text{cost}(i) \qquad (\text{for } j \geq 0)$$

So, concretely, the cost function (11) is a function of $y^{(i)}$ and of the output value of the NN, $h_\Theta\left(x^{(i)}\right)$. And if we could go inside the NN and just change those $z_j^{(l)}$ values a little bit, then that would affect $h_\Theta\left(x^{(i)}\right)$ and so that would end up changing the cost function.

Recall that these $\delta_j^{(l)}$ terms turn out to be the partial derivatives of the cost function with respect to these intermediate terms $z_j^{(l)}$ that we're computing. So they measure how much would we like to change the NN's weights in order to affect these intermediate values of the computation, $z_j^{(l)}$, so as to affect the output of the neural network $h_\Theta\left(x^{(i)}\right)$ and, therefore, affect the overall cost. Hopefully, that change should be defined such that a decrease in the cost function was observed.

(In case this partial derivative intuition didn't make sense, don't worry about it, we can go through the rest of this video without really talking about partial derivatives.)

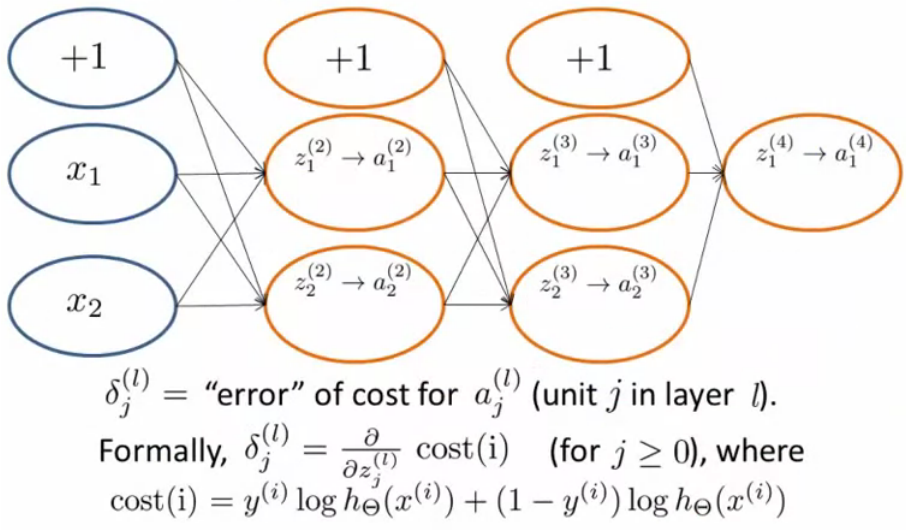


Figure 34: Intuition of backpropagation.

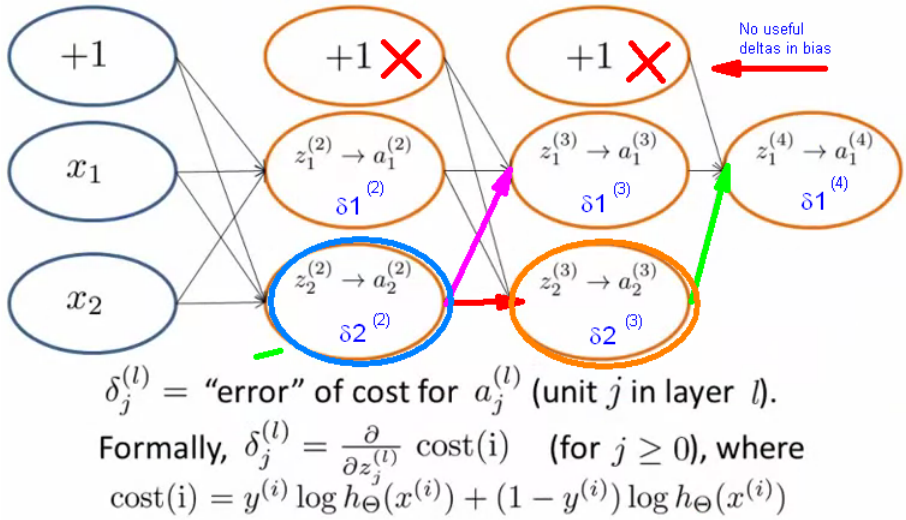


Figure 35: Intuition of backpropagation continued.

Let's look in more detail at what backpropagation is doing. For the output layer, if first sets

$$\delta_1^{(4)} = y^{(i)} - a_1^{(4)}$$

so it's really the error, it's the actual measured value of $y^{(i)}$ in this training set minus the value predicted $a_1^{(4)}$.

Next we're going to propagate the value $\delta_1^{(4)}$ backwards, by computing the δ terms of the previous layer, $\delta_1^{(3)}$ and $\delta_2^{(3)}$, and also $\delta_1^{(2)}$ and $\delta_2^{(2)}$ in the first hidden layer, the layer 2. The BP calculation is a lot like running the forward propagation algorithm, but doing it backwards (Fig. 35).

So here's what I mean. Let's look at $\delta_2^{(2)}$ (node covered with the blue ellipse in Fig. 35). Similar to forward propagation, let me label a couple of the weights. The weight in the magenta arrow is $\Theta_{12}^{(2)}$ and the weight in the red arrow is $\Theta_{22}^{(2)}$. So if we look at how $\delta_2^{(2)}$ is computed it's really a weighted sum of the δ values upstream weighted by the corresponding edge strength. So concretely, it is

$$\delta_2^{(2)} = \Theta_{12}^{(2)}\delta_1^{(3)} + \Theta_{22}^{(2)}\delta_2^{(3)}$$

And just as another example, let's look at $\delta_2^{(3)}$ (node covered with the orange ellipse). The weight in the arrow highlighted in green is $\Theta_{12}^{(3)}$ delta 3(1)2, then we have

$$\delta_2^{(3)} = \Theta_{12}^{(3)}\delta_1^{(4)}$$

And by the way, so far I've been writing the delta values only for the hidden units and I've been excluding the bias units. Depending on how you define the BPA you may end up implementing something to compute delta values for these bias units as well. The bias unit is always output the value +1, and there's no way for us to change the value. The way I usually implement it, I do end up computing these delta values, but I just discard them and we don't use them, because they don't end up being part of the calculation needed to compute the derivatives.

So, hopefully, that gives you a little bit of intuition about what BP is doing. In case all of this still seem so magical and so black box, later, in the "putting it together" video, I'll try to give a little more intuition about what that backpropagation is doing.

But, unfortunately, the BPA is a difficult algorithm to try to visualize and understand what it is really doing. But fortunately often I guess, many people have been using it very successfully for many years and if you adopt the algorithm, you have a very effective learning algorithm, even though the inner details of exactly how it works can be harder to visualize.

11 Implementation Note - Unrolling Parameters

In the previous video, we talked about how to use backpropagation to compute the derivatives of the cost function. In this video, I want to quickly tell you about one implementation detail of unrolling your parameters from matrices into vectors, which we need in order to use the advanced optimization routines.

Concretely, let's say you've implemented a cost function that takes the parameters Θ as input and returns the cost function $J(\Theta)$ and returns also derivatives (or gradient). Then you can pass Θ to the advanced optimization algorithm implemented by `fminunc()` or other similar advanced optimization algorithms. All of them take as input a pointer `@costFunction` for the cost function, and some initial value of Θ named `initialTheta`.

These routines assume that Θ and the initial value of Θ, say $\Theta_{initial}$, are parameter *vectors*, maybe $\mathbb{R}^{n+1}$. And your cost function will return as a second return value this `gradient` which is also $\mathbb{R}^{n+1}$, with the same dimension as Θ.

This worked fine when we were using logistic regression, but now in the full NN our parameters are no longer vectors, but instead they are these parameter matrices $\Theta^{(1)}$, $\Theta^{(2)}$, $\Theta^{(3)}$ that we might represent in Octave as these matrix names `Theta1`, `Theta2` and `Theta3`. And similarly these gradient terms $D^{(1)}$, $D^{(2)}$, $D^{(3)}$ that we are expected to return, we showed how to compute these gradient matrices, which we might represent in Octave as `D1`, `D2`, `D3`.

In this video I want to quickly tell you about the idea of how to take these matrices and unroll them into vectors, so that they end up being in a format suitable for passing into the functions as Θ, or for getting out from the `costFunction` as `gradient` or $D^{(j)}$.

Advanced optimization

```
function [jVal, gradient] = costFunction(theta)
   ...
optTheta = fminunc(@costFunction, initialTheta, options)
```

Neural Network (L=4):

$\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ - matrices (`Theta1, Theta2, Theta3`)

$D^{(1)}, D^{(2)}, D^{(3)}$ - matrices (`D1, D2, D3`)

"Unroll" into vectors

Figure 36: Template for minimization of cost function.

Example

$s_1 = 10, s_2 = 10, s_3 = 1$

$\Theta^{(1)} \in \mathbb{R}^{10\times 11}, \Theta^{(2)} \in \mathbb{R}^{10\times 11}, \Theta^{(3)} \in \mathbb{R}^{1\times 11}$

$D^{(1)} \in \mathbb{R}^{10\times 11}, D^{(2)} \in \mathbb{R}^{10\times 11}, D^{(3)} \in \mathbb{R}^{1\times 11}$

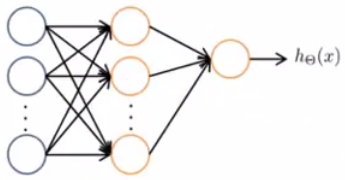


Figure 37: Example for illustrating matrix unrolling into vectors.

Concretely, in Fig. 37 we have a NN with one input layer with 10 units, one hidden layer with 10 units and one output layer with just one unit.

So, the s_l are the number of units in layer l. In this case, the dimension of the matrices Θ and D are going to be given by

$$\Theta^{(1)} \in \mathbb{R}^{10\times 11}, \quad \Theta^{(2)} \in \mathbb{R}^{10\times 11}, \quad \Theta^{(3)} \in \mathbb{R}^{1\times 11}$$

and

$$D^{(1)} \in \mathbb{R}^{10\times 11}, \quad D^{(2)} \in \mathbb{R}^{10\times 11}, \quad D^{(3)} \in \mathbb{R}^{1\times 11}$$

that is, $\Theta^{(1)}$ is going to a 10×11 matrix, and so on.

So, if you want to convert these matrices into vectors, you take your $\Theta^{(1)}$, $\Theta^{(2)}$ and $\Theta^{(3)}$ and write this piece of Octave code

```
thetaVec = [ Theta1(:); Theta2(:); Theta3(:)]
```

and this will take all the elements of your three $\Theta^{(l)}$ matrices and unroll them and put all the elements into a big long vector, `thetaVec`. And similarly

```
DVec = [ D1(:); D2(:); D3(:)]
```

would take all of your D matrices and unroll them into a big long vector, called `DVec`.

And finally, if you want to go back from the vector representations to the matrix representations, what you do is use the `reshape()` function. For instance to get back to $\Theta^{(1)}$ (or `Theta1` in Octave) when coming from `thetaVec`, you need to pull out the first 110 elements of `thetaVec` and also give to `reshape()` the size of the matrix to be built:

```
Theta1 = reshape(thetaVec(1:110),10,11);
```

```
octave-3.6.1.exe:1> PS1('>> ')
>> Theta1=ones(2,3)
Theta1 =

   1   1   1
   1   1   1

>> Theta2=2*ones(2,3)
Theta2 =

   2   2   2
   2   2   2

>> Theta3=3*ones(1,3)
Theta3 =

   3   3   3

>> thetaVec = [ Theta1(:); Theta2(:); Theta3(:)];
>> thetaVec'
ans =

   1   1   1   1   1   1   2   2   2   2   2   2   3   3   3

>> size(thetaVec)
ans =

   15    1

>> reshape(thetaVec(1:6),2,3)
ans =

   1   1   1
   1   1   1

>> reshape(thetaVec(7:12),2,3)
ans =

   2   2   2
   2   2   2

>> reshape(thetaVec(13:15),1,3)
ans =

   3   3   3

>> _
```

Figure 38: Matrix-vector unrolling and the `reshape()` function. The last three `resahape()` commands recover the original Theta matrices.

So, $\Theta^{(1)}$ has 110 elements because it's a 10×11 matrix, so you pull out the first 110 elements of `thetaVec` . And similarly for `Theta2` and `Theta3`:

```
Theta2 = reshape(thetaVec(111:220),10,11);
Theta3 = reshape(thetaVec(221:231),1,11);
```

In Fig. 38 is presented a quick Octave demo of that process (N.T: I use smaller matrices in order to display them in the Octave console: the dimensions 10 and 11 were converted to 2 and 3, respectively. Accordingly, the boundaries used in the `reshape()` commands are changed to conform with those modified dimensions.).

In general, `thetavec` is going to be a very long vector, 231 elements if the original sizes of the matrices are taken into account.

`thetaVec` has all the elements of the first matrix, all the elements of the second matrix, and then all the elements of the third matrix. I want to get back my original matrices, I can do `reshape(thetaVec()...)`, as is shown in Fig. 38 in the snapshot of the Octave console.

To make this process really concrete, here's how we use the unrolling idea to implement our learning algorithm.

Learning Algorithm

Have initial parameters $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$.

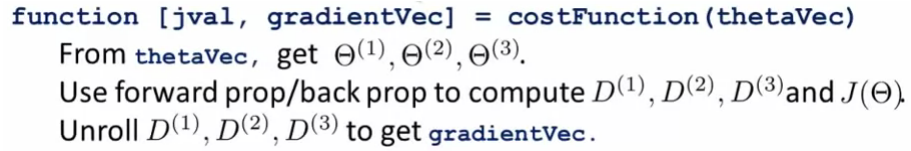
Unroll to get `initialTheta` to pass to

`fminunc(@costFunction, initialTheta, options)`

Let's say that we have some initial value of the parameters $\Theta^{(1)}$, $\Theta^{(2)}$ and $\Theta^{(3)}$. We take these and unroll them into

a long vector we're gonna call `initialTheta` to pass into `fminunc()` as the initial setting of the parameters `Theta`.

The other thing we need to do is to implement the cost function $J(\Theta)$ as `costFunction()` in Octave. Here's my implementation of the cost function.



The `costFunction()` is going to receive as input the vector `thetaVec`, which has all the matrices of parameters unrolled into a single vector. So the first thing to do is to get `thetaVec` and use the `reshape()` functions to recover the $\Theta^{(l)}$ matrices. That gives me a more convenient form in which to use these matrices so that I can run forward propagation and backpropagation to compute the derivatives $D^{(l)}$ and to compute the cost function $J(\Theta)$.

And finally, I unroll my derivatives into the vector `gradientVec` keeping the elements in the same ordering as I did when I unroll my Θ matrices. And now my cost function can return a vector of the derivatives, `gradientVec`, and also the cost `jval`.

So, hopefully, you now have a good sense of how to convert back and forth between matrix and vector representations of the parameters. The advantage of the matrix representation is that it's more convenient when you're doing forward propagation and backpropagation, and then it's easier to take advantage of the vectorized implementations. Whereas, in contrast, the advantage of the vector representation, when you have things like `thetaVec` or `DVec`, is that the advanced optimization algorithms tend to assume that you have all of your parameters unrolled into a big long vector. And so, with what we just went through, hopefully you can now quickly convert between the two parameter representations as needed.

12 Gradient Checking

In the last few videos, we talked about how to do forward propagation and backpropagation in a NN in order to compute derivatives.

But the BPA has a lot of details and can be a little bit tricky to implement. An unfortunate property is that there are many ways to have subtle bugs in BPA so that if you run it with gradient descent, or with some other optimization algorithm, it could actually look like it's working.

And your cost function $J(\Theta)$ may end up decreasing on every iteration of gradient descent, and the error could pull through even though there might be some bug in your implementation of backpropagation. So it looks like $J(\Theta)$ is decreasing, but you might just wind up with a neural network that has a higher level of error than you would with a bug-free implementation and you might just not know that there was this subtle bug that's giving you this bad performance.

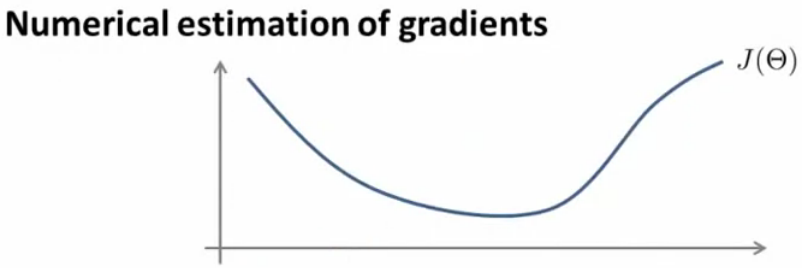
So what can we do about this? There's an idea called gradient checking that eliminates almost all of these problems. So, today, every time I implement BP or a similar gradient descent algorithm on the NN or on any other reasonably complex model, I always implement gradient checking.

And if you do this, it will help you make sure and sort of gain high confidence that your implementation of forward propagation, backpropagation or whatever, is 100% correct.

And in what I've seen this gradient checking pretty much elliminates all the problems associated with sort of a buggy implementation of the backpropagation.

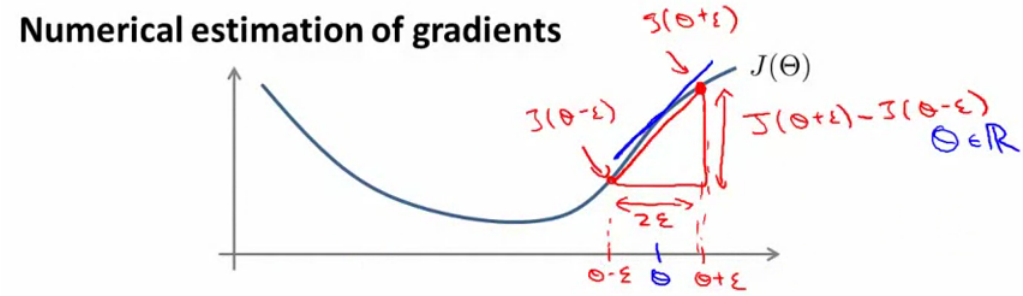
In the previous videos, I sort of asked you to take on faith that the formulas I gave for computing the δ's, and D's, and so on, actually do compute the gradients of the cost function. But once you implement numerical gradient checking, which is the topic of this video, you'll be able to verify for yourself that the code you're writing is indeed computing the derivative of the cost function $J(\Theta)$.

So, here's the idea. Consider the following example.



Suppose I have the function $J(\Theta)$ and I have some value, Θ, a real number, i.e. $\Theta \in \mathbb{R}$. And let's say I want to estimate the derivative of this function $J(\Theta)$ at this point Θ. And so the derivative is equal to the slope of that sort of tangent line at that point.

Here's a procedure for numerically approximating the derivative: I'm going to compute $\Theta + \epsilon$ a value a little bit to the right of Θ, and to compute $\Theta - \epsilon$, a value a little bit to the left of Θ. And I'm going to look at those two points, $(\Theta - \epsilon, J(\Theta - \epsilon))$ and $(\Theta + \epsilon, J(\Theta + \epsilon))$, and connect them by a straight line (in red, in the picture below). And I'm going to use the slope of that little red line as my approximation to the true derivative, which is the slope of the blue line over there.



So, you know, it seems like it would be a pretty good approximation.

Mathematically, the slope of this diagonal red line is this vertical height divided by this horizontal width. So this point on top is $J(\Theta+\epsilon)$ and this in the bottom is $J(\Theta-\epsilon)$. So this red vertical difference is $J(\Theta+\epsilon)-J(\Theta-\epsilon)$ and this horizontal red distance is just 2ϵ. So, my approximation of the derivative of $J(\Theta)$ with respect to Θ is going to be the *two-sided difference*

$$\frac{d}{d\Theta}J(\Theta) \approx \frac{(\Theta+\epsilon) - J(\Theta-\epsilon)}{2\epsilon} \tag{12}$$

Usually, I use a pretty small value for ϵ and set $\epsilon \approx 10^{-4}$.

There's usually a large range of different values for ϵ that work just fine. And in fact, if you let ϵ become really small then, mathematically, this approximation here actually becomes the derivative, becomes exactly the slope of the function at this point Θ.

We don't want to use ϵ that's too small because then you might run into numerical problems. So, you know, I usually use $\epsilon \approx 10^{-4}$, say.

And, by the way, some of you may have seen before an alternative formula for estimating the derivative

$$\frac{d}{d\Theta}J(\Theta) \approx \frac{(\Theta+\epsilon) - J(\Theta)}{\epsilon} \tag{13}$$

This one is called the *one-sided difference*, whereas the formula (12) that's called a two-sided difference. The two-sided difference gives a slightly more accurate estimate of te derivative $\frac{d}{d\Theta}J(\Theta)$, so I usually use it rather than just this one-sided difference estimate (13).

So, concretely, what you implement in Octave is

```
gradApprox = (J(theta + EPSILON) - J(theta - EPSILON))
             /(2*EPSILON)
```

You compute `gradApprox` which is going to be the approximation to the derivative given by the two-sided difference (12). And this will give you a numerical estimate of the gradient (or derivative) at that point Θ. And in this example it seems like it's a pretty good estimate.

Let's look at the more general case of where θ is a parameter vector, instead of a real number.

Parameter vector θ

$\theta \in \mathbb{R}^n$ (E.g. θ is "unrolled" version of $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$)

$\theta = \theta_1, \theta_2, \theta_3, \ldots, \theta_n$

So let's say $\theta \in \mathbb{R}^n$, and it might be an unrolled version of the parameters or weights of our neural network $\Theta^{(1)}$, $\Theta^{(2)}$ and $\Theta^{(3)}$. So θ is a vector that has n elements, $\theta = [\theta_1, \theta_2, \cdots, \theta_n]$.

We can then use a similar idea to approximate all of the partial derivative terms:

$$\frac{\partial}{\partial\theta_1} J(\theta) \approx \frac{J(\theta_1+\epsilon,\theta_2,\theta_3,\ldots,\theta_n) - J(\theta_1-\epsilon,\theta_2,\theta_3,\ldots,\theta_n)}{2\epsilon}$$

$$\frac{\partial}{\partial\theta_2} J(\theta) \approx \frac{J(\theta_1,\theta_2+\epsilon,\theta_3,\ldots,\theta_n) - J(\theta_1,\theta_2-\epsilon,\theta_3,\ldots,\theta_n)}{2\epsilon}$$

$$\vdots$$

$$\frac{\partial}{\partial\theta_n} J(\theta) \approx \frac{J(\theta_1,\theta_2,\theta_3,\ldots,\theta_n+\epsilon) - J(\theta_1,\theta_2,\theta_3,\ldots,\theta_n-\epsilon)}{2\epsilon}$$

Concretely, the partial derivative of the cost function $J(\Theta)$ with respect to the i-th parameter, θ_i, can be obtained by taking $J()$ with the argument θ_i *increased* by ϵ, subtracting from it the value of $J()$ calculated now with the argument θ_i *minus* ϵ, and dividing the result of that subtraction by 2ϵ, that is:

$$\frac{\partial}{\partial\theta_i} J(\theta) \approx \frac{J(\theta_1, \cdots, \theta_i + \epsilon, \cdots, \theta_n) - J(\theta_1, \cdots, \theta_i - \epsilon, \cdots, \theta_n)}{2\epsilon}$$

and doing this for $i = 1, 2, \cdots, n-1, n$.

So, these equations give you a way to numerically approximate the partial derivative of $J()$ with respect to any one of your parameters. Concretely, what you implement in Octave is therefore the following

```
for i = 1:n,
   thetaPlus = theta;
   thetaPlus(i) = thetaPlus(i) + EPSILON;
   thetaMinus = theta;
   thetaMinus(i) = thetaMinus(i) - EPSILON;
   gradApprox(i) = (J(thetaPlus) - J(thetaMinus))
                   /(2*EPSILON);
end;
```

We set an index `i` which sweeps all the `n` elements in the parameter vector `theta`, which is the unrolled version of the parameters, just a long list of all the parameters in my NN.

The vector `thetaPlus` will be equal to `theta` but for the i-th element `thetaPlus(i)` which is set to `theta(i)+EPSILON`; and similarly thetaMinus is something similar, except that `thetaMinus(i)` is set to `theta(i)-EPSILON`, that is

$$\text{thetaPlus} \approx \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_i + \epsilon \\ \vdots \\ \theta_n \end{bmatrix} \qquad \text{thetaMinus} \approx \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_i - \epsilon \\ \vdots \\ \theta_n \end{bmatrix}$$

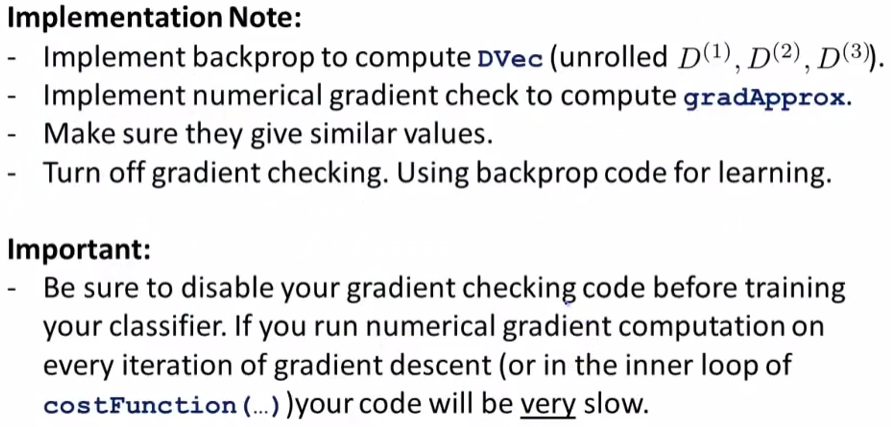


Figure 39: Summary of numerical gradient checking.

And then, finally, you implement

```
gradApprox(i) = (J(thetaPlus)-J(thetaMinus))/(2*EPSILON)
```

and this will give you your approximation to the partial derivative $\frac{\partial}{\partial \theta_i} J(\theta)$.

And the way we use this in our NN implementation, is we implement the `for` loop to compute partial derivatives of the cost function $J(\theta)$ with respect to every parameter θ_i in our network, that is, we get an estimate of the gradient, and then take the gradient that we got from backpropagation, `DVec`, i.e. the derivatives we got from the BPA, and recall BP is a relatively efficient way to compute the partial derivatives of a cost function with respect to all of our parameters. And then we check the approximation

$$\text{gradApprox} \approx \text{DVec}$$

down to small values of numerical round off .

And if these two ways of computing the derivative give me the same answer or, at least, give me very similar answers, you know, up to a few decimal places, then I'm much more confident that my implementation of BPA is correct. And when I plug the `DVec` vectors into gradient descent or some advanced optimization algorithm, I can then be much more confident that I'm computing the derivatives correctly and therefore, hopefully my codes will run correctly and do a good job optimizing $J(\theta)$.

Finally, in Fig. 39 I put everything together and tell you how to implement numerical gradient checking.

First thing I do, is implement backpropagation to compute `DVec`.

So, this is a procedure we talked about in an earlier video to compute `DVec`, our unrolled version of the $D^{(l)}$ matrices. Then I implement a numerical gradient checking to compute `gradApprox`, and you should make sure that `DVec` and `gradApprox` have similar values, let's say up to a few decimal places.

And finally, and this is a very important step, **before you start to use your code for learning, for seriously training your network, it is important to turn off gradient checking** and to no longer compute `gradApprox` using the numerical derivative formulas. And the reason for that is because the numeric gradient checking code is very computationally expensive, it is a very slow way to try to approximate the derivative. Whereas, in contrast, the backpropagation algorithm where we compute `DVec`, or $D^{(1)}$, $D^{(2)}$, $D^{(3)}$, is a much more computationally efficient way of computing the derivatives. So, once you've verified that your implementation of BP is correct you should turn off gradient checking, and just stop using that.

So just to reiterate, **you should be sure to disable your gradient checking code before running your algorithm for many iterations of gradient descent or of the advanced optimization algorithms in order to train your classifier**.

Concretely, if you were to run numerical gradient checking on every single integration of gradient descent, or if you were in the inner loop of your `costFunction()`, then your code would be very slow. Because the numerical

gradient checking code is much slower than the BPA, i.e. than the BP method where you, remember, were computing $\delta^{(4)}$, $\delta^{(3)}$, $\delta^{(2)}$, and so on, which is a much faster way to compute derivatives than gradient checking.

So, when you're ready, once you verify the implementation of BPA is correct, make sure you turn off, or you disable, your gradient checking code while you train your algorithm, or else your code could run very slowly.

So that's how you take gradients numerically. And that's how you can verify that your implementation of BP is correct. Whenever I implement backpropagation or a similar gradient descent algorithm for a complicated model, I always use gradient checking. This really helps me make sure that my code is correct.

13 Random Initialization

In the previous videos we put together almost all the pieces you need in order to implement and train your network. There's just one last idea I need to share with you, which is the idea of random initialization.

Initial value of Θ

For gradient descent and advanced optimization method, need initial value for Θ.

```
optTheta = fminunc(@costFunction,
           initialTheta, options)
```

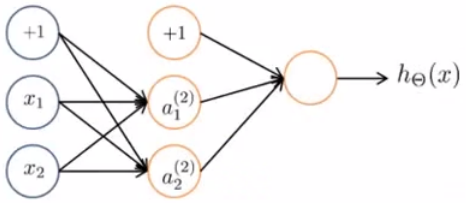
Consider gradient descent

Set `initialTheta = zeros(n,1)` ?

When you run an algorithm like gradient descent or some of the advanced optimization algorithms, you need to pick some initial value for the parameters Θ. The advanced optimization algorithm `fminunc()` assumes that you will pass into it some initial values for the parameters Θ. Now let's consider gradient descent; we also need to initialize Θ to something. And then we can slowly take steps downhill to minimize the function $J(\Theta)$.

So, what do we set the initial value of Θ to? Is it possible to set the initial Θ to a vector of zeroes. Whereas this worked okay when we were using logistic regression, initializing all of your parameters to zero actually does not work when you're training a neural network.

Zero initialization



$$\Theta_{ij}^{(l)} = 0 \text{ for all } i, j, l.$$

Consider training the NN above, and let's say we initialized all of the parameters $\Theta_{ij}^{(l)}$ to zero. And if you do that, then all the weights from layer 1 to layer 2 are all zero. What that means is that both of your hidden units $a_1^{(2)}$ and a_2^2 are going to be computing the same function of your inputs, i.e. $a_1^{(2)} = a_2^{(2)}$. And thus, for every your training examples, you end up with $a_1^{(2)} = a_2^{(2)}$.

And moreover, because the weights between the hidden and the output layers are the same, it can also be shown that the delta values in BP are also going to be the same, i.e. $\delta_1^{(2)} = \delta_2^{(2)}$.

And if we work through the map further, we find that the partial derivatives with respect to your parameters will satisfy some relations, such as

$$\frac{\partial}{\partial \Theta_{01}^{(1)}} J(\Theta) = \frac{\partial}{\partial \Theta_{02}^{(1)}} J(\Theta)$$

(these go from the +1 bias unit in the input layer to the hidden layer units $a_1^{(2)}$ and a_2^2).

And so, what this means is that, even after one gradient descent update, those two weights, $\Theta_{01}^{(1)}$ and $\Theta_{02}^{(1)}$, will end up the same as each other; they'll be some non-zero value now, but they are equal to each other

$$\Theta_{01}^{(1)} = \Theta_{02}^{(1)}$$

And similarly, even after one gradient descent update,

$$\begin{aligned} \Theta_{11}^{(1)} &= \Theta_{12}^{(1)} \\ \Theta_{21}^{(1)} &= \Theta_{22}^{(1)} \end{aligned}$$

So, after each update, the parameters corresponding to inputs going to each of the two hidden units are identical.

And what that means is that even after one iteration of say, gradient descent, you find that your two hidden units are still computing exactly the same function that the input, so you still have $a_1^{(2)} = a_2^{(2)}$. And so you're back to the initial case. And as you keep running gradient descent the equalities above will remain.

And what this means is that your NN really can't compute very interesting functions. Imagine that you had not only two hidden units but many many hidden units. Then what this is saying is that all of your hidden units are computing the exact same feature, the exact same function of the input. And this is a highly redundant representation because that means that your final logistic regression unit, you know, really only gets to see one feature because all of the units in the next-to-the-last layer are computing the same feature, and it prevents your NN from learning something interesting.

Random initialization: Symmetry breaking

Initialize each $\Theta_{ij}^{(l)}$ to a random value in $[-\epsilon, \epsilon]$
(i.e. $-\epsilon \leq \Theta_{ij}^{(l)} \leq \epsilon$)

E.g.

```
Theta1 =  rand(10,11)*(2*INIT_EPSILON)
          - INIT_EPSILON;

Theta2 =  rand(1,11)*(2*INIT_EPSILON)
          - INIT_EPSILON;
```

In order to get around this problem, the way to initialize the parameters of a NN is with random initialization. The problem we saw before is sometimes called the **problem of symmetric weights**, and so with random initialization is how we perform symmetry breaking.

We initialize each value $\Theta_{ij}^{(l)}$ to a random number between $-\epsilon$ and $+\epsilon$ (interval $[-\epsilon, \epsilon]$), so my weights or my parameters satisfy $-\epsilon \leq \Theta_{ij}^{(l)} \leq +\epsilon$. The way to code this in Octave is shown above, when $\Theta^{(1)}$ is a 10×11 dimensional matrix and $\Theta^{(2)}$ is a 1×11 matrix. The function `rand()` outputs values from a uniform distribution between 0 and 1, and so if you multiply those by 2ϵ and subtracts ϵ, then you end up with a number that's inside the interval $[-\epsilon, \epsilon]$.

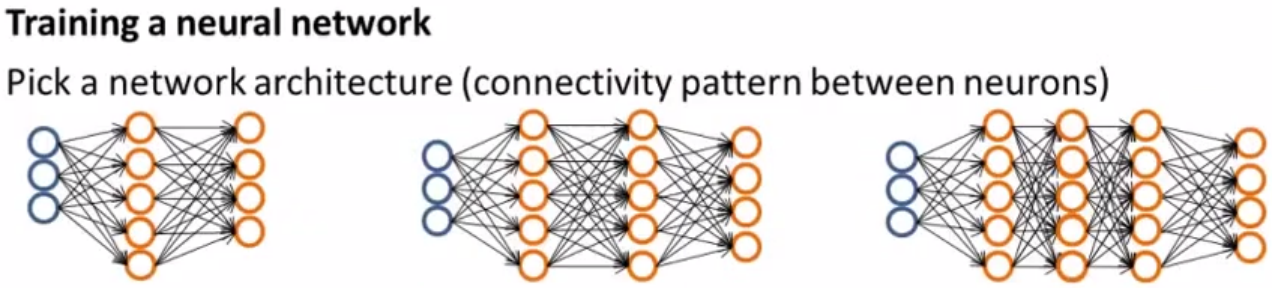
(And incidentally, this ϵ here has nothing to do with the ϵ that we were using before when we were doing gradient checking. When we were doing numerical gradient checking, we were adding some values of ϵ to Θ, and this one and that in random initialization are unrelated values of ϵ, which is why I am denoting it by INIT_EPSILON in the Octave code for random initialization, just to distinguish it from the value of ϵ we were using in gradient checking.)

So, to summarize, to train a NN what you should do is randomly initialize the weights $\Theta_{ij}^{(l)}$ to small values close to 0, between $-\epsilon$ and $+\epsilon$ say, and then implement backpropagation, do gradient checking and use either gradient descent or one of the advanced optimization algorithms to minimize $J(\Theta)$ as a function of the parameters Θ. And by doing symmetry breaking, which is this process, hopefully the optimization algorithms will be able to find a good value of Θ.

14 Putting It Together

So, it's taken us a lot of videos to get through the NN learning algorithm. In this video, what I'd like to do is try to put all the pieces together to give an overall summary, or a bigger picture view, of how all the pieces fit together and of the overall process of how to implement a NN learning algorithm.

When training a NN, the first thing you need to do is pick some network architecture, and by architecture I just mean the connectivity pattern between the neurons.



So we might choose between, say, a NN with 3 input units and 5 hidden units and 4 output units (the left architecture), versus one of 3 input, 5 hidden, 5 hidden, 4 output units (architecture in the center) and in the architecture in the right there are 3 units in the input layer, 5 units in each of the three hidden layers, and 4 output units. And so these choices of how many hidden units are in each layer, and how many hidden layers are in the NN, those are the **architecture choices**.

So, how do you make these choices? Well, the number of input units $x^{(i)}$, i.e. the number of features, is pretty well defined. Once you decide on the fix set of features $x^{(i)}$, then the number of input units will just be the dimension of your features $x^{(i)}$. And if you are doing multiclass classification, the number of output units will be determined by the number of classes in your classification problem.

And just as a reminder, if you have a multiclass classification where y takes on, say, values between 1 and 10, i.e. $y \in \{1, 2, 3 \ldots, 9, 10\}$, so that you have ten possible classes, then remember to write your output y a vector, in this case a vector with one '1' and nine '0', where the '1' indicates, and is in the position of, whichever class to which the training example we're using belongs:

$$y^{(i)} \in \left\{ \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ \vdots \end{bmatrix}, \cdots, \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} \right\}$$

So, if one of these examples takes on the fifth class, you know, $y = 5$, then what you do is set $y = [0, 0, 0, 0, 1, 0, 0, 0, 0, 0]^T$ supposing that there are 10 classes.

So, the choice of the number of input units and number of output units is somewhat straightforward. And as for the number of hidden units and the number of hidden layers, a reasonable default is to use a single hidden layer and so the type of NN shown on the left of the above picture, with just one hidden layer, is probably the most common (N.T: this architecture with a single hidden layer is called in most of the literature *single-layer perceptron* or, simply, **perceptron**).

If you use more than one hidden layer, again the reasonable default will be to have the same number of hidden units in every single layer. That's what happens in the architectures shown in the previous figure, with two and three hidden layers of 5 units, respectively.

The network architecture on the left (percepton) is a perfectly able and reasonable *default*. And as for the number of hidden units, usually *the more hidden units the better*; it's just that if you have a lot of hidden units, it can become more computationally expensive to train the NN, but very often having more hidden units is a good thing.

And usually *the number of hidden units in each layer will be comparable to the number of features*, i.e. comparable to the dimension of x, or eventually, it could be anywhere from that number upto maybe twice of that or three or four times of that.

So, having the number of hidden units comparable, or a few times larger than the number of input features, is often a useful thing to do.

So, hopefully this gives you one reasonable set of default choices for defining the NN architecture, and if you follow these guidelines you will probably get something that works well, but in a later set of videos where I will talk specifically about advices for how to apply algorithms, I will actually say a lot more about how to choose the NN architecture. I actually have quite a lot I want to say later to make good choices for the number of hidden units, the number of hidden layers, and so on.

Next, here's what we need to implement in order to train the neural network. There are actually six steps. I have four on this slide and two more steps on the next slide.

Training a neural network

1. Randomly initialize weights
2. Implement forward propagation to get $h_\Theta(x^{(i)})$ for any $x^{(i)}$
3. Implement code to compute cost function $J(\Theta)$
4. Implement backprop to compute partial derivatives $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$

The first step (1) is to set up the neural network and to randomly initialize the values of the weights. And we usually initialize the weights to small values near zero.

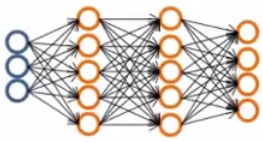
Then (2) we implement forward propagation so that we can input any training example, $x^{(i)}$, in the neural network and compute $h_\Theta(x^{(i)})$ which is this output vector, or the computed y values.

We then (3) also implement code to compute the cost function $J(\Theta)$. And next we implement the backpropagation algorithm to compute the partial derivatives terms with respect to the parameters, i.e. $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$.

Concretely, to implement the BPA usually we will do a `for` loop over the training examples:

```
for i = 1:m
```

Perform forward propagation and backpropagation using example $(x^{(i)}, y^{(i)})$
(Get activations $a^{(l)}$ and delta terms $\delta^{(l)}$ for $l = 2, \ldots, L$).



Some of you may have heard of frankly very advanced vectorization methods where you don't have a for loop over the `m` training examples $(x^{(i)}, y^{(i)})$, but the first time you're implementing BP there should almost certainly be the `for` loop in your code, where you're iterating over the examples $(x^{(i)}, y^{(i)})$, then so you do forward propagation and BP on the first example $(x^{(1)}, y^{(1)})$, and then in the second example $(x^{(2)}, y^{(2)})$, and so on, until you get through the final example $(x^{(m)}, y^{(m)})$.

So, there should be a `for` loop in your implementation of BP, at least the first time you're implementing it.

And then there are frankly somewhat complicated ways to do this without a `for` loop, but I definitely do not recommend trying to do that much more complicated version the first time you try to implement the BPA.

So, concretely, we have a `for` loop over the m training examples and inside the `for` loop we're going to perform forward propagation and backpropagation using each example $(x^{(i)}, y^{(i)})$. And what that means is that we're going to take $x^{(i)}$, and feed that to the input layer, perform forward propagation, perform backpropagation and that will give all of the activations $a^{(l)}$ and all $\delta^{(l)}$ for all of the layers $l = 2, \cdots, L$ of all my units in the neural network.

Then, still inside the `for` loop, we're going to compute the big delta terms

$$\Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)} \left(a^{(l)}\right)^T$$

which is the formula that we gave earlier.

And then, finally, outside the `for` loop, after having computed the $\Delta^{(l)}$ terms or accumulation terms, we would then have some other code to compute the partial derivative terms,

$$\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$$

and these partial derivative terms have to take into account the regularization parameter λ as well. And so, those formulas were already given in an earlier video.

So, having done that you now hopefully have code to compute these partial derivative terms.

Training a neural network

5. Use gradient checking to compare $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$ computed using backpropagation vs. using numerical estimate of gradient of $J(\Theta)$.
   Then disable gradient checking code.
6. Use gradient descent or advanced optimization method with backpropagation to try to minimize $J(\Theta)$ as a function of parameters Θ

Next, in step (5), what I do is use gradient checking to compare the partial derivative terms $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$ that were computed using BP versus the partial derivatives computed using the numerical estimates of the derivatives. So, I do gradient checking to make sure that both of these give you very similar values, and thus validate the BPA implementation.

Having done gradient checking just now reassures us that our implementation of backpropagation is correct, and is then very important that we disable gradient checking, because the gradient checking code is computationally very slow.

And finally (6), we then use an optimization algorithm such as gradient descent, or one of the advanced optimization methods such as BFGS, conjugate gradient or others which are embodied into `fminunc()` or other optimization methods. We use these together with BP, that computes the partial derivatives $\frac{\partial}{\partial \Theta_{jk}^{(l)}} J(\Theta)$ for us, and so we know how to compute the cost function, we know how to compute the partial derivatives using backpropagation, so we can use one of these optimization methods to try to minimize $J(\Theta)$ as a function of the parameters Θ.

And, by the way, **for neural networks the cost function $J(\Theta)$ is not convex**, and so it can theoretically be susceptible to stuck at local minima. And, in fact, algorithms like gradient descent and the advanced optimization methods can, in theory, get stuck in local optima, but it turns out that in practice this is not usually a huge problem and even though we can't guarantee that these algorithms will find a global optimum, usually algorithms like gradient descent will do a very good job minimizing the cost function $J(\Theta)$ and they get a very good local minimum Θ_{min}, even if they don't get the true global minimum.

Finally, gradient descent for a NN might still seem a little bit magical. So, let me just show one more figure (Fig. 40) to try to get that intuition about what gradient descent for a neural network is doing.

This is actually similar to the figure that I was using earlier to explain gradient descent. We have some cost function $J(\Theta)$, and we have a number of parameters in our NN, but here in Fig. 40 I've just written down two of the parameters, $\Theta_{11}^{(1)}$ and $\Theta_{12}^{(1)}$. In reality, of course, in the neural network we can have lots of parameters inside these matrices $\Theta^{(1)}$, $\Theta^{(2)}$, ..., right?

So we can have very high dimensional parameter matrices, but because of the limitations of what we can draw, I'm pretending that we have only two parameters in this NN although, obviously, we have a lot more in practice.

Now, the cost function $J(\Theta)$ measures how well the neural network fits the training data. So, if you take a point where $J(\Theta)$ is pretty low – say point (A) in Fig. 41, – this will correspond to a setting of the parameters $\Theta_{11}^{(1)}$ and $\Theta_{12}^{(1)}$ where, for most of the training examples, the output of my hypothesis may be pretty close to $y^{(i)}$

$$h_\Theta(x^{(i)}) \approx y^{(i)} \qquad \text{(near point (A) in the figure)}$$

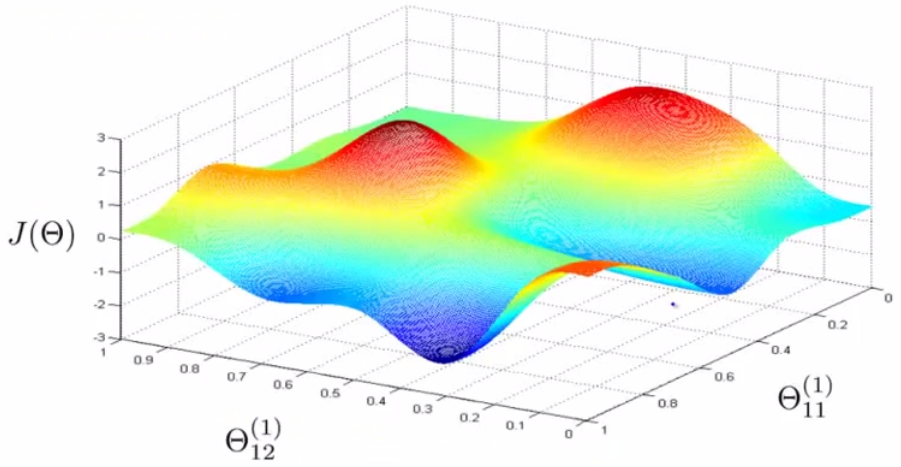


Figure 40: Cost function in a 3D plot.

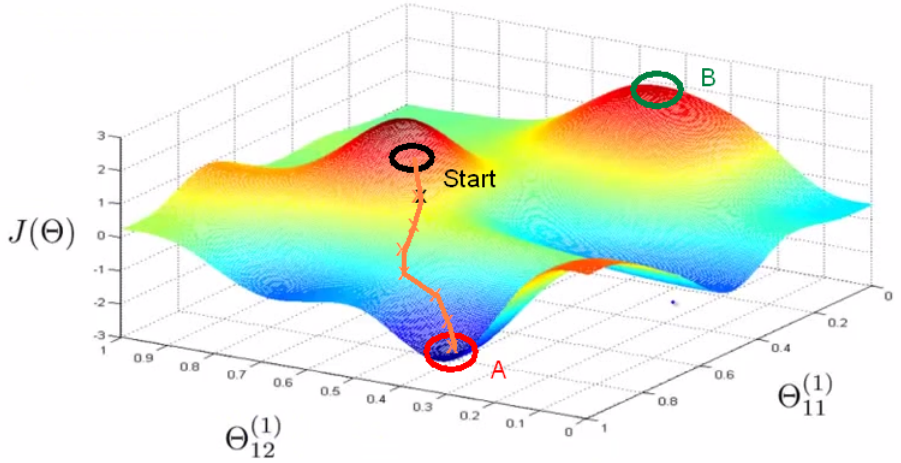


Figure 41: Important points in the landscape, and the path followed from the starting point (Start) until arriving at the (local?) optimum (A).

and if this is true, than that's what causes my cost function to be pretty low.

Whereas, in contrast, if you were to take a value like that in point (B), likely that corresponds to values of the parameters $\Theta_{11}^{(1)}$ and $\Theta_{12}^{(1)}$ where, for many training examples, the output of the NN is far from the actual value $y^{(i)}$ that was observed in the training set:

$$h_\Theta(x^{(i)}) \text{ very different from } y^{(i)} \qquad \text{(near point (B) in the figure)}$$

So, points like (B) correspond to where the hypothesis, i.e. where the NN, is outputting values on the training set that are far from $y^{(i)}$, so it's not fitting the training set well, whereas points like (A) with low values of the cost function correspond to where $J(\Theta)$ is low, and therefore corresponds to where the NN happens to be fitting the training set well, because this is what's needed to be true in order for $J(\Theta)$ to be small.

So, what gradient descent does is start from some random initial point (Start) in Fig. 41, and it will repeatedly go downhill. And so, what BP is doing is computing the direction of the gradient, and what gradient descent is doing is it's taking little steps downhill until hopefully it gets to, in this case, a pretty good local optimum in point labeled (A).

So, when you implement BP and use gradient descent or one of the advanced optimization methods, this picture in Fig. 41 sort of explains what the algorithm is doing. It's trying to find a value of the parameters where the output values in the neural network closely matches the values of the $y^{(i)}$ observed in the training set.

So, hopefully this gives you a better sense of how the many different pieces of NN learning fit together. In case, even after this video, you still feel like there are a lot of different pieces and it's not entirely clear what some of them do, or how all of these pieces come together, that's actually OK. NN learning and BPA is a complicated algorithm, and even though I've seen the math behind backpropagation for many years and I've used BP, I think very successfully, for many years, even today sometimes I still feel like I don't always have a great grasp of exactly what BP is doing, and what the optimization process of minimizing $J(\Theta)$ looks like.

This is a much harder algorithm to feel like I have a good handle on exactly what this is doing compared to say, linear regression or logistic regression. Which were mathematically and conceptually much simpler and much cleaner algorithms.

So, in case you feel the same way, that's actually perfectly OK. But if you do implement BP, hopefully what you find is that this is one of the most powerful learning algorithms and if you implement BP and one of these optimization methods, you find that backpropagation will be able to fit very complex, powerful, non-linear functions to your data.

And NN is one of the most effective learning algorithms we have today.

Figure 42: Control panel shown in a movie where an autonomous car relying in neural networks is seen in actual self driving.

15 Autonomous Driving

In this video, I'd like to show you a fun and historically important example of NN Learning, which is autonomous driving where a car learns to drive itself.

The video is something that I've gotten from Dean Pomerleau, a colleague from Carnegie Mellon University, and in part of the video you see visualizations like Fig. 42.

On the lower left is the view seen by the car, you will kind of see a road.

On top, the first horizontal bar shows the direction selected by the human driver and the bright white band shows the steering direction selected by the human driver, where far to the left corresponds to steering hard left and so on. In the picture the human driver is steering a bit to the left.

The second (lower) bar corresponds to the steering direction selected by the learning algorithm; the location of this sort of white band means the neural network was here selecting a steering direction just slightly to the left. And in fact, before the neural network starts learning initially, the network outputs a uniform grey band throughout the complete bar, so that the uniform grey color corresponds to the neural network having been randomly initialized and, initially, having no idea how to drive the car or what direction to steer in. And it's only after it's learned for a while that it will then start to output like a solid white band in just a small part of the region corresponding to choosing a particular steering direction. And that corresponds to when a neural network becomes more confident in selecting one steering direction.

Now, here is the video soundtrack.

"ALVINN is a system of artificial neural networks that learns to steer by watching a person drive. ALVINN is designed to control modified Army Humvee who could put sensors, computers and actuators for autonomous navigation experiments. The initial spec in configuring ALVINN is training. During training the person drives the vehicle while ALVINN watches. Once every two seconds, digitizes a video image of the road ahead and records the person's steering direction. This training image is reduced in resolution to 30×32 *pixels (lower left panel in Fig. 42) and is provided as input to ALVINN's three-layer network. Using the back propagation learning algorithm, ALVINN is training to output the same steering direction as the human driver for that image.*

Figure 43: ALVINN. Autonomous vehicle with Neural Network.

Figure 44: ALVINN in a common two-lane road.

Initially, the network's steering response is random. After about two minutes of training, the network learns to accurately imitate the steering reactions of the human driver. This same training procedure is repeated for other road types (Fig. 44). After the networks have been trained, the operator pushes the run switch and ALVINN begins driving itself. 12 times per second, ALVINN digitizes a road image and feeds it to its neural networks. Each network, running in parallel, produces a steering direction and a measure of it's confidence in its response (down-right panel in Fig. 42). The steering direction from the most confident network, in this case the network trained for the one-lane road, is used to control the vehicle.

Suddenly, an intersection appears ahead of the vehicle. As the vehicle approaches the intersection, the confidence of the one-lane network decreases. As it crosses the intersection, and the two-lane road ahead comes into view, the confidence of the two-lane network rises. When it's confidence rises, the two-lane network is selected to steer, safely guiding ALVINN into it's lane, on the two-lane road."

So that was autonomous driving using a neural network. Of course, there are more modern attempts to do autonomous driving in a few places in the U.S., in Europe, and so on. They're giving more robust driving controllers than ALVINN but I think it's still pretty remarkable, and pretty amazing, how a simple neural network trained with backpropagation can actually learn to drive a car somewhat well.