Deep Learning

by DeepLearning.AI

Lucas Paiolla Forastiere

September 6, 2021

Contents

Chapter 1

Introduction

The term deep learning refers to training *neural networks*, sometimes very big neural networks. But what are neural networks?

So let's suppose we want to predict housing prices based on the size of the house. And let's say we'll use Logistic Regression to do that. But as we know, house prices can't be negative, so we simply say the value of the house is 0 if the Logistic Regression would predict something negative.

That's indeed the simplest neural network we can have, we have a single input `size` and a single output `price` and in the middle we have a single neuron: the logistic regression.

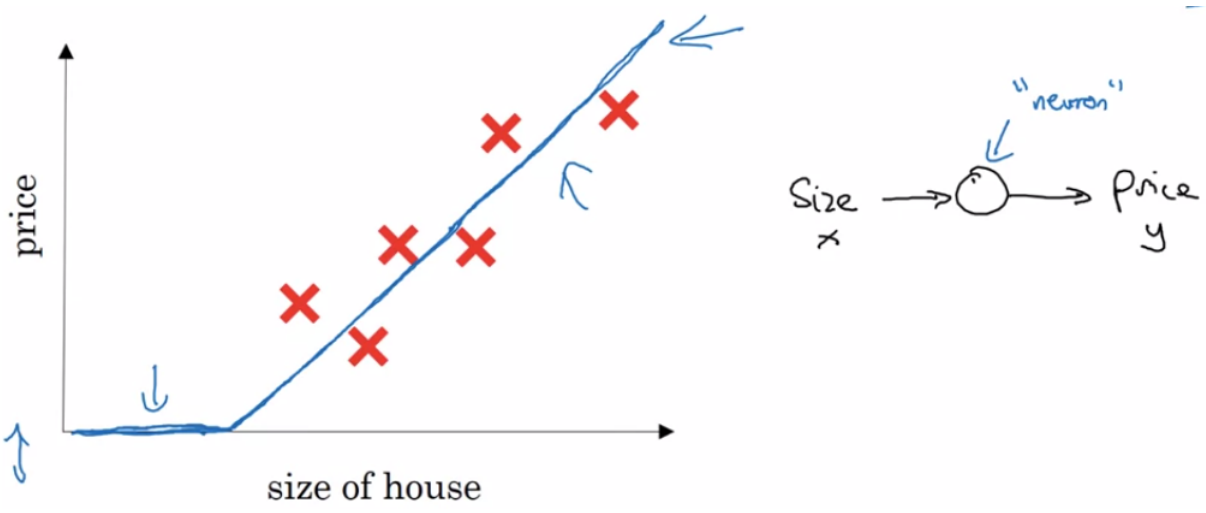


Figure 1.1: Here we see the graph of the problem we described.

That function which is zero and than linear is called *ReLU* and it's used a lot in neural networks. It stands for *Rectified Linear Unit*.

So to get a bigger neural network, we stack these neurons. Instead of predicting using only the size of house, we could use the number of bedrooms, zip code and wealth. We could use the size and number of bedrooms to predict the family size; use the zip code to predict the walkability; and use the zip code and wealth to predict the school quality. And then, we could use the family size, walkability and school quality to predict the price. See in the picture:

However, in general what we have is something a little more complex than that. We would have something like figure 1.3. Here we see that the internal nodes (which are called **hidden nodes** or **hidden neurons** or **hidden units**) receive the output of all the

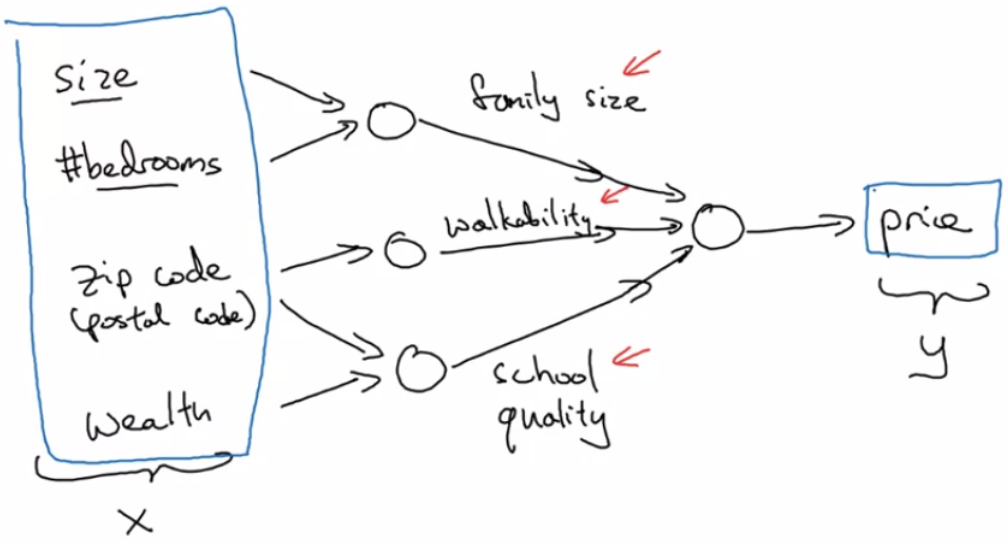


Figure 1.2: Now we have a more complex neural network, which is the stack of many ReLUs.

previous nodes to make it predictions. These hidden nodes don't really have a meaning like the example we gave. We don't try to predict family size or walkability or whatever, we simply let the neural network decide what that neuron will output in order to predict the final output `price` in the better way it can.

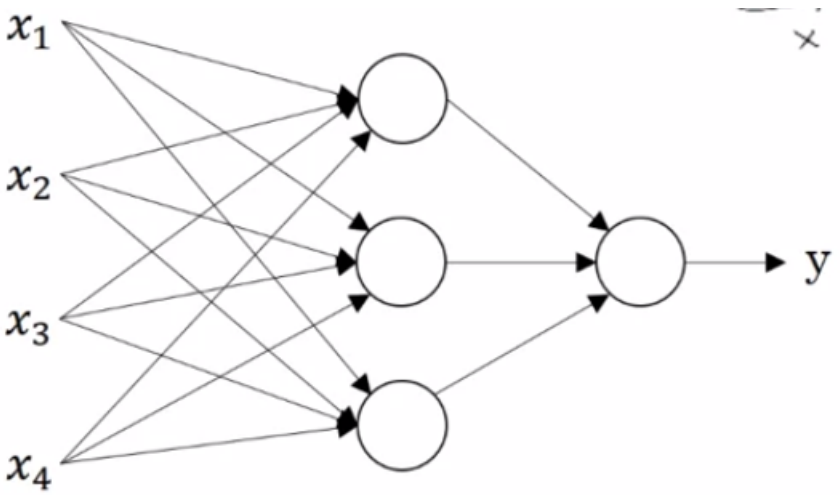


Figure 1.3: The generic form of a neural network.

We can use neural networks in many applications, here we're going to foucos in **supervised learning**, which are problems that you have a set of variables called input (represented by x) and an output (y) related to that input. In order to solve these kind of problems, there are many kinds of neural networks. The one we saw is the most common one, but there are others, like convolutional nn or recurrent nn.

Another thing that's important to decide what kind of nn we'll use is knowing if the data we're leading with is *structured* or *unstructured*.

Structured data is data in the form of a table. We have a very clear set of input variable X and a set of output variables y. Each line of our table represents one instance

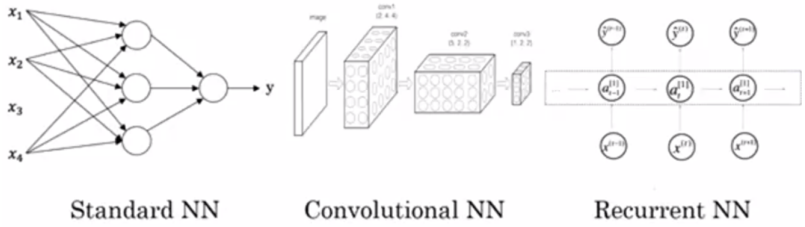


Figure 1.4: Examples of neural networks.

of data with many inputs and one or more outputs related to those inputs.

Unstructured data is all the other kinds of data: audio, video, texts, images, etc.

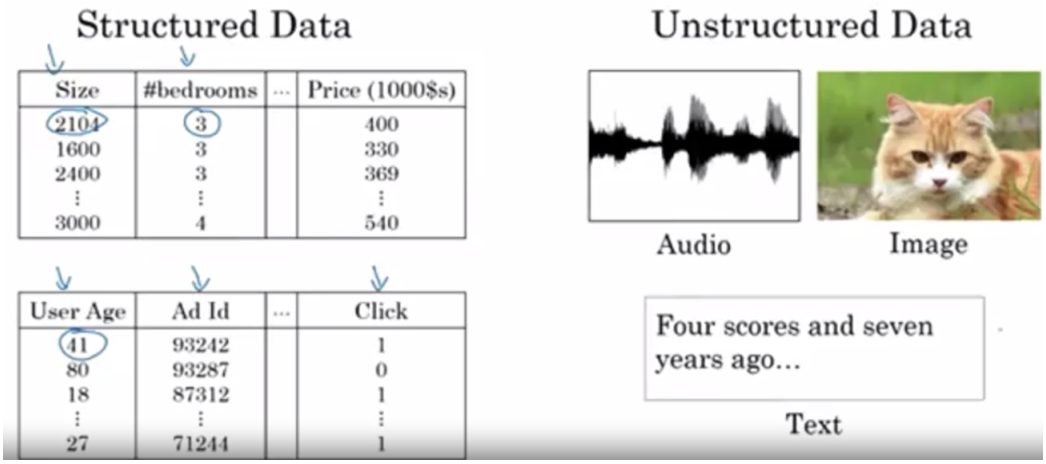


Figure 1.5: The two kinds of data.

It turns out that machine learning algorithms performed better on structured data over the years and more recently neural networks are performing better also on unstructured data.

Why is Deep Learning taking off? This is one of the questions we must ask ourselves when begining to learn deep learning. Let's see the graph of the performance of the machine learning algorithms versus the amount of data that we provide to then. We see that traditional learning algorithms have a plato where they can't improve anymore, which neural networks can lead with that data as we make than bigger and bigger.

We also see in the graph that when we don't have a large amount of data, NNs all algorithms perform pretty much the same.

So in order to answer our question, we have to understand the evolution of three things: *data*, *computation* and *algorithms*.

Through the years, the amount of data available was inscreased a lot, so NNs can take advantage from that. Also the computation power was inscreased with the use of GPUs to make a large amount of computations. And finally new algorithms have been developed to make NNs faster. That's the main reason why deep learning is taking off.

1.1 Notation

Before continuing, we need to define the notation we're going to use.

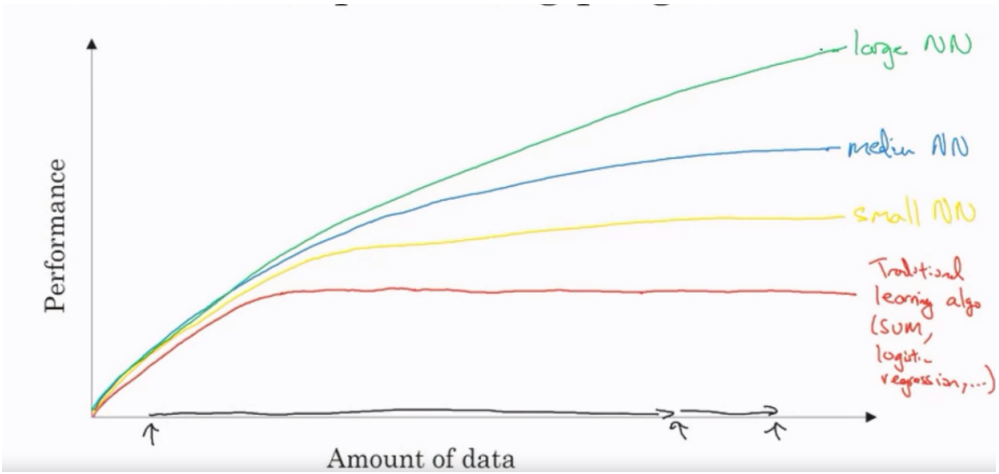


Figure 1.6: The performance of machine learning algorithms in respect to the data we provide to them.

- (x, y) will denote a single training input;
- m or m_{train} denotes the number of training examples;
- $x^{(i)}$ denotes the i-th training input;
- $y^{(i)}$ denotes the i-th traning output;
- n_x or n denotes the number of dimensions x has (or the number of features);
- m_{test} denotes the number of testing examples;
- X is the matrix of all traning examples. It's defined as:

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$$

  X is an $m \times n$ matrix;

- Y is the matrix of all outputs. It's defined as:

$$Y = \begin{bmatrix} y^{(1)} & y^{(2)} & \cdots & y^{(m)} \end{bmatrix}$$

  Y is a $1 \times m$ matrix.

Observation. In other courses we might see X defined as the transpouse of the matrix we've just defined. But it turns out that when using this definiting, it's much easier to implement algorithms, so remember te definition we're going to use through out the course.

The same thing for Y. We see that here Y is the transpouse of that it's tends to be in other courses.

1.2 Logistic Regression as a Neural Network

To end this introduction, we'll see the basics of neural network programming using the simplest NN we can: a logistic regression.

So let's recall what's logistic regression and why it's useful. Logistic Regression is used in binary classification, the kind of problem where we have an input and want to predict between 0 or 1. An example could be an image and we want to say it what's a cat (1) or not (0).

Basicly we want an algorhthm to estimate the probability of $y = 1$ given x. In math we write:

$$\hat{y} = P(y = 1 \mid x), \qquad x \in \mathbb{R}^n$$

Logistic Regression estimates this quantity using the formula:

$$\hat{y} = \sigma(w^T x + b),$$

where w and b are parameters to be discovered and σ is the **sigmoid function**:

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

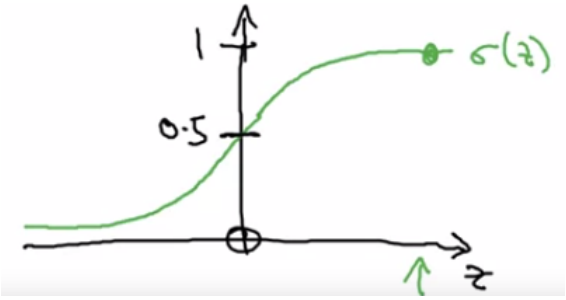


Figure 1.7: A sigmoid graph.

It's also common to create a new input $x_0 = 1$ and use the x vector as $x \in \mathbb{R}^{n+1}$ and use the formula $\hat{y} = \sigma(\theta^T x)$, where

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix} \qquad \theta_0 = b \qquad w = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix}$$

To find the parameters b and w, we need to define a **cost function**, which is a function that says how badly our algorithm is performing. This is a function that we want to minimize and when we minimize, we find the best values of b and w.

The **cost** function is a function of all training examples, while a **loss function** or **error function** is a function of a single traning example that measures how well our algorithm is performing.

For logistic regression, we use the loss function:

$$\mathcal{L}(\hat{y}, y) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$$

Notice that this is the same as:

$$\mathcal{L}(\hat{y}, y) = \begin{cases} -\log(\hat{y} - 1), & \text{if } y = 0 \\ -\log \hat{y}, & \text{if } y = 1 \end{cases}$$

That give us the cost function:

$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

1.2.1 Gradient Descent

We know have:

- A way of predicting the classes 0 or 1 using the sigmoid function;
- A way of measuring the error of our predictions.

What we need now is a way of chaning our parameters b and w in order to minimize the error. That's what the **gradient descent** algorithm does.

Let's first see a general graph of the cost function. In general, it looks like figure 1.8

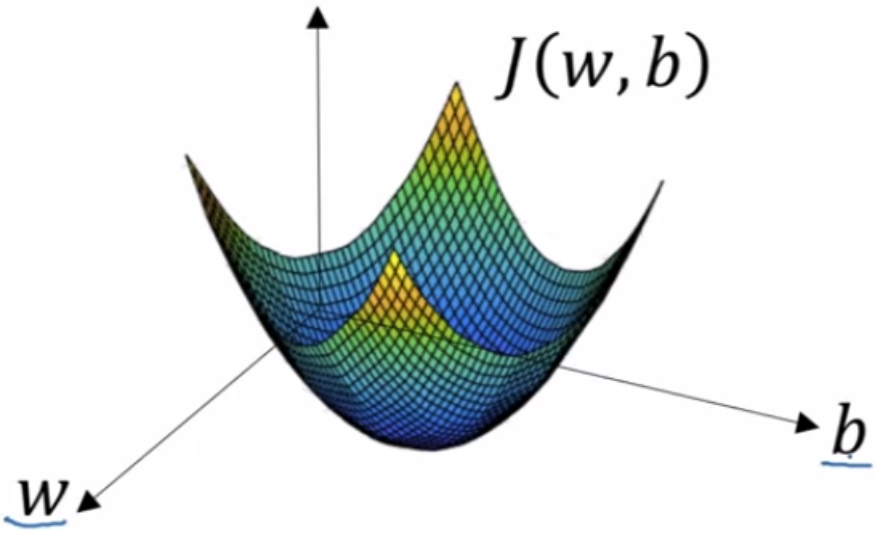


Figure 1.8: A generic graph of the cost function.

We see that J is what we call a **convex function**, which means that it is a function that was only one **local minimum** (or local maxima). This property is very important if we want to apply the gradient descent algorithm.

In Gradient Descent, we initialize b and w randomly and take steps into the direction that leads us to the lowest possible value of J. In order to do that, we calculate the **gradient** (the derivatives in each direction) of the function J and take a step in the opposite direction of the gradient.

Proposition 1.1

The gradient gives us the direction of the maximum increase of the a function.

Algorithm 1.1: Gradient Descent

Repeat {

$w := w - \alpha \frac{\partial w}{\partial J(w,b)}$

$b := b - \alpha \frac{\partial b}{\partial J(w,b)}$

}

In the algorithm, α is what we call the **learning rate**. It's how large we should step in the direction of the maximum decrease. If we take big steps, we can go much faster to the global minumum, but we might not be so accurated. On the other hand, we be take small steps, we can find the global minum accuratedly, but achieve it much slower.

Gradient Descent is a general optimization algorithm and can be applied to any convex function. So now we need to understand how to use it with logistic regression.

After calculating the derivatives, we'll have:

Algorithm 1.2: Gradient Descent for Logistic Regression

Repeat {

$J = 0;\ dw = 0;\ db = 0$

For $i = 1 \cdots m$:

$z^{(i)} = w^T x^{(i)} + b$

$a^{(i)} = \sigma(z^{(i)})$

$J+ = -\left[y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)})\right]$

$dz^{(i)} = a^{(i)} - y^{(i)}$

$dw+ = x^{(i)} dz^{(i)}$

$db+ = dz^{(i)}$

$J/ = m$

$dw/ = m;\ db/ = m$

$w := w - \alpha dw$

$b := b - \alpha db$

}

This version of the algorithm uses a for loop to compute J, dw and db. But when implementing the code into Python or other language, we always try to **vectorize** the code to make it faster.

Algorithm 1.3: Gradient Descent for Logistic Regression Vectorized

Repeat {

$Z = w^T X + b$

$A = \sigma(Z)$

$dZ = A - Y$

$dw = \frac{1}{m} X dZ^T$

$db = \frac{1}{m} \sum dZ$

$w := w - \alpha dw$

$b := b - \alpha db$

```
}
```

Chapter 2

Basical Concepts

So let's start introducing some notation.

Whenever we use:

$$z^{[i]}$$

we're talking about the z values of the i-th *layer* of the neural network.

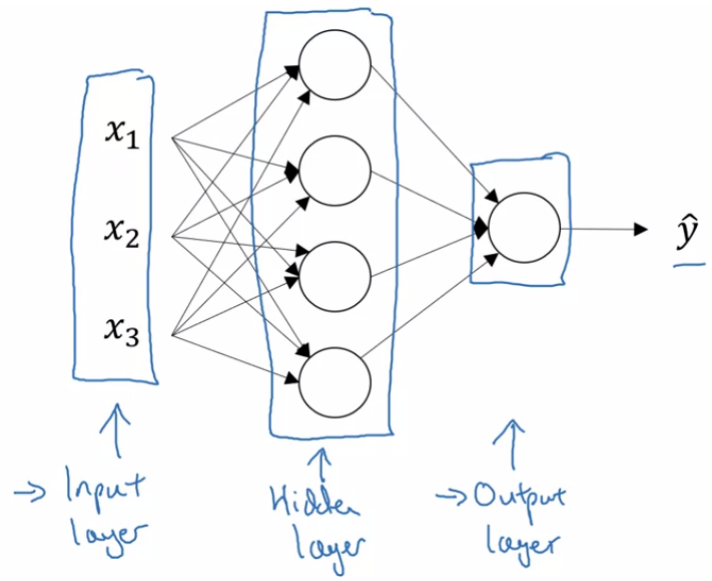
Thus, we have the following important equation:

$$\boxed{z^{[i]} = W^{[i]} a^{[i-1]} + b^{[i]}} \tag{2.1}$$

$$\boxed{a^{[i]} = \sigma(z^{[i]})} \tag{2.2}$$

2.1 Representation

Let's ilustrate a simple neural net:



This nn is divided into three parts:

- The **input layer** is the layer were we enter the data, the features. It's represented by a single vector x, which is a column of our inputs matrix X. Another way to denote the input is by $a^{[0]}$;

- The **output layer** is the layer were we get our answer. It can be one of more nodes and it's represented by a single vector y. It can be a binary vector, continuous vector, it could be even a codification of an image, sound, word or any codification just like the input;
- Finally, the **hidden layer** are all the nodes which are between the input and output, they are the intermediately computations that our neural net does. We don't need to stick to just one hidden layer, there could be many of them.

One more thing to keep in mind is that the neural network drawn in the picture is called a *two layer* neural network, because we don't count the input layer as a "real" layer (it's the layer zero).

Each layer (real layer) will have **parameters** associated with them, which we'll denote $W^{[i]}$ and $b^{[i]}$ for the i-th layer.

To understand what are node computes given an input, let's focus in the first node of the hidden layer in the figure. We can see that the inputs x_1, x_2, x_3 are all passed to that node, which uses tham to output something to the node in the output layer (or it could be outputed to the next hidden layer).

So basicaly a node is a logistic regression, it computes:

$$a = \sigma(Wx + b)$$

But as we have many computations like this, we have to use indeces to denote the first hidden layer and the first node of that layer. So the computations would be described as following:

$$z_1^{[1]} = W_1^{[1]T} x + b_1^{[1]}$$
$$a_1^{[1]} = \sigma(z_1^{[1]})$$

The superscript means we're talking about the first (real) layer and the subscript means we're talking about the first node of that layer. Therefore $a_i^{[l]}$ is the **activation value** (output value) of the i-th node in the l-th layer.

Of course, when coding a neural network, we don't compute each one of these equations using a for loop, we vectorize in order to compute the whole vector $z^{[l]}$ at once using equation 2.1.

Vectorizing the input Now, one more thing that we want is to be able to predict *mulpible* inputs at once. Indeed we can do that with some vectorization. Let's see how.

So let's recall that

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$$

and let's define a matrix $Z^{[l]}$ in which the j-th column is the vector $z^{[l](j)}$ (i.e., the value before the activation function of the layer l when applied to the input j).

$$Z^{[l]} = \begin{bmatrix} | & | & & | \\ z^{[l](1)} & z^{[l](2)} & \cdots & z^{[l](m)} \\ | & | & & | \end{bmatrix}$$

and also define the matrix $A^{[l]}$, which is $\sigma(Z^{[l]})$:

$$A^{[l]} = \begin{bmatrix} | & | & & | \\ a^{[l](1)} & a^{[l](2)} & \cdots & a^{[l](m)} \\ | & | & & | \end{bmatrix}$$

We can see that each row of matrix A tell us the activation value for a particular node of that layer for each input. For instance, the value $A^{[l]}_{i,j}$ is the value of the i-th neuron in the l-th layer of the neural network when we input the j-th input.

Given these matrices, we can train over all inputs using the vectorized formula:

$$\boxed{Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}} \tag{2.3}$$

$$\boxed{A^{[l]} = \sigma(Z^{[l]})} \tag{2.4}$$

and recall that $X = A^{[0]}$.

2.2 Activation Functions

The **activation function** is the function g applied to z. We've been currently using the *sigmoid function* σ, but there are some other functions that can be used and can significatly change our results and performances.

Another function one can use is the tanh function, given by the formula:

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

which is a shifted version of the sigmoid function. We have $\tanh(0) = 0$ instead of $\sigma(0) = 0.5$.

It turns out that the tanh function frequently works better than the sigmoid, because the values are between -1 and 1 instead of 0 and 1 and thus the average values tend to be closer to zero.

The exception is for the output layer, where it's good to have a number between 0 and 1 in many cases, so we could keep using the sigmoid function there and use the tanh on the hidden layers.

One of the down sides of these two functions is that when z is very large, the gradient is very small, because they're very flat for large (positive or negative) values of z.

That's why many times we see the **ReLU** (rectified linear unit) function being used:

$$ReLU(z) = \max(0, z)$$

Indeed the ReLU is the most used activation in practice and should be the first choise. One of the only down sides of it is that the derivative for values lower than zero is zero, but it works fine in practice.

There's modified version of the ReLU called **Leaky ReLU**, which is given by:

$$LReLU(z) = \max(0.01z, z)$$

It's just like the ReLU function, but it's not zero for any negative value of z and, thus, the derivative is not null on those points. Actually, indeed there no reason for the 0.01, it could be any small value and we can turn that into a hyperparameter of our algorithm.

Why activation functions Now that we've seen so many activation functions we might want to understand why do we even need them. And most importantly, why do we need they to be non-linear.

We could try using $a = g(z) = z$ (i.e., doing nothing). It turns out that if we have just the identity function, we can't express *complex* (non-linear) decision boundaries.

We'll not prove it here, but if we just use a linear function in all hidden nodes and a sigmoid, it is equivalent to a standard logistic regression in terns of what it can express.

$$a^{[1]} = w^{[1]}x + b^{[1]}$$

$$\begin{aligned} a^{[2]} &= w^{[2]}a^{[1]} + b^{[2]} \\ &= w^{[2]}\left(a^{[1]} = w^{[1]}x + b^{[1]}\right) + b^{[2]} \\ &= \left(w^{[2]}w^{[1]}\right)x + \left(w^{[2]}b^{[1]} + b^{[2]}\right) = \\ &= w'x + b' \end{aligned}$$

Above we can see that from one layer to the next we still have a linear equation (a composition of two linear functions is a linear function).

2.3 Backpropagation

Now we're going to understand how to find the optimal values for W and b using back-propagation.

The first step in this direction is calculating the derivative of the activation functions. So let's to it.

Sigmoid

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

$$\begin{aligned} \frac{dg(z)}{dz} &= \frac{1}{1 + e^{-z}}\left(1 - \frac{1}{1 + e^{-z}}\right) \\ &= g(z)(1 - g(z)) \end{aligned}$$

Tanh

$$g(z) = \tanh(z)$$

$$\begin{aligned} \frac{dg(z)}{dz} &= 1 - (\tanh(z))^2 \\ &= 1 - g(z)^2 \end{aligned}$$

ReLU

$$g(z) = \max(0, z)$$

$$\frac{dg(z)}{dz} = \begin{cases} 0, & \text{if } z < 0 \\ 1, & \text{if } z > 0 \\ \text{undef}, & \text{if } z = 0 \end{cases}$$

Leaky ReLU

$$g(z) = \max(0.01z, z)$$

$$\frac{dg(z)}{dz} = \begin{cases} 0.01, & \text{if } z < 0 \\ 1, & \text{if } z > 0 \\ \text{undef}, & \text{if } z = 0 \end{cases}$$

Now let's understand how to do **gradient descent**.

Let's denote L the number of layers of our network. Them, the last value derivative (the first we're going to compute) is given by the formula:

$$dz^{[L]} = a^{[L]} - y \tag{2.5}$$

The values of dW and db, for any layer are given by the formulas:

$$dW^{[l]} = dz^{[l]} a^{[l-1]T} \tag{2.6}$$

$$db^{[l]} = dz^{[l]} \tag{2.7}$$

The other values of dz for any layer other than the output layer are given by the formulas:

$$dz^{[l]} = W^{[l+1]T} dz^{[l+1]} * g^{[l]\prime}\left(z^{[l]}\right) \tag{2.8}$$

where $*$ denotes que *element-wise* product.

There's also a vectorized way of computing this for all the training examples at once.

$$dZ^{[L]} = A^{[L]} - Y \tag{2.9}$$

$$dW^{[l]} = \frac{1}{m} dZ^{[l]} A^{[l-1]T} \tag{2.10}$$

$$db^{[l]} = \frac{1}{m} \text{sum}\left(dZ^{[l]}, \text{axis=1}\right) \tag{2.11}$$

$$dZ^{[l]} = W^{[l+1]T} dZ^{[l+1]} * g^{[l]\prime}\left(Z^{[l]}\right) \tag{2.12}$$

2.4 Random Initialization

Finanlly, we need to initialize the parameters of the neural network. It turns out that initializing everything as zero or everything as the same number, then all nodes of a layer will be exactly the same even with many iterations of gradient descent.

For the values of b, we can initialize with zero, but for W, we prefer to initialize with *small* random numbers. We like the numbers to be small though because if we're using tanh or σ, big numbers will have a very plat derivative and, thus, the algorithm will learn slower.

2.5 Notation for deep neural networks

We'll use L to describe the number of layers the neural network have (remember, the input layer is not a real layer).

$n^{[l]}$ denotes the number of nodes on layer l.
$a^{[l]}$ denotes the activation values on layer l.
$g^{[l]}$ denotes the activation function used on layer l.
Also, the dimensions of $W^{[l]}$ and $b^{[l]}$ are $\left(n^{[l]},\ n^{[l-1]}\right)$ and $\left(n^{[l]},\ 1\right)$.

Chapter 3

Practical Aspects of Deep Learning

In this chapter we're going to focus on the practical aspects of deep learning, like how to choose the hyperparameters, how to make the code faster, how to apply regularization and others.

The first thing to understand is that, in practical, we divide our data set into three:

- *Training set*, in which we're going to train the model;
- *Hold-out cross validation set (or dev set)*, in which we're going to evaluate our model and choose hyperparameters;
- *Test set*, in which we're going to evaluate our final model, with the hyperparameters setted.

In the previous era of Machine Learning, we would use 70% for the training/dev sets and 30% for testing, or 60/20/20%. But now in the big data era, we can use much small fractions for the testing and hold-out sets (like 1% or even less, since this can be equivalent to about 10000 examples).

Another important thing to keep in mind is that the training set and test set must have the *same distribution* of that. For instance, if we train on images of cats coming from the web, than we should not test on images of cats coming from mobile cameras.

3.1 Regularization

We've seen that the cost function of our neural network is a function J such that:

$$J\left(W^{[1]}, b^{[1]}, \cdots, W^{[L]}, b^{[L]}\right) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y^{(i)}}, y^{(i)})$$

Regularizing a neural net model means adding a penality when the model makes $W^{[i]}$ bigger. This helps the model to prevent overfitting.

Thus, we have:

$$J\left(W^{[1]}, b^{[1]}, \cdots, W^{[L]}, b^{[L]}\right) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y^{(i)}}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L}\left\|W^{[l]}\right\|^2$$

Where

$$\left\|W^{[l]}\right\|^2 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} \left(W_{i,j}^{[l]}\right)^2$$

is called the *Frobenius norm* (sometimes we write $\left\|W^{[l]}\right\|_F^2$).

Here λ is called the **regularization factor** and the bigger it is, the more we penalise the model for having bigger $W^{[i]}$.

Another kind of regularization is called **dropout regularization**. In this kind of regularization, we set a small probability of removing a node at all and have a new smaller network.

One way to implement it is called *inverted dropout*.

```
l = 3
keep-prob = 0.8
d3 = np.random.rand(a3.shape[0]. a3.shape[1]) < keep-prob
a3 = np.multiply(a3, d3)
a3 /= keep-prob
```

The last line is used to keep the espected value of a the same, so we keep the same scale of z when doing $z = wa + b$.

When using dropout, though, we *don't apply it when using the neural net to predict values*.

To explain intuitively why dropout works, we need to think about the nodes connected to a particular node. If the NN just give importance to one of the those nodes, then it might be dropped and the model will perform poorly. Therefore, to work well, the model needs to spread the weights into all the neurons, shirking them.

There are other ways to perform regularization.

Data augmentation is when we change the data to get more data. For instance, if we have a image dataset, we can flip the images, reduce saturation, blur the image, and do many other things to get more images to train the make our mode better.

If we have a sound dataset, we can make the sound loud, or add noise, supression and so on.

Early stopping is when we stop iterating our model using the error in the dev set. While the error in the train error one decreases, the error in the dev set tends to decrease for a while and than starts incrasing. What early stopping does is stop iterating the model when the error in the dev set starts increasing.

3.2 Optimizing the Problem

3.2.1 Normalizing input

Normalizing the input is very important because it garantees to us that the input data will have zero mean and variance one. To perform normalization, we calculate:

$$\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$$

and

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^{(i)2}$$

where $x^{(i)2}$ mean elementwise power.

Please note that these values are calculated using the *training set*. And them we'll apply the below formula to *every* input that we'll feed into the network (it can come from the training, dev or test set).

$$x := \frac{x - \mu}{\sigma}$$

Normalizing the input is important because it garantees that Gradient Descent will converge much faster, since J will have a more circular bow shape, since of a elliptical one.

3.2.2 Vanishing / Exploding Gradients

In this section we're going to cover a problem that we can face when traning neural networks. Sometimes, the gradients become too small or too big and we can't training anything at all.

To understand that, suppose we have a very deep neural network. It's not so hard to imagine that as computations are performed upon computations, we can get very large or very small numbers for values like the activation function.

For instance, suppose all $W^{[l]}$ have values of 1.5 then if we don't use an activation function, our final output will be something proportional to $W^{[1]}W^{[2]}\cdots W^{[L]}$, which will me proportional to 1.5^L. If L is very big, this can be huge.

If we changed 1.5 by 0.5, then we would have amoust zero.

To solve (parcially) this problem, we need to have a careful initialization of our weights.

Let's just consider a single neuron. If we ignore b, then we have

$$z = w_1x_1 + w_2x_2 + \cdots + w_nx_n$$

The later n is, the smaller we want w_i to be, because we want z to be approximately in the range $[-1, 1]$. One thing that we would wish is to have

$$\text{Var}\left(w_i\right) = \frac{1}{n}$$

So we can initialize it like:

$$W^{[i]} = \text{np.random.randn(shape)} \cdot \text{np.sqrt}\left(\frac{1}{n^{[l-1]}}\right)$$

It turns out that if we are using ReLU, it's better to have a variance of $\frac{2}{n}$, so we can replace the one by 2 in the sqrt.

The variatian with 1 is called the *Xavier initialization*, but we can use another variatian in which we multiply by:

$$\sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}$$

These factors could all be used as a base model, but we can really tune this as a hyperparameter if we want, although it's not a very important hyperparameter.

3.2.3 Gradient Checking

A very common technique when using Gradient Descent is to use **Gradient Checking** to make sure our gradient computations are been performed right. It will not go to the final model, but we use it to make sure we've coded everything right.

To do it, basicaly we approximate the gradients numericaly and compare with the gradient we're computing using the formulas. Of course we don't do that when we want to predict any value or use our model in the real world because we're computing the gradient two times (and computing it numericaly is not efficient). We just use this technique when coding the backpropagation to make sure we're doing it right.

To approximate the gradient, it's very since, we just use:

$$g(\theta) \approx \frac{f(\theta + \varepsilon) - f(\theta - \varepsilon)}{2\varepsilon}$$

which is called *two-sided approximation*.

So the first thing in order to use gradient checking is to reshape our parameters $W^{[1]}, b^{[1]}, \cdots, W^{[L]}, b^{[L]}$ into just one big vector θ. And it's derivative will be $d\theta$

Now we're going to compute $d\theta_{\approx}$ which can be computed doing:

$$d\theta_{\approx i} = \frac{J(\theta_1, \theta_2, \cdots, \theta_i + \varepsilon, \cdots, \theta_k) - J(\theta_1, \theta_2, \cdots, \theta_i - \varepsilon, \cdots, \theta_k)}{2\varepsilon}$$

For each value of i.

And what we want is

$$d\theta_{\approx} \approx d\theta$$

To now that, we can calculate:

$$\frac{\|d\theta_{\approx} - d\theta\|}{\|d\theta_{\approx}\| + \|d\theta\|} < \delta$$

for a small value of δ. It could be 10^{-7} to make sure it's really correct or 10^{-5}. If the difference is greater than something like 10^{-3} than probability we have a bug somewhere.

3.3 Optimization Algorithms

In this section, we're going to view many algorithms that allow us to train much faster.

3.3.1 Mini-batch Gradient Descent

What we've been doing in terms of gradient descent so far is to process the whole training set to take a step in the opposite direction of the gradient.

If m, our number of training exemples, is very large (which is normal in Big Data applications), then a single gradient step is very expensive.

One way to optimize that is to use what's called **mini-batches**. Basically we split our training set into multiple training sets (if say 1000 examples) and we run each step of gradient descent using one of these sets.

That's what we call an **epoch** of gradient descent (a pass of forward-prop, cost computation, back-prop and parameters update).

Of course, by doing this, we're not optimizing the original J function (which is computed for every training example). However, we'll get a low cost model at the end.

If we plot the cost of J at each iteration (epoch), we'll not have that classical curve in which J decreases at each iteration. The curve will be a little noisy, increasing and decreasing. But in the long run, it'll converge to some low value. And not just that, it does it much faster than the original gradient descent (which is also called **batch gradient descent**).

One important concept to mention is the size of the mini-batches. If the size is one, than we call the algorithm **stochastic gradient descent**. This will make our epochs really fast, but we'll lead to an algorithm with a much higher cost than we could get using greater batches. Stochastic gradient descent will be *around* the minimum optima of J. The greater our mini-batches are, the more closer the that mininum optima we'll lead. That's the trade-off between speed and precision.

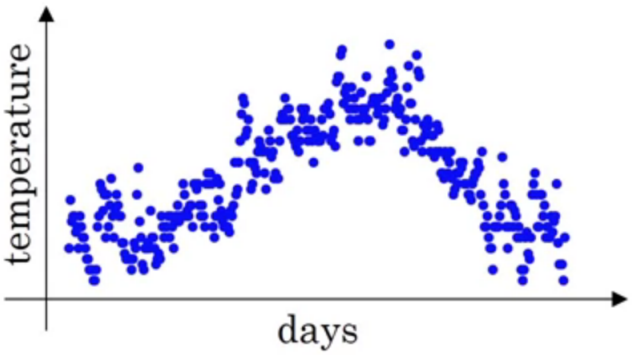
We can also train with mini-batch until the cost is very close to the mininum optima and then change to batch gradient descent and run some more epochs to make sure that our algorithm is really precise.

In practical, because how the computer memory and vectorization works, people tend to use powers of two for the size of the mini-batches. So that's a thing to keep in mind. Common values are 2^6 up to 2^9.

3.3.2 Exponentially weighted averages

Now we're going to cover other optimization algorithms that are better than gradient descent. But to understand them, we need to talk about a concept of statistics which is called **exponentially weighted (moving) averages**.

To understand that, let's use an example. We'll plot the temperatures over the days.



Here we sse that the values are very noisy, going up and down on each day. However, if we want to visualize the *trend* of the values, we can apply the moving average technique.

Basicaly, instead of plotting these points (let's call them θ_i), we'll plot V_i.

First, we initialize V_0 as zero.

$$V_0 = 0$$

then, we get the next value by averaging the previous value of V_i and the current value of θ_i.

$$V_1 = 0.9V_0 + 0.1\theta_1$$

a general formula would be:

$$\boxed{V_i = 0.9V_{i-1} + 0.1\theta_i}$$

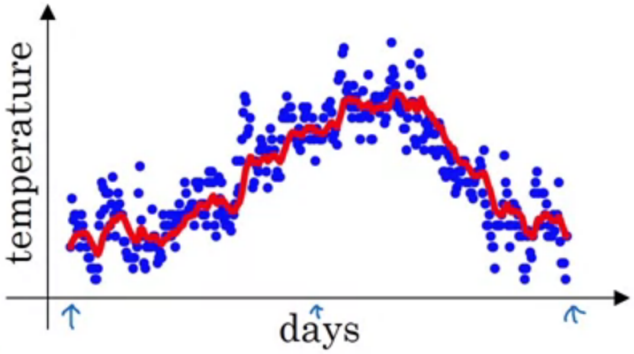


Figure 3.1: The moving average of the temperatures.

If we compute and plot that, we'll get figure 3.1

We see that we got a much more smooth curve because we're always considering the previous value to create the next one. We could make the formula even more general:

$$\boxed{V_i = \beta V_{i-1} + (1-\beta)\theta_i}$$

where β is the percentage of how much we would like to consider the previous value.

Indeed the value of β will dictate approximately how many previous iterations we're considering on each point. To have an idea of how many previous points we're considering, we use the formula

$$\frac{1}{1-\beta}$$

Therefore, for $\beta = 0.9$, we're considering approximately the 10 previous temperatures in order to get the current value. That's why it's called a *moving* average or a *local* average.

The lower β is, the less temperatures we'll be considering on each average and, therefore, the faster those averages will change, leading to a noiser curve, but that adapts faster. And if β is big, we'll be considering more days and, therefore, the average will change more slowly, leading to a smoother curve that has some latency to adapt to new trends.

This value will be a hyperparameter in our models.

To understand a little better this algorithm, let's try to get a non-iterative formula. We'll get something like:

$$V_i = 0.1\theta_i + 0.1 \times 0.9\theta_{i-1} + 0.1 \times 0.9^2\theta_{i-2} + \cdots + 0.1 \times 0.9^k\theta_{i-k} + \cdots$$

So, indeed, we're calculating V_i as an average of *all* previous values. But the farest the values are from the current we're calculating, the less impact it have on the current one. Indeed, we see that the influence decays *exponentially* (that's why this term apears in the name).

If we sum all coeficients, we would except to get exactly 1, since this is an average. But unfortunatelly, that's not what happens. In fact, we say that this is approximately an average, because we're missing an **bias term**.

However, using this kind of average is very good computacionally, because we don't have to keep track of (say) the last n terms to compute a local average. We can do that in $O(1)$ for each element.

If we would implement this algorithm, it would be like:

Algorithm 3.1: Exponentially weighted averages

$V_\theta = 0$
Repeat {
  $V_\theta := \beta V_\theta + (1 - \beta)\theta_t$
}

As one can see, this algorithm is very memory and time efficient.
Finally, we can make a correction to the bias term by dividing V_i by $1 - \beta^i$.
So we can use the formula:

$$V_i = \frac{\beta V_{i-1} + (1 - \beta)\theta_i}{1 - \beta^i}$$

This will help a lot in the first iterations where we still don't have many values in the window. Think of it as $1 - \beta^i$ being a good approximation for the sum of the coefficient of the average.