A Tutorial on Deep Learning
Part 1: Nonlinear Classifiers and The Backpropagation Algorithm

Quoc V. Le
qvl@google.com
Google Brain, Google Inc.
1600 Amphitheatre Pkwy, Mountain View, CA 94043

December 13, 2015

1 Introduction

In the past few years, Deep Learning has generated much excitement in Machine Learning and industry thanks to many breakthrough results in speech recognition, computer vision and text processing. So, what is Deep Learning?

For many researchers, Deep Learning is another name for a set of algorithms that use a *neural network* as an architecture. Even though neural networks have a long history, they became more successful in recent years due to the availability of inexpensive, parallel hardware (GPUs, computer clusters) and massive amounts of data.

In this tutorial, we will start with the concept of a linear classifier and use that to develop the concept of neural networks. I will present two key algorithms in learning with neural networks: the stochastic gradient descent algorithm and the backpropagation algorithm. Towards the end of the tutorial, I will explain some simple tricks and recent advances that improve neural networks and their training. For that, let's start with a simple example.

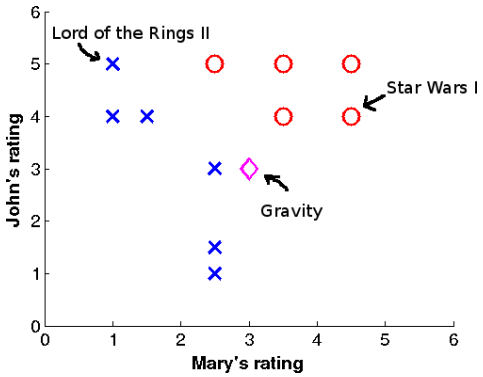
2 An example of movie recommendations

It's Friday night, and I am trying to decide whether I should watch the movie "Gravity" or not. I ask my close friends Mary and John, who watched the movie last night to hear their opinions about the movie. Both of them give the movie a rating of 3 in the scale between 1 to 5. Not outstanding but perhaps worth watching?

Given these ratings, it is difficult for me to decide if it is worth watching the movie, but thankfully, I have kept a table of their ratings for some movies in the past. For each movie, I also noted whether I liked the movie or not. Maybe I can use this data to decide if I should watch the movie. The data look like this:

Movie name	**Mary's rating**	**John's rating**	**I like?**
Lord of the Rings II	1	5	No
...	...	...	...
Star Wars I	4.5	4	Yes
Gravity	3	3	?

Let's visualize the data to see if there is any trend:



In the above figure, I represent each movie as a red "O" or a blue "X" which correspond to "I like the movie" and "I dislike the movie", respectively. The question is with the rating of (3, 3), will I like Gravity? Can I use the past data to come up with a sensible decision?

3 A bounded decision function

Let's write a computer program to answer this question. For every movie, we construct an example x which has two dimensions: the first dimension x_1 is Mary's rating and the second dimension x_2 is John's rating. Every past movie is also associated with a *label* y to indicate whether I like the movie or not. For now, let's say y is a scalar that should have one of the two values, 0 to mean "I do not like" or 1 to mean "I do like" the movie. Our goal is to come up with a *decision function* $h(x)$ to approximate y.

Our decision function can be as simple as a weighted linear combination of Mary's and John's ratings:

$$h(x;\theta,b) = \theta_1 x_1 + \theta_2 x_2 + b, \quad \text{which can also be written as} \quad h(x;\theta,b) = \theta^T x + b \tag{1}$$

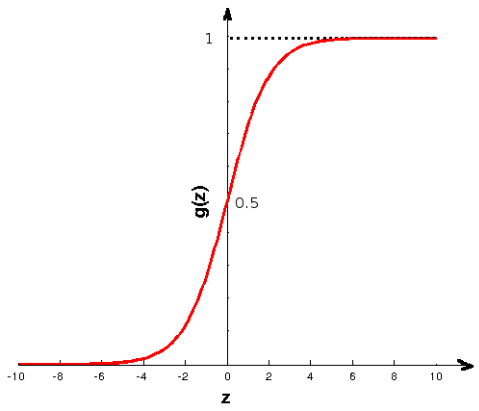
In the equation above, the value of function $h(x)$ depends on θ_1, θ_2 and b, hence I rewrite it as $h(x;(\theta_1,\theta_2),b)$ or in vector form $h(x;\theta,b)$.

The decision function h unfortunately has a problem: its values can be arbitrarily large or small. We wish its values to fall between 0 and 1 because those are the two extremes of y that we want to approximate.

A simple way to force h to have values between 0 and 1 is to map it through another function called the sigmoid function, which is bounded between 0 and 1:

$$h(x;\theta,b) = g(\theta^T x + b), \quad \text{where } g(z) = \frac{1}{1+\exp(-z)}, \tag{2}$$

which graphically should look like this:



The value of function h is now bounded between 0 and 1.

4 Using past data to learn the decision function

We will use the past data to learn θ, b to approximate y. In particular, we want to obtain θ, b such that:

$h(x^{(1)}; \theta, b) \approx y^{(1)}$, where $x^{(1)}$ is Mary's and John's ratings for 1st movie, "Lord of the Rings II"
$h(x^{(2)}; \theta, b) \approx y^{(2)}$, where $x^{(2)}$ is Mary's and John's ratings for 2nd movie
...
$h(x^{(m)}; \theta, b) \approx y^{(m)}$, where $x^{(m)}$ is Mary's and John's ratings for m-th movie

To find the values of θ and b we can try to minimize the following *objective function*, which is the sum of differences between the decision function h and the label y:

$$\begin{aligned} J(\theta, b) &= \left(h(x^{(1)}; \theta, b) - y^{(1)}\right)^2 + \left(h(x^{(2)}; \theta, b) - y^{(2)}\right)^2 + ... + \left(h(x^{(m)}; \theta, b) - y^{(m)}\right)^2 \\ &= \sum_{i=1}^{m} \left(h(x^{(i)}; \theta, b) - y^{(i)}\right)^2 \end{aligned}$$

5 Using stochastic gradient descent to minimize a function

To minimize the above function, we can iterate through the examples and slowly update the parameters θ and b in the direction of minimizing each of the small objective $\left(h(x^{(i)}; \theta, b) - y^{(i)}\right)^2$. Concretely, we can update the parameters in the following manner:

$$\theta_1 = \theta_1 - \alpha \Delta\theta_1 \tag{3}$$
$$\theta_2 = \theta_2 - \alpha \Delta\theta_2 \tag{4}$$
$$b = b - \alpha \Delta b \tag{5}$$

where α is a small non-negative scalar. A large α will give aggressive updates whereas a small α will give conservative updates. Such algorithm is known as *stochastic gradient descent (or SGD)* and α is known as the *learning rate*.

Now the question of finding the optimal parameters amounts to finding $\Delta\theta$'s and Δb such that they are in the descent direction. In the following, as our objective function is composed of function of functions, we use the *the chain rule* to compute the derivatives. Remember that the chain rule says that if g is a function of $z(x)$ then its derivative is as follows:

$$\frac{\partial g}{\partial x} = \frac{\partial g}{\partial z} \frac{\partial z}{\partial x}$$

This chain rule is very useful when taking the derivative of a function of functions.

Thanks to the chain rule, we know that a good descent direction for any objective function is its gradient. Therefore, at example $x^{(i)}$, we can compute the partial derivative:

$$\begin{aligned} \Delta\theta_1 &= \frac{\partial}{\partial \theta_1} \left(h(x^{(i)}; \theta, b) - y^{(i)}\right)^2 \\ &= 2\left(h(x^{(i)}; \theta, b) - y^{(i)}\right) \frac{\partial}{\partial \theta_1} h(x^{(i)}; \theta, b) \\ &= 2\left(g(\theta^T x^{(i)} + b) - y^{(i)}\right) \frac{\partial}{\partial \theta_1} g(\theta^T x^{(i)} + b) \end{aligned} \tag{6}$$

Apply the chain rule, and note that $\frac{\partial g}{\partial z} = [1 - g(z)]g(z)$, we have:

$$\begin{aligned}
\frac{\partial}{\partial \theta_1} g(\theta^T x^{(i)} + b) &= \frac{\partial g(\theta^T x^{(i)} + b)}{\partial (\theta^T x^{(i)} + b)} \frac{\partial (\theta^T x^{(i)} + b)}{\partial \theta_1} \\
&= \big[1 - g(\theta^T x^{(i)} + b)\big] g(\theta^T x^{(i)} + b) \frac{\partial (\theta_1 x_1^{(i)} + \theta_2 x_2^{(i)} + b)}{\partial \theta_1} \\
&= \big[1 - g(\theta^T x^{(i)} + b)\big] g(\theta^T x^{(i)} + b) x_1^{(i)}
\end{aligned}$$

Plug this to Equation 6, we have:

$$\Delta\theta_1 = 2\big[g(\theta^T x^{(i)} + b) - y^{(i)}\big]\big[1 - g(\theta^T x^{(i)} + b)\big] g(\theta^T x^{(i)} + b) x_1^{(i)} \tag{7}$$

where

$$g(\theta^T x^{(i)} + b) = \frac{1}{1 + \exp(-\theta^T x^{(i)} - b)} \tag{8}$$

Similar derivations should lead us to:

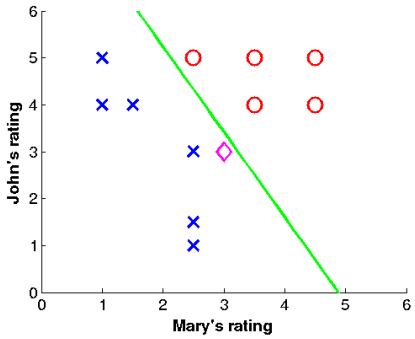
$$\Delta\theta_2 = 2\big[g(\theta^T x^{(i)} + b) - y^{(i)}\big]\big[1 - g(\theta^T x^{(i)} + b)\big] g(\theta^T x^{(i)} + b) x_2^{(i)} \tag{9}$$

$$\Delta b = 2\big[g(\theta^T x^{(i)} + b) - y^{(i)}\big]\big[1 - g(\theta^T x^{(i)} + b)\big] g(\theta^T x^{(i)} + b) \tag{10}$$

Now, we have the stochastic gradient descent algorithm to learn the decision function $h(x; \theta, b)$:

1. Initialize the parameters θ, b at random,
2. Pick a random example $\{x^{(i)}, y^{(i)}\}$,
3. Compute the partial derivatives θ_1, θ_2 and b by Equations 7, 9 and 10,
4. Update parameters using Equations 3, 4 and 5, then back to step 2.

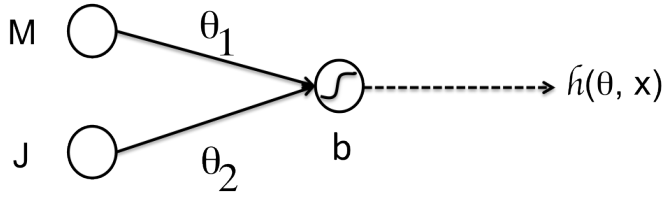
We can stop stochastic gradient descent when the parameters do not change or the number of iteration exceeds a certain upper bound. At convergence, we will obtain a function $h(x; \theta, b)$ which can be used to predict whether I like a new movie x or not: $h > 0.5$ means I will like the movie, otherwise I do not like the movie. The values of x's that cause $h(x; \theta, b)$ to be 0.5 is the "decision boundary." We can plot this "decision boundary" to have:



The green line is the "decision boundary." Any point lying above the decision boundary is a movie that I should watch, and any point lying below the decision boundary is a movie that I should not watch. With

this decision boundary, it seems that "Gravity" is slightly on the negative side, which means I should not watch it.

By the way, here is a graphical illustration of the decision function h we just built ("M" and "J" indicate the input data which is the ratings from Mary and John respectively):

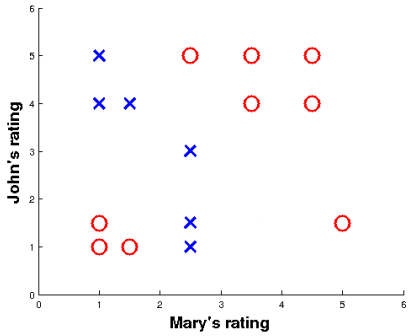


This network means that to compute the value of the decision function, we need the multiply Mary's rating with θ_1, John's rating with θ_2, then add two values and b, then apply the sigmoid function.

6 The limitations of linear decision function

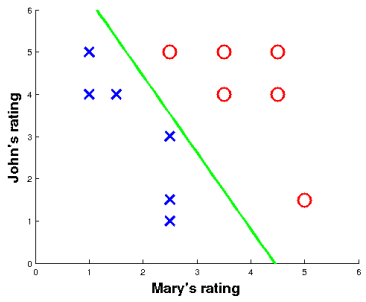
In the above case, I was lucky because the the examples are linearly separable: I can draw a linear decision function to separate the positive and the negative instances.

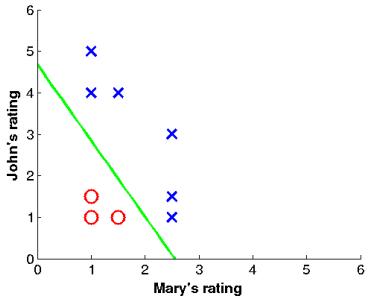
My friend Susan has different movie tastes. If we plot her data, the graph will look rather different:



Susan likes some of the movies that Mary and John rated poorly. The question is how we can come up with a decision function for Susan. From looking at the data, the decision function must be more complex than the decision we saw before.

My experience tells me that one way to solve a complex problem is to decompose it into smaller problems that we can solve. We know that if we throw away the "weird" examples from the bottom left corner of the figure, the problem is simple. Similarly, if we throw the "weird" examples on the top right figure, the problem is again also simple. In the figure below, I solve for each case using our algorithm and the decision functions look like this:





Is it possible to combine these two decision functions into one final decision function for the original data? The answer turns out to be yes and I'll show you how.

7 A decision function of decision functions

Let's suppose, as stated above, the two decision functions are $h_1(x; (\theta_1, \theta_2), b_1)$ and $h_2(x; (\theta_3, \theta_4), b_2)$. For every example $x^{(i)}$, we can then compute $h_1(x^{(i)}; (\theta_1, \theta_2), b_1)$ and $h_2(x^{(i)}; (\theta_3, \theta_4), b_2)$

If we lay out the data in a table, it would look like the first table that we saw:

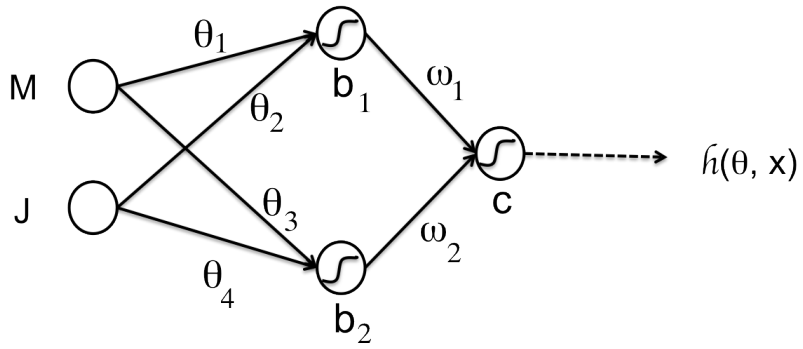
Movie name	**Output by decision function h_1**	**Output by decision function h_2**	**Susan likes?**
Lord of the Rings II	$h_1(x^{(1)})$	$h_2(x^{(2)})$	No
...	...	...	...
Star Wars I	$h_1(x^{(n)})$	$h_2(x^{(n)})$	Yes
Gravity	$h_1(x^{(n+1)})$	$h_2(x^{(n+1)})$	?

Now, once again, the problem becomes finding a new parameter set to weigh these two decision functions to approximate y. Let's call these parameters ω, c, and we want to find them such that $h((h_1(x), h_2(x)); \omega, c)$ can approximate the label y. This can be done, again, by stochastic gradient descent.

In summary, we can find the decision function for Susan by following two steps:

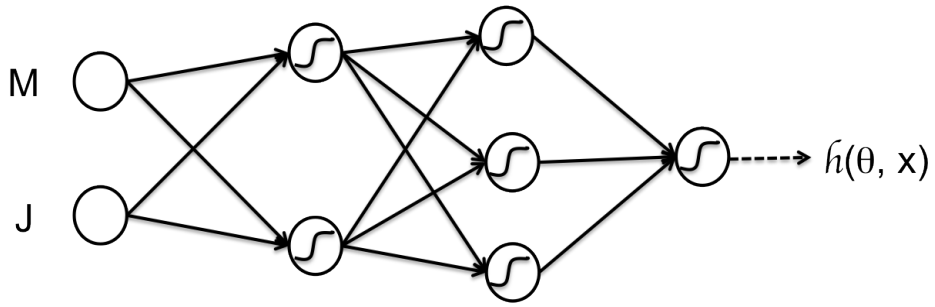
1. Partition the data into two sets. Each set can be simply classified by a linear decision. Then use the previous sections to find the decision function for each set,
2. Use the newly-found decision functions and compute the decision values for each example. Then treat these values as input to another decision function. Use stochastic gradient descent to find the final decision function.

A graphical way to visualize the above process is the following figure:



What you just saw is a special architecture in machine learning known as "neural networks." This instance of neural networks has one hidden layer, which has two "neurons." The first neuron computes values for function h_1 and the second neuron computes values for function h_2. The sigmoid function that maps real value to bounded values between 0, 1 is also known as "the nonlinearity" or the "activation function." Since we are using sigmoid, the activation function is also called "sigmoid activation function." In the future, you may encounter other kinds of activation functions. The parameters inside the network, such as θ, ω are called "weights" where as b, c are called "biases."

If you have a more complex function that you want to approximate, you may want to have a deeper network, maybe one that looks like this:



This network has two hidden layers. The first hidden layer has two neurons and the second hidden layer has three neurons.

Let's get back to our problem of finding a good decision function for Susan. It seems so far so good, but in the above steps, I cheated a little bit when I divided the dataset into two sets because I looked at the data and decided that the two sets should be partitioned that way. Is there any way that such a step can be automated?

It turns out the answer is also yes. And the way to do it is by not doing two steps sequentially, but rather, finding all parameters ω, c, θ, b at once on the complex dataset, using the stochastic gradient descent. To see this more clearly, let's write down how we will compute $h(x)$:

$$\begin{aligned} h(x) &= g\Big(\omega_1 h_1(x) + \omega_2 h_2(x) + c\Big) \\ &= g\Big(\omega_1 g\big(\theta_1 x_1 + \theta_2 x_2 + b_1\big) + \omega_2 g\big(\theta_3 x_1 + \theta_4 x_2 + b_2\big) + c\Big) \end{aligned}$$

We will find all these parameters $\omega_1, \omega_2, c, \theta_1, \theta_2, \theta_3, \theta_4, b_1, b_2$ at the same time.

Notice that the stochastic gradient descent is quite general: as long as we have a set of parameters, we can find the partial derivative at each coordinate and simply update one coordinate at a time. So the trick is to find the partial derivatives. For that, we need a famous algorithm commonly known as the backpropagation algorithm.

8 The backpropagation algorithm

The goal of the backpropagation algorithm is to compute the gradient (a vector of partial derivatives) of an objective function with respect to the parameters in a neural network. As the decision function $h(x)$ of the neural network is a function of functions, we need to use the chain rule to compute its gradient. The backpropagation algorithm is indeed an implementation of the chain rule specifically designed for neural networks. It takes some effort to arrive at this algorithm, so I will skip the derivation, and just show you the algorithm.

To begin, let's simplify the notations a little bit. We will use θ for all the weights in the network and b for all the biases. $\theta_{ij}^{(l)}$ means weight at layer l-th connecting neuron (or input) j-th to the neuron i-th in layer $l+1$, $b_i^{(l)}$ is bias of neuron i. The layers are indexed by $1(input), 2, ..., L(output)$. The number of neurons in the layer l is s_l. In this notation, the decision function $h(x)$ can be recursively computed as:

$$\begin{aligned} h^{(1)} &= x \\ h^{(2)} &= g\Big(\big(\theta^{(1)}\big)^T h^{(1)} + b^{(1)}\Big) \\ &... \end{aligned}$$

$$h^{(L-1)} = g\Big(\big(\theta^{(L-2)}\big)^T h^{(L-2)} + b^{(L-2)}\Big)$$
$$h(x) = h^{(L)} = g\Big(\big(\theta^{(L-1)}\big)^T h^{(L-1)} + b^{(L-1)}\Big) \qquad \text{[this is a scalar]}$$

We use matrix-vectorial notations so that it is easier to read and note that $h^{(l-1)}$ is a vector.

Here's the backpropagation algorithm. The steps are:

1. Perform a "feedforward pass," to compute $h^{(1)}, h^{(2)}, h^{(3)}, ..., h^{(L)}$.
2. For the output layer, compute
$$\delta_1^{(L)} = 2(h^{(L)} - y)\; g'\Big(\sum_{j=1}^{s_{L-1}} \theta_{1j}^{(L-1)} h_j^{(L-1)} + b_1^{(L-1)}\Big)$$
3. Perform a "backward pass," for $l = L-1, L-2, .., 2$
For each node i in layer l, compute
$$\delta_i^{(l)} = \Big(\sum_{j=1}^{s_{l+1}} \theta_{ji}^{(l)} \delta_j^{(l+1)}\Big)\; g'\Big(\sum_{j=1}^{s_{l-1}} \theta_{ij}^{(l-1)} h_j^{(l-1)} + b_i^{(l-1)}\Big)$$
4. The desired partial derivatives can be computed as
$$\Delta\theta_{ij}^{(l)} = h_j^{(l)} \delta_i^{(l+1)}$$
$$\Delta b_i^{(l)} = \delta_i^{(l+1)}$$

The indices make the algorithm look a little busy. But we can simplify these equations by using more matrix-vectorial notations. Before we proceed, let's use the notation $\odot$ for element-wise dot product. That is if b and c are vectors of n dimensions, then $b \odot c$ is a vector a of n dimensions where $a_i = b_i c_i, \forall i \in 1, ..., n$. Using this notation, the algorithm above can be rewritten in the following *vectorized* version:

1. Perform a "feedforward pass," to compute $h^{(1)}, h^{(2)}, h^{(3)}, ..., h^{(L)}$.
2. For the output layer, compute
$$\delta_1^{(L)} = 2\big(h^{(L)} - y\big) \odot g'\Big(\big(\theta^{(L-1)}\big)^T h^{(L-1)} + b^{(L-1)}\Big)$$
3. Perform a "backward pass," for $l = L-1, L-2, .., 2$
For each note i in layer l, compute
$$\delta^{(l)} = \Big(\big(\theta^{(l)}\big)^T \delta^{(l+1)}\Big) \odot g'\Big(\big(\theta^{(l-1)}\big)^T h^{(l-1)} + b^{(l-1)}\Big)$$
4. The desired partial derivatives can be computed as
$$\Delta\theta^{(l)} = \delta^{(l+1)}\big(h^{(l)}\big)^T \qquad [uv^T \text{ is also known as the cross product of } u \text{ and } v]$$
$$\Delta b^{(l)} = \delta^{(l+1)}$$

A nice property of the backpropagation algorithm is that it can be made efficient because in the feedforward pass, some of the intermediate values can be cached, and then used to compute the gradient in the backward pass.

9 Debug the backpropagation algorithm by numerical differentiation

The backpropagation algorithm can be difficult to implement and debug. A simple trick to debug the algorithm is to compare the partial derivative computed by backpropagation algorithm (known as the analytical derivative) and its numerical approximation.

Before we dive into the details of what the trick looks like, let's try to understand the idea of partial derivative.

If we have a function $f(x)$, the derivative of f at x is:

$$\frac{\partial f}{\partial x} = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

Let's suppose we have a function f which is so complicated that we cannot compute the analytical derivative by hand (maybe because we are not good at math!), the right hand side of the above equation can come in handy. It gives us a numerical approximation of the analytical derivative. Such approximation requires us to pick some values for ϵ, then evaluate the function at $x + \epsilon$ and x.

For example, if we want to compute the numerical approximation of partial derivative of $f(x) = x^3$ at $x = 2$, we can pick $\epsilon = 0.001$ and have:

$$\frac{f(2+0.001) - f(2)}{0.001} = \frac{f(2.001) - f(2)}{0.001} = \frac{2.001^3 - 2^3}{0.001} = 12.006$$

On the other hand, we also know that $\frac{\partial f}{\partial x} = 3x^2$, which is evaluated at $x = 2$ to have value of 12 which is quite close to the numerical approximation 12.006.

This idea can be generalized to function with many variables. As you can see above, our function J is a function of θ and b. To compute its derivative, we can randomly generate the parameters θ's and b's, then iterate through each parameter at a time, vary each value by ϵ.

For example, if we want to compute the numerical partial derivative of J at coordinate θ_i, we can do:

$$\frac{\partial J(\theta_1, \theta_2, ..., \theta_n, b)}{\partial \theta_i} = \frac{J(\theta_1, \theta_2, \theta_3, ..., \theta_{i-1}, \theta_i + 0.001, \theta_{i+1}, ..., \theta_n, b) - J(\theta_1, \theta_2, \theta_3, ..., \theta_{i-1}, \theta_i, \theta_{i+1}, ..., \theta_n, b)}{0.001}$$

A correct implementation of the backpropagation algorithm should give a gradient very similar to this approximation.

Finally, for stability, it is sometimes preferred to use the following formula to compute the numerical approximation:

$$\frac{\partial f}{\partial x} = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon}$$

In fact, we can try to compute the numerical derivative of $f(x) = x^3$ at $x = 2$ again

$$\frac{f(2+0.001) - f(2-0.001)}{2 \times 0.001} = \frac{f(2.001) - f(1.999)}{0.002} = \frac{2.001^3 - 1.999^3}{0.002} = 12.000$$

10 Some advice for implementing neural networks

- Make sure to check the correctness of your gradient computed by backpropagation by comparing it with the numerical approximation.
- It's important to "break the symmetry" of the neurons in the networks or, in other words, force the neurons to be different at the beginning. This means that it's important to initialize the parameters θ and b randomly. A good method for random initialization is Gaussian random or uniform random. Sometimes tuning the variance of the initialization also helps.

- Also make sure that the random initialization does not "saturate" the networks. This means that most of the time, for your data, the values of the neurons should be between 0.2 and 0.8. This is because we do not want to neurons to have too many values of zeros and ones. When that happens, the gradient is small and thus the training is much longer.

- Have a way to monitor the progress of your training. Perhaps the best method is to compute the objective function J on the current example or on a subset of the training data or on a held-out set.

- Picking a good learning rate α can be tricky. A large learning rate can change the parameters too aggressively or a small learning rate can change the parameters too conservatively. Both should be avoided, a good learning rate is one that leads to good overall improvements in the objective function J. To select good α, it's also best to monitor the progress of your training. In many cases, a learning rate of 0.1 or 0.01 is a very good start.

- Picking good hyperparameters (architectural parameters such as number of layers, number of neurons on each layers) for your neural networks can be difficult and is a topic of current research. A standard way to pick architectural parameters is via cross-validation: Keep a hold-out validation set that the training never touches. If the method performs well on the training data but not the validation set, then the model *overfits*: it has too many degrees of freedom and remembers the training cases but does not generalize to new cases. If the model overfits, we need to reduce the number of hidden layers or number of neurons on each hidden layer. If the method performs badly on the training set then the model *underfits*: it does not have enough degrees of freedom and we should increase the number of hidden layers or number of neurons. We will also talk about overfitting in Section 16. Be warned that bad performance on the training set can also mean that the learning rate is chosen poorly.

- Picking good hyperparameters can also be automated using grid search, random search or Bayesian optimization. In grid search, every possible combination of hyperparameters will be tried and cross-validated with a hold-out validation set. In case that grid search is expensive because the number of hyperparameters is large, one can try random search where hyperparameter configurations are generated and tried at random. Bayesian optimization looks at the performances of networks at previous hyperparameter combinations and fits a function through these points, it then picks the next combination that maximizes some utility function such as the mean plus some function of the uncertainty (e.g., [19]).

- Neural networks can take a long time to train and thus it's worth spending time to optimize the code for speed. To speed up the training, make use of fast matrix vector libraries, which often provide good speed-up over naïve implementation of matrix vector multiplication. The vectorized version of the backpropagation algorithm will come in handy in this case.

- It is possible to use single precision for the parameters in the networks instead of double precision. This reduces the memory footprint of the model in half and usually does not hurt the performances of the networks. A downside is that it is more tricky to check the correctness of the gradient using the numerical approximation.

- What is inconvenient about neural networks is that the objective function is usually *non-convex* with respect to the parameters. This means that if we obtain a minimum, it's likely to be a local minimum and may not be global minimum. Neural networks are therefore sensitive to random initialization. Other randomization aspects of the learning process could also affect the results. For example, factors such as the choice of learning rate, the order of examples we iterate through can produce different optimal parameters for different learning trials.

- It is possible to run stochastic gradient descent where every step we touch more than one example. This is called minibatch stochastic gradient descent and the number of examples we look at per

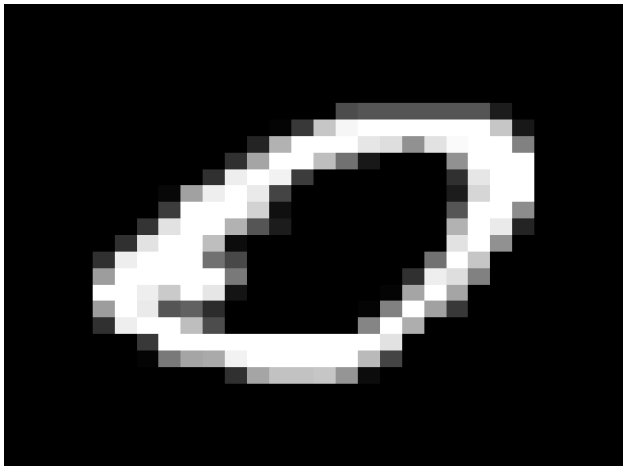
iterations is called the "minibatch size." (When the minibatch size is 1, we recover stochastic gradient descent.) In many environments, using a larger minibatch can be a good idea because the gradient is less noisy (computed as an average over the examples) and faster because matrix-vector libraries work better with larger matrices.

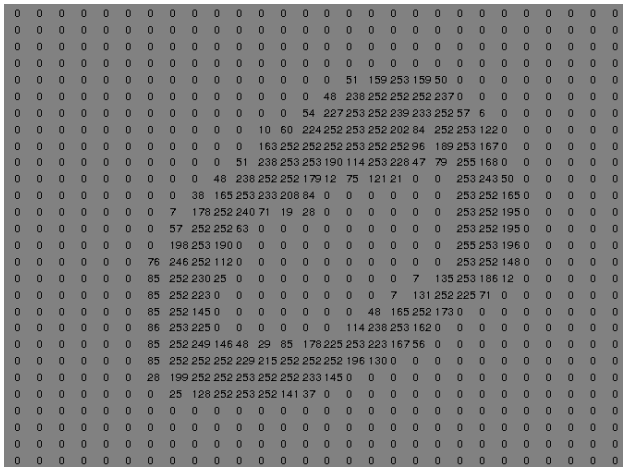
11 What problems are neural networks good for?

As you can see so far, neural networks are general nonlinear classifiers. Neural networks are flexible, and we can make a lot of changes to them to solve other problems. But as far as classifying data goes, are neural networks good? And what are they good for?

Many experiments have shown that neural networks are particularly good with natural data (speech, vision, language) which exhibit highly nonlinear properties.

Let's take an example of computer vision where the task is to recognize the digit from a handwritten input image. Below I visualize an image of the digit 0 written by my friend on the left. On the right side, I visualize the typical representations that computer scientists use as inputs to machines:





As can be seen from the figure, the input representations for machines are typically a 2D map of pixel values (or a matrix of pixel values). Each pixel has value between 0 and 255. The higher the value the brighter the pixel is. So the image on the left is represented on the right as mostly zeros, except from the center part where the pixel values have nonzero values.

To convert this matrix to an input vector and present that to neural networks, a standard approach in machine learning is to concatenate all the rows of the matrix and "straighten" that into a vector. A consequence of such representation is that if the digit gets shifted to the left, or to the right, the difference of the shifted digit to the original digit is very large. But to us, they are just the same digit and should belong to the same category (we call this *translational invariance*). At the same time, if I place the digit 6 at exactly the same location and with exactly the same shape, the difference between the digit 6 and the digit 0 is rather small.

More concretely, let's suppose I have the following three images, each has 28 x 28 pixels:

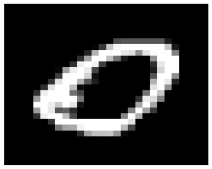


Image1


Image2

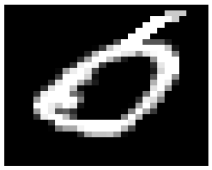


Image3

For us, $Image1$ and $Image2$ should have the same label of 0 whereas $Image3$ should have the label of 6. But let's check the distances between them. First, the (Euclidean) distance between $Image1$ and $Image2$ is:

$$distance(Image1, Image2) = \sqrt{\sum_{i=1}^{28}\sum_{j=1}^{28}\Big(Image1(i,j) - Image2(i,j)\Big)^2} \approx 3 \times 10^3 \tag{11}$$

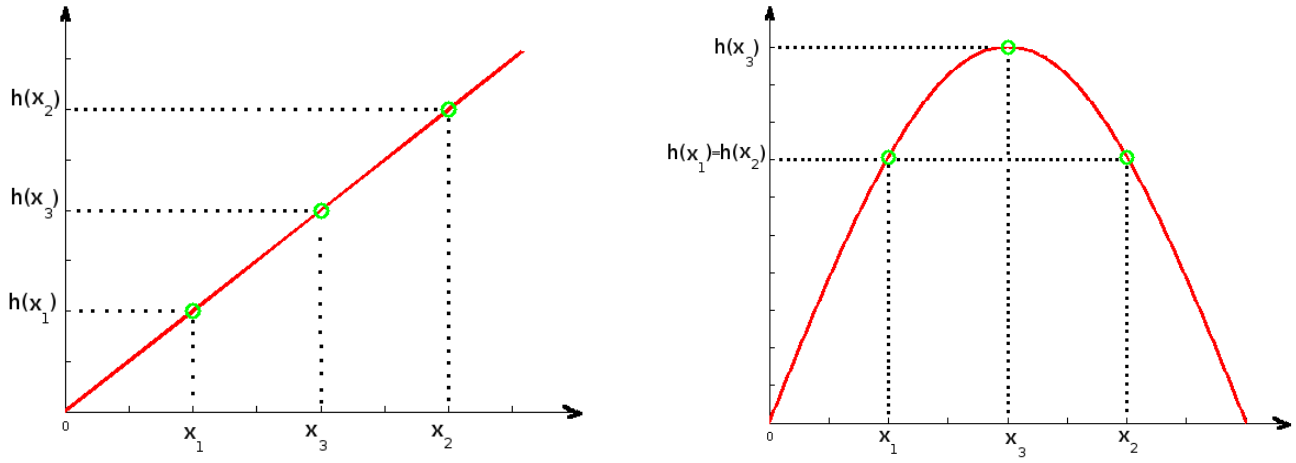
While the distance between $Image1$ and $Image3$ is:

$$distance(Image1, Image3) = \sqrt{\sum_{i=1}^{28}\sum_{j=1}^{28}\Big(Image1(i,j) - Image3(i,j)\Big)^2} \approx 10^3 \tag{12}$$

So $distance(Image1, Image3) < distance(Image1, Image2)$. This means that $Image1$ is closer to $Image3$ than $Image2$, even though we want the label for $Image1$ to be the same with $Image2$ and different from the label for $Image3$. Practically, we want a decision function h that $h(Image1) = h(Image2) = 0 \neq h(Image3) = 6$.

Such property is difficult to achieve, if not impossible, with a linear decision function. For the purpose of visualization, let's take an example of a one dimensional task and suppose we have $x_1 = 2, x_2 = 6, x_3 = 4$, ($x_1$ is closer to x_3 than x_2), and we want a decision that have the property that $h(x_1) = h(x_2) \neq h(x_3)$.

It's not possible to construct a linear function $h(x) = ax + b$ to achieve such property yet it's possible to do so with a nonlinear function. The figure below shows such property. On the right, we show an example of a nonlinear function which can achieve the property that $h(x_1) = h(x_2) \neq h(x_3)$, something that a linear function would fail to capture (left):



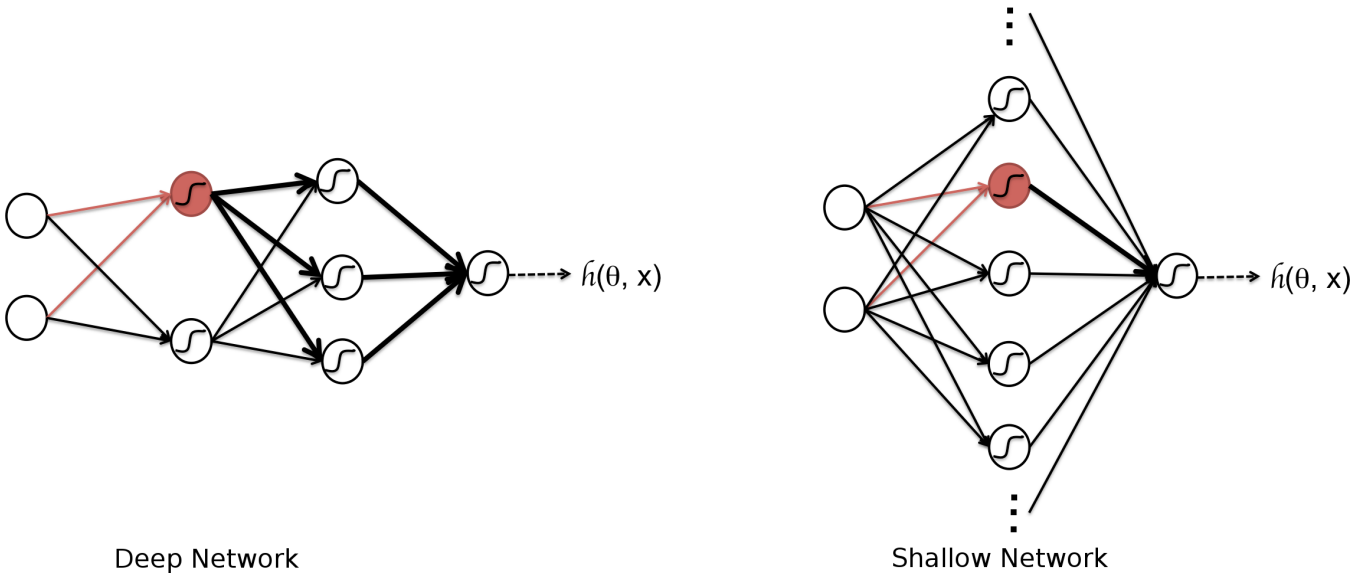
Interestingly, this property of having nonlinear decision functions arises very often in natural data, such as vision and speech. Perhaps this is because natural inputs are generated under adversarial conditions: we lack of energy (food!) to create hardware (eyes, cameras, robots) that perfectly aligns the images, or sounds and our preys/predators avoid our attention by changing colors etc.

But not every problem requires nonlinear decision function. A good way to test is to compare the distances between a subset of the training set with their categories. Unless the problem has nearby examples with different labels and far-away examples with the same label, you may not need a nonlinear decision function.

12 Deep vs. shallow networks

When the problem does exhibit nonlinear properties, deep networks seem computationally more attractive than shallow networks. For example, it has been observed empirically that in order to get to the same level of performances of a deep network, one has to use a shallow network with many more connections (e.g., 10x number of connections in speech recognition [1, 8]). It is thus much more expensive to compute the decision function for these shallow networks than the deep network equivalences because for every connection we need to perform a floating-point operation (multiplication or addition).

An intuition of why this is the case is as follows. A deep network can be thought of as a program in which the functions computed by the lower-layered neurons can be thought of as subroutines. These subroutines are re-used many times in the computation of the final program. For example in the following figure, the function computed by the red neuron in the first layer is re-used three times in the computation of the final function h. In contrast, in the shallow network, the function computed by the red neuron is only used once:



(Bolded edges mean computation paths that need the red neuron to produce the final output.)

Therefore, using a shallow network is similar to writing a program without the ability of calling subroutines. Without this ability, at any place we could otherwise call the subroutine, we need to explicitly write the code for the subroutine. In terms of the number of lines of code, the program for a shallow network is therefore longer than a deep network. Worse, the execution time is also longer because the computation of subroutines is not properly re-used.

A more formal argument of why deep networks are more "compact" than shallow counterparts can be found in Chapter 2 of [2].

13 Deep networks vs. Kernel methods

Another interesting comparison is deep networks vs. *kernel methods* [4, 18]. A kernel machine can be thought of as a shallow network having a huge hidden layer. The advantage of having a huge number of neurons is that the collection of neurons can act as a database and therefore can represent highly nonlinear functions. The beauty of kernel methods lies in the fact that even though the hidden layer can be huge, its computation can be avoided by the *kernel trick*. To make use of the kernel trick, an algorithm designer would rewrite the optimization algorithm (such as stochastic gradient descent) in such a way that the hidden layer always appears in a dot product with the hidden layer of another example: $< \phi(x), \phi(x') >$. The

dot product of two representations, with certain activation functions, can be computed in an inexpensive manner.

Kernel methods however can be expensive in practice. This is because the these methods keep around a list of salient examples (usually near the boundary of the decision function) known as "support vectors." The problem is that the number of support vectors can grow as the size of the training set grows. So effectively, the computation of the decision function h can be large for large datasets. The computation of the decision function in neural networks, on the other hand, only depends on how many connections in the neural networks and does not depend on the size of the training set.

As a side note, more recently, researchers have tried to avoid the *kernel trick* and revisited the idea of using a extremely large number of neurons and representing them explicitly. Interestingly, it can be proved that even with random weights, the hidden layer is already powerful: it can represent highly nonlinear functions. This idea is also known as Random Kitchen Sinks [15]. In making the connections random, these methods can enjoy faster training time because we do not have to train the weights of the hidden layer. The objective function is also *convex* and thus the optimization is not sensitive to weight initialization. But this only solves half of the problem because the computation of the decision function is still expensive.

14 A brief history of deep learning

The field of artificial neural networks has a long history, dated back to 1950's. Perhaps the earliest example of artificial neural networks is the Perceptron algorithm developed by Rosenblatt in 1957 [16]. In the late 1970's, researchers discovered that Perceptron cannot approximate many nonlinear decision functions, for example the XOR function. In 1980's, researchers found a solution to that problem by stacking multiple layers of linear classifiers (hence the name "multilayer perceptron") to approximate nonlinear decision functions. Neural networks again took off for a while but due to many reasons, e.g., the lack of computational power and labeled data etc., neural networks were left out of mainstream research in late 1990's and early 2000's.

Since the late 2000's, neural networks have recovered and become more successful thanks to the availability of inexpensive, parallel hardware (graphics processors, computer clusters) and a massive amount of labeled data. There are also new algorithms that make use of unlabeled data and achieve impressive improvements in various settings, but it can be argued that the core is almost the same with old architectures of 1990's, which is what you have seen in this tutorial. Key results are when the networks are deep: in speech recognition (e.g., [6]), computer vision (e.g., [3, 9]), and language modeling (e.g., [12]). And thus the field is also associated with the name "Deep Learning."

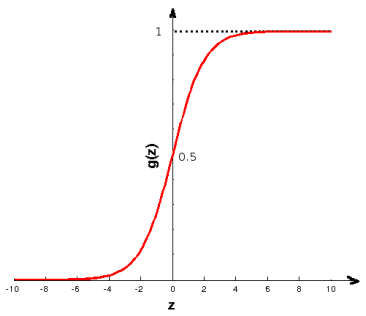
There are many reasons for such success. Perhaps the most important reason is that neural networks have a lot of parameters, and can approximate very nonlinear functions. So if the problem is complex, and has a lot of data, neural networks are good approximators for it. The second reason is that neural networks are very flexible: we can change the architecture fairly easily to adapt to specific problems/domains (such as convolutional neural networks and recurrent neural networks, which are the topics of the next tutorial).

The name Deep Learning can mean different things for different people. For many researchers, the word "Deep" in "Deep Learning" means that the neural network has more than 2 layers. This definition reflects the fact that successful neural networks in speech and vision are both deeper than 2 layers. For many other researchers, the word "Deep" is also associated with the fact that the model makes use of unlabeled data. For many people that I talk to in the industry, the word "Deep" means that there is no need for human-invented features. But for me, "Deep Learning" means a set of algorithms that use neural networks as an architecture, and learn the features automatically.

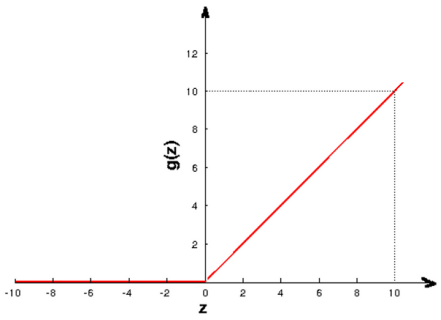
From 2006 to mid 2014, there have been several clever algorithmic ideas which, in certain cases, improve the performances of deep networks. Among these are the use of rectified linear units and dropout, which will be discussed in the following sections. These two algorithms address two core aspects of deep learning: ease, speed of training (optimization) and prediction quality (generalization), respectively.

15 Rectified linear as a better activation function

One of the recent advances in neural networks is to use the rectified linear units (a.k.a. ReLUs, $g(z) = \max(0, z)$) as an activation function in place of the traditional sigmoid function [13]:



The sigmoid activation function



The rectified linear activation function

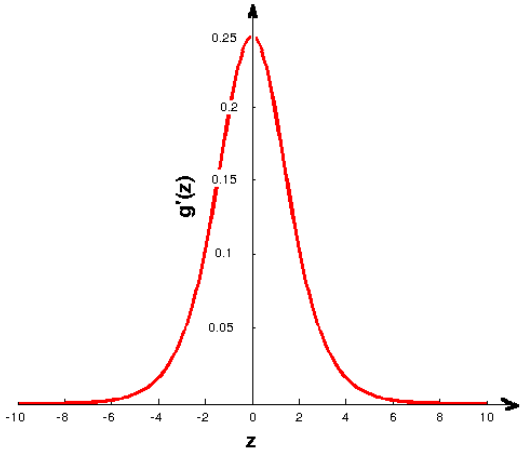
The change from sigmoid to ReLUs as an activation function in a hidden layer is possible because the hidden neurons need not to have bounded values. It has been observed empirically that this activation function allows for better approximation quality: the objective function is lower on the training set. It is therefore recommended to use rectified linear units instead of sigmoid function in your neural networks.

The reason for why rectified linear units work better than sigmoid is still an open question for research, but maybe the following argument can give an intuition.

First, notice that in the backpropagation algorithm to compute the gradient for the parameters in one layer, we typically multiply the gradient from the layer above with the partial derivative of the sigmoid function:

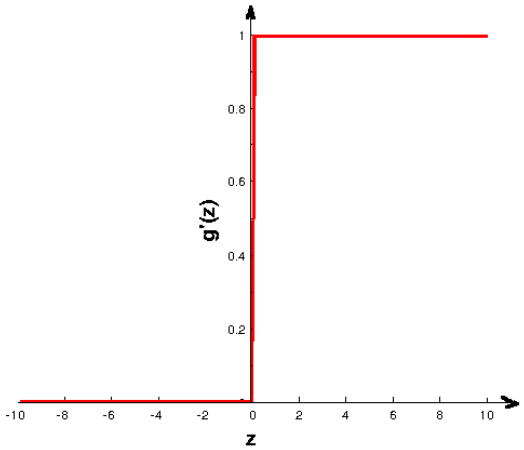
$$\delta^{(l)} = \left(\left(\theta^{(l)}\right)^T \delta^{(l+1)} \right) \odot g' \left(\left(\theta^{(l-1)}\right)^T h^{(l-1)} + b^{(l-1)} \right)$$

The issue is that the derivative of the sigmoid function has very small values (near zero) everywhere except for when the input has values of 0, as the figure below suggests:



which means that the lower layers will likely to have smaller gradients in terms of magnitude, compared to the higher layers. This is because $g'(.)$ is always less than 1, with most values being 0. This imbalance in gradient magnitude makes it difficult to change the parameters of the neural networks with stochastic gradient descent. A large learning rate is suitable for lower layers but not suitable for higher layers, and causes great fluctuations in the objective function. A small learning rate is suitable for lower layers but not suitable for higher layers, and causes small changes in the parameters in higher layers.

This problem can be addressed by the use of rectified linear activation function. This is because the derivative of the rectified linear activation function can have many nonzero values:



which in turn means that the magnitude of the gradient is more balanced throughout the networks.

The above argument is consistent with a recent empirical result of using an improved version of ReLU called PReLU (i.e., $\max(x, ax)$ – to obtain better results on ImageNet), where the derivative of the activation function is nonzero except for one point (see [5]).

16 Dropout to improve generalization

One problem that many of us encounter in training neural networks is that neural networks tend to overfit, especially when the training set is small. Overfitting means that the performance on the training set is much better than the performance on the test set. In other words, the model "remembers" the training set but does not generalize to new cases.

To overcome overfitting, there are many strategies:

- Use unlabeled data to train a different network known as an autoencoder and then use the weights to initialize our network (I will discuss this "pretraining" idea using autoencoders in the next tutorial),
- Penalize the weights by adding a penalty on the norm of the weights in the network to the objective function J, e.g., using $J + \|\theta\|_2^2$ as the final objective function,
- Use dropout [7].

To use dropout, at every iteration of stochastic gradient descent, we select randomly a fraction of neurons in each layer and drop them out of the optimization by setting their values to zero. At test time, we do not drop out any neurons but have to scale the weights properly. Concretely, let's say the neurons at layer l are dropped out with probability p ($p = 0.5$ meaning that we drop half of the neurons at an iteration), then at test time the *incoming* weights to the layer l should be scaled by p: $\theta_{test}^{(l-1)} = p\theta^{(l-1)}$.

It is probably easy to see why dropout should work. By dropping a fraction of the neurons out, the algorithm forces the decision function to collect more evidence from other neurons rather than fitting to a particular phenomenon. The decision function is therefore more robust to noise.

17 Recommended readings

This tutorial is inspired by Andrew Ng's tutorial on autoencoders [14]. Some advice in this tutorials is inspired by Yann LeCun et. al. [10].

Backpropagation was probably invented earlier but gained most attention through the work of Werbos [20] and especially Rumelhart et. al. [17].

The concept of an artificial neuron, a computational unit that multiplies some stored parameters with inputs, adds some bias then applies some threshold logic function, was perhaps first invented by McCulloch and Pitts [11]. It is also known as the McCulloch-Pitts' neuron models (or MCP neuron).

18 Miscellaneous

This tutorial was written as preparation materials for the machine learning summer school at CMU (MLSS'14). Videos of the lectures are at `http://tinyurl.com/pduxz2z` . An exercise associated with this tutorial is at `http://ai.stanford.edu/~quocle/exercise1.py` .

If you find bugs with this tutorial, please send them to me at qvl@google.com .

19 Acknowledgements

I would like to thank Jeff Dean, Coline Devin, Josh Levenberg, Thai Pham and members of the Google Brain team for many insightful comments and suggestions. I am also grateful to many readers who gave various comments and corrections to the tutorial.

References

[1] L. Ba and R. Caurana. Do deep nets really need to be deep? *arXiv preprint arXiv:1312.6184*, 2013.

[2] Y. Bengio. Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–127, 2009.

[3] D. C. Ciresan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In *International Joint Conference on Artificial Intelligence*, 2011.

[4] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[5] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *arXiv preprint arXiv:1502.01852*, 2015.

[6] G. Hinton, L. Deng, D. Yu, G. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal Processing Magazine*, 2012.

[7] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[8] P. S. Huang, H. Avron, T. Sainath, V. Sindhwani, and B. Ramabhadran. Kernel methods match deep neural networks on timit. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 1, page 6, 2014.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, 2012.

[10] Y. LeCun, L. Bottou, G. Orr, and K. R. Müller. Efficient backprop. In *Neural networks: Tricks of the trade*. Springer, 1998.

[11] W. McCulloch and W. Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5(4), 1943.

[12] T. Mikolov. *Statistical Language Models based on Neural Networks*. PhD thesis, Brno University of Technology, 2012.

[13] V. Nair and G. Hinton. Rectified Linear Units improve Restricted Boltzmann Machines. In *International Conference on Machine Learning*, 2010.

[14] A. Ng. CS294A lecture notes – sparse autoencoder. `http://web.stanford.edu/class/cs294a/sparseAutoencoder_2011new.pdf`, 2011. [Online; accessed 15-July-2014].

[15] A. Rahimi and B. Recht. Weighted sums of random kitchen sinks: Replacing minimization with randomization in learning. In *Advances in Neural Information Processing Systems*, 2009.

[16] F. Rosenblatt. *The Perceptron – a perceiving and recognizing automaton*. Number 85-460-1. Cornell Aeronautical Laboratory, 1957.

[17] D. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[18] B. Schölkopf and A. J. Smola. *Learning with kernels: Support vector machines, regularization, optimization, and beyond*. the MIT Press, 2002.

[19] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, 2012.

[20] P. J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, 1974.