Coursera

Notes

Neural Networks and Deep Learning

Carlos García González

Summer 2020

Contents

Chapter 1

Neural Networks and Deep Learning

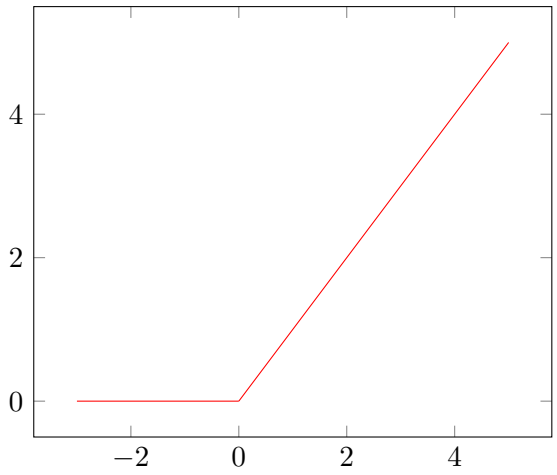


Figure 1.1: Plot of an example RELU function

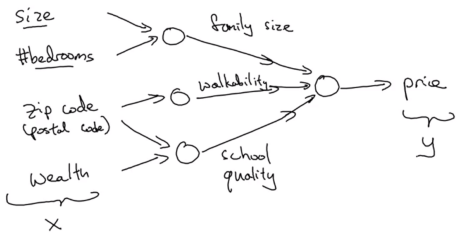
Introduction to Deep Learning

What is a Neural Network?

The housing price prediction problem can be seen as the simplest of Neural Networks. First, lets start by assuming that the house pricing is only affected by its size.

Size x -¿ Neuron (does the determined function) -¿ Price y

In this particular case, a RELU (Rectified Linear Unit) function [1.1] is presented and is often seen in Neural Network examples. There can not be a house priced at $0, hence the implementation of this type of function. A neural network is produced by taking many single neurons and by stacking them together.
Now letś add more variables for the same problem, the representation of the network would be similar to the following graph:



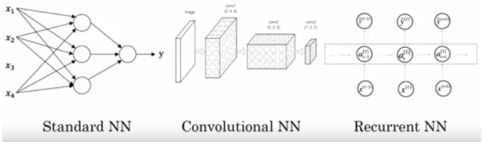
The inter-connecting nods can be seen as individual RELU representations that lead to the values seen on the lines which represent important characteristics in a model and that lead to the final result; X is equal to the stack of the given inputs and Y is equal to the desired output. The "in-between" nods represent hidden features, these are the ones created subsequential to the input data and the network itself defines the relation between these.
The network processes two different types of data:

- The X input values
- Training data

With these three the network can predict the output Y value. Giving a Neural Network enough correlational X:Y data will make it better at figuring out the sort of equations that are similar to the ones on the training data.

Supervised Learning

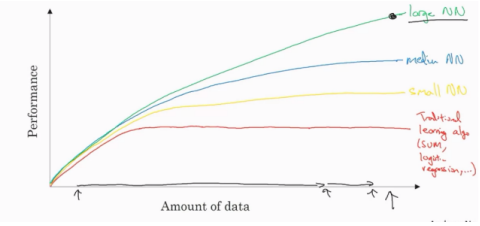
Economic value of deep learning implementations really come from Supervised learning. You have an input x and you want to have a result y. In the previous problem it would be x = Home features, y = Price and the application would impact on the real estate business. Another example could be related to e-marketing, where x = Ad & user info, y = Click on add? (t/f) and the would impact on online advertising. Another examples could be related to computer vision, audio recognition, machine translation and other topic related projects. Simple neural networks such as the two first examples have usually a standard NN implementation, image processing is often convolutional and sequential data implementations (such as audio, because it has a temporal component) are often recursive NN; for particular or more complex implementations there can be custom or hybrid architectures.



Structured data (DB with specific fields) vs unstructured data (audio, images, [···]), this tangible difference is important to take into account because historically it has been more difficult to calculate from the latter, hence the importance of the architecture design importance. Although the most "exciting" implementations are often the unstructured data related systems, the market has more demand for well-built structured data solutions.

Why is Deep Learning taking off?

In the traditional learning algorithms, there was a point where the increment of input data did nothing for the performance of the system. In modern problem solving we have a lot of data to handle, so the design of a system which could profit from it was considered crucial. Neural networks are capable of incrementing the system performance in a better fashion than traditional implementations.



One thing to note is that the "Amount of Data" refers only to the amount of labeled data (contains x and y definitions). Also, the notation for the cardinality is expressed as (m).
Although the increment in "learning room" is evident, the performance also has a cap that can occur if you run out of data or if the NN is big enough that it becomes really slow to train.
Large neural nets are only necessary when a lot of data is required to process. There can be little to no difference in large NN and small NN systems where the data-set (m) is relatively small.
Deep learning progress has developed a weighted importance in data, computation-related (hardware-wise) and algorithmic advancements that allow the systems to come to a reality. One of the most notable innovations is the transition from sigmoid to RELU functions; sigmoid functions had areas (where b = 0) that led to the decrease of learning speed due to the slow parameter revision. The innovations of the three disciplines have helped the development cycle (idea -¿ code -¿ experiment) that leads to more architecture revisions.

Logistic Regression as a Neural Networook

Binary Classification

Usually the data processing is based on going through the entire dataset without explicitly using a for loop. Usually the process consists of a forward pause/forward propagation step, followed by a backward pause/backward propagation step.
Binary classification consists of an input x that is processed by an algorithm that outputs a binary value called label which indicates a certain input behavior or feature. For image classification (cat vs no cat), the image is decomposed to every pixel value per channel and then concatenated into a single object.
A single training example is denoted by the following notation:
$(x, y), x \in \mathbb{R}^{n_x}, y \in \{0, 1\}$; where n_x is the dimension of the input features x.
The training set is comprised of all the pairs of training examples, lowercase m is commonly used as the indicator for the number of training examples.
In order to put the training examples in a more compact fashion, the following model must be used:

$$X = \begin{Bmatrix} \dots & \dots & \dots & \dots \\ x^1 & x^2 & \dots & x^m \\ \dots & \dots & \dots & \dots \end{Bmatrix} \tag{1.1}$$

This matrix X will have m columns, where m is the number of training samples and n_x rows ($X \in \mathbb{R}^{n_x \times m}$). Although there are other implementations, using this convention will make further implementation easier.
For the label output, stacking the results in columns has also been commonly used, and is defined by the following model:

$$Y = \begin{Bmatrix} y^1 & y^2 & \dots & y^m \end{Bmatrix}, Y \in \mathbb{R}^{1 \times m} \tag{1.2}$$

Logistic Regression

Learning algorithm used when output labels Y are either 0 or 1. Given an input feature vector x, you need an algorithm that can output a value $\hat{y}$ representing the prediction weather the feature vector has or has not a certain behavior; the prediction of $\hat{y}$ being equal to 1 can be written as $\hat{y} = P(y = 1|x)$.
Given the parameters $w \in \mathbb{R}^{n_x}, b \in \mathbb{R}$, we must be able to construct a function that goes through the feature vector x and outputs $\hat{y}$. The function

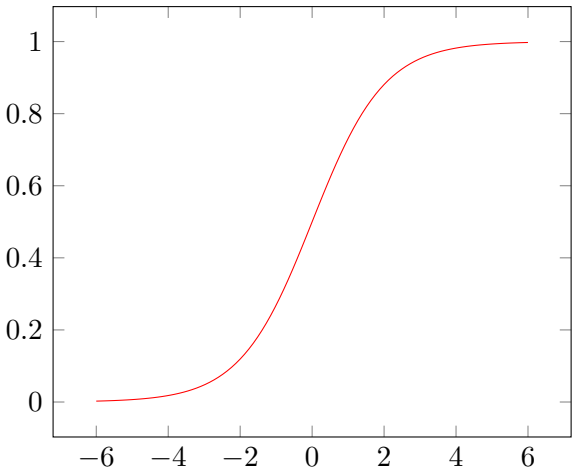


Figure 1.2: Plot of an example sigmoid function

is a modification of $w^T x+b$ so that the range does not exceed 1 or be smaller than 0, for this, the use of a sigmoid function is required. The sigmoid of z [1.2] can be calculated through the expression $\sigma(z) = \frac{1}{1+e^{-z}}$; if the z value is very large the output will be closer to 1, on the contrary it will be closer to 0. Given the previous knowledge, we can now assume the following function: $z = w^T x + b, \hat{y} = \sigma(z)$. When implementing logistic regression, it is crucial to learn parameters w and b so that $\hat{y}$ becomes a good estimate of the chance of y being equal to 1; b usually corresponds to an inter-spectrum parameter. In neural network implementations it is recommended to keep these parameters separate although there are some implementations in which these are contained in the same vector.

Logistic Regression Cost Function

To train parameters w and b it is essential to define a cost function. As a result, it is desired that $\hat{y}^{(i)} \approx y^{(i)}$, each training example can be represented as $\hat{y} = \sigma(w^t x^{(i)} + b)$.
To measure the effectiveness of the developed algorithm it is necessary to use the loss function. Square error is often used in other disciplines, but due to the nature of the network-related problems the following formula is used instead: $\mathcal{L}(\hat{y}, y) = -(ylog\hat{y} + (1-y)log(1-\hat{y}))$. The desired result for $\mathcal{L}$ is for it to be as small as possible.
The loss function checks for every training example how it is behaving, in

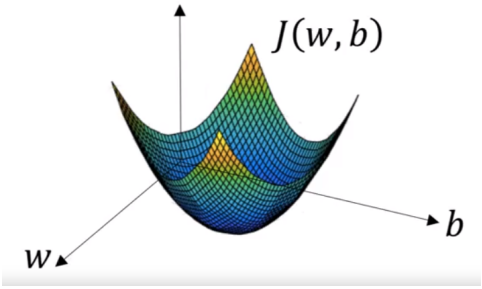


Figure 1.3: Plot of a gradient descent

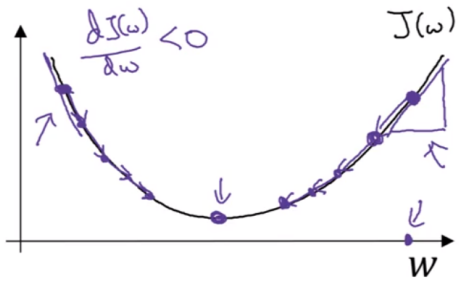


Figure 1.4: Example of a bidimensional gradient descent

order to check the complete training set a cost function must be used. The cost function is defined by: $J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)})$.

Gradient Descent

The gradient descent algorithm is used to learn the parameters w and b. In the figure [1.3] the parameters are represented in the x and y axis while the result of the cost function J is represented in the z axis. In the figure it's clear that the objective should be to find the values in which w and b converge while the cost function is 0. Analyzing the convex representation of the gradient descent also reveals the use of the previously-mentioned cost function; if square error was utilized, the graph would show crevices leading to erroneous calculations.
The process to estimate the final values consists of an "initial guess" for the parameters, some people use random generators for this step but is quite rare. Gradient descent starts at the given point and "descends" downhill through the graph until it reaches the global optimal. To better understand

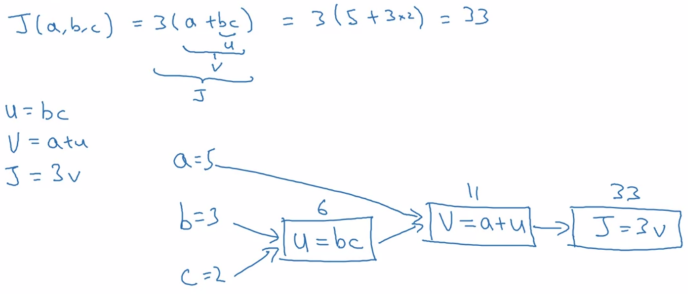
the gradient descent process let $J(w)$ be a concave function that we want to find the w value for it to be 0; a pseudocode for the descent would be:

```
Repeat{
    w := w - α dJ(w)/dw
}
```

This process is repeated until the algorithm can encounter an acceptable answer for the conversion. α is the representation for the learning rate, this controls the size of the step for each iteration, and the quantity obtained by the derivative is simply the update of the change that the parameter w will suffer. A visual representation can be the one on figure [1.4].
Of course, thanks to the cost function consisting of two variables the updates would have to occur for both parameters. The updates would be the following for $J(w, b)$:
$w := w - \alpha \frac{\partial J(w,b)}{\partial w}, b := b - \alpha \frac{\partial J(w,b)}{\partial b}$

Computation Graph

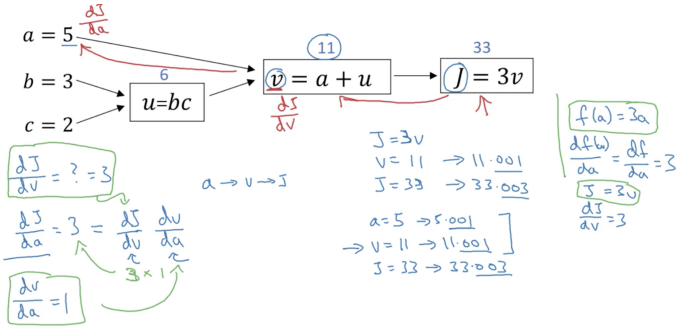
Computation of a NNET are organized by forward pass/forward propagation in which the output is calculated, followed by a backward pass/propagation step in which the gradients are computed; the computation graph is a representation in which it's shown why it is done this way. A computation graph is a graphical representation of the calculation processes in a computer. An easy example can be the following:



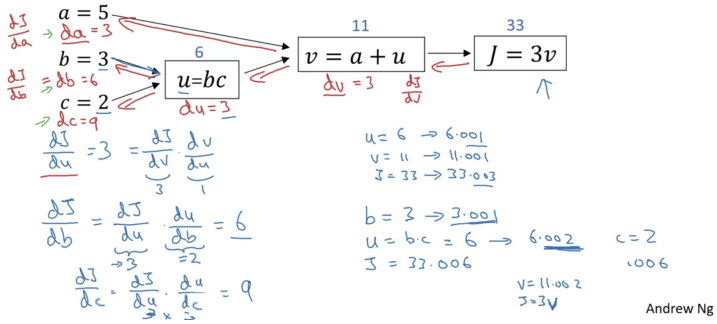
These are useful when there is an output variable that can be optimized; in logistic regression, the output J of the cost function is expected to be minimized as much as possible. The process for this to be possible is a "right to left" (from the result to the original inputs) that can be translated to derivatives in computing terms.

Derivatives with a Computation Graph

In order to calculate the back propagation steps in the computation graphs, it is necessary to calculate multiple derivatives to show the impact of the variables in the end result. A simple example seen on the online course is the following:



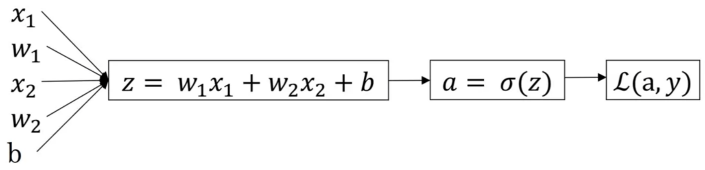
For back propagation, there is usually one output variable that we care about in order to optimize it. In programming, there is a convention to write this variables so they are more understandable; in the example above this would be J. The calculations of the derivatives in back propagations will be this final output variable in respect of another intermediate variable of the system, and the convention follows the structure of only writing only a "d" followed by the index of the intermediate variable (if "var", then it would be dvar).
The complete example shown in the online course is the following:



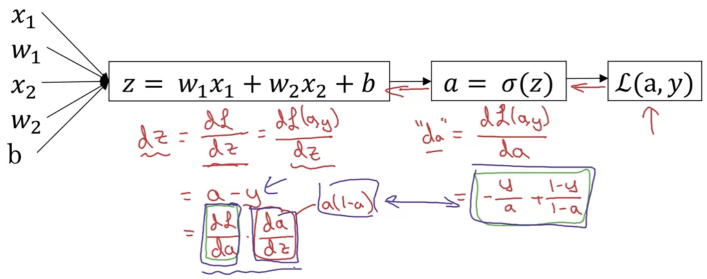
It's rather easier to compute from "right to left" and storing the derivatives for the chain rule as soon as the algorithm encounters its definition in order to simplify further calculations.

Logistic Regression Gradient Descent

Using the computation graph for gradient descent is rather overkill, but easier to begin to understand the concept. Recap of the functions used by the logistic regression:
$z = w^T x + b$
$\hat{y} = a = \sigma(z)$
$\mathcal{L}(a, y) = -(ylog(a) + (1 - y)log(1 - a))$
The computational graph for these functions would be the following:



In logistic regression, the goal is to modify the parameters w_m and b such that the function $\mathcal{L}$ becomes closer to 0. Using rules of derivation, we can go through the graph to calculate the derivatives:



The final step for the backwards propagation process is to calculate the values for the inputs so the loss is minimal. This is calculated simply through the following process: $\frac{\partial \mathcal{L}}{\partial w_m} = x_m dz$ (represented as $\mathrm{dw_m}$ in the code), $db = dz$ and by modifying the values in the following fashion:
$w_m := w_m - \alpha dw_m$
$b := b - \alpha db$
It's important to take into account that $\frac{d\mathcal{L}}{dz} = a - y$ and is represented as dz in the code.

Gradient Descent on m Examples

The derivative of the overall cost function in respect of a w value is the average of derivatives respect to the same value of the individual loss terms,

this would result in the overall gradient that will be used in gradient descent. A simple introduction to the algorithm is the following:

```
J=0; dw1=0; dw2=0; db=0
For i=1 to m
    z(i)=wTx(i)+b
    a(i)=σ(z(i))
    J+=-[y(i)log(a(i))+(1-y(i))log(1-a(i))]
    dz(i)=a(i)-y(i)
    // This example has two training examples, if there were more they
         should be processed in this section
    dw1+= x1(i)-dz(i)
    dw2+= x2(i)-dz(i)
    db+=dz(i)
J/=m
dw1/=m; dw2/=m; db/=m
```

We are using the updated parameters located at the last line as accumulators, do not forget that they are the derivative of the referenced parameter in respect of the total J function.

The algorithm above exemplifies only one step of the gradient descent, meaning that for it to work properly it must be repeated several times. It becomes clear that for loops could be used to solve this problem and to iterate over the w accumulators, but in deep learning algorithms explicit for loops tend to impact greatly on its efficiency. In the deep learning era it's important to avoid the use of explicit for loops to process bigger sets of data, a common solution is to use vectorization techniques.

Python and Vectorization

Vectorization is the art of removing for loops inside of the script. Deep learning nowadays is trained through big datasets, and avoiding for loops is a good practice to reduce resource cost. Where a non-vectorized algorithm goes through the whole vector iterating over the counter, vectorized scripts compute the whole vector in one step. In python, a simple script that is used to calculate z is the following:

```
z=np.dot(w,x)+b
```

Where np is the call to the NumPy library and the dot product calculates $w^T x$. An example was developed that showed the immense advantage of using vectorization, this can be found on the following path:

`0-Coursera_notes/python/1-vectorization.py`

It is known that GPU cores are often used in deep learning processes (although CPUs cores may also be used), these contain parallelization instructions that are called Single Instruction Multiple Data. Functions such as the np.dot() instruction benefits greatly to the lack of explicit for loop existence and enables to take advantage of parallelization on CPUs and GPUs.
Using vectorization, the previously seen code for logistic regression would be something like the following pseudocode:

```
J=0; dw=np.zeros(n-x, 1); db=0
For i=1 to m
    z^(i)=w^T x^(i)+b
    a^(i)=σ(z^(i))
    J+=-[y^(i)log(a^(i))+(1-y^(i))log(1-a^(i))]
    dz^(i)=a^(i)-y^(i)
    dw+=x^(i)dz^(i)
    db+=dz^(i)
J/=m
dw/=m; db/=m
```

We got rid of one for loop although there is still the one that goes through the training examples.

Vectorizing Logistic Regression

To make a prediction with m training examples you have to compute the forward propagation with the following: $z^{(i)} = w^T x^{(i)} + b$ and the activation $a = \sigma(z^{(i)})$, these for each training example. There is a way to remove the explicit for loop of the code to implement this functionality.

We defined X as a (n_x, m) matrix that contains the training inputs and the objective of the vectorization process is to compute z and a for the whole vector in only one call inside of the code.
For z, the output matrix can be represented as a (1, m) matrix containing $[z^{(1)}, z^{(2)}, \cdots, z^{(m)}]$ and is the result of $w^T X + [b, b, \cdots, b]$ that would generate an output $[w^T x^{(1)} + b, w^T x^{(2)} + b, \cdots, w^T x^{(m)} + b]$ that of course is a (1, m) vector. When you stack your training examples x horizontally it becomes X, and when you also stack your z values horizontally it is defined as Z following the same logic. In the Python environment, the calculation for this value can be represented as:

```
Z = np.dot(w.T,x) + b
```

Although b is a real number in this example, Python will automatically expand it as a row vector; this is known as "broadcasting". Similar to Z and X, when the values of a (the activation function) are stacked horizontally it is referenced as A. To implement the activation using vectorization there needs to be a calculated Z which would be processed by a sigmoid function implementation.

Vectorizing Logistic Regression's Gradient Output

As it was shown before, the derivatives needed were calculated by doing m derivatives of the form $dz = dz^{(i)} = a^{(i)} - y^{(i)}$. Similar to the previously seen calculations, the (1, m) matrix generated with the results of the dz computations will be defined as dZ. Based on the previously-seen A and Y definitions, dZ can be computed with these two variables. Although some for loops were successfully evaded there is still one while iterating over the training examples and db.
The value of db only changes by summing up the value of the multiple dz outputs and calculating the average; it is essentially $db = \frac{1}{m} \sum_{i=1}^{m} dz^{(i)}$, and in Python this can be translated as:

```
(np.sum(dZ)) / m
```

Following the same logic, dw can be seen as $dw = \frac{1}{m} X dZ^T$ that in Python would look like:

```
(X*dZ)/m
```

With these implementations, both dw and db are updated without the need of a single for loop. The vectorized implementation as a whole would look something like the following pseudocode:

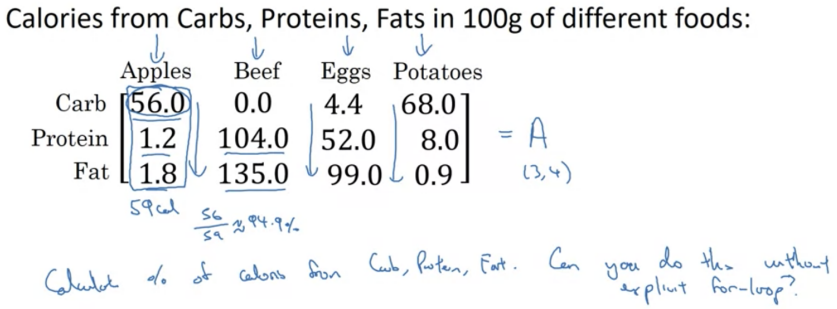
```
Z = w^T X + b // np.dot(w.T, x) + b
A = σ(Z)
dZ = A - Y
dw = 1/m X dZ^T
db = 1/m np.sum(dZ)
w := w - α dw
b := b - α db
```

With this code, the forward propagation, back propagation, prediction computing and derivative computing can be processed for the m training examples without using a for loop. This is basically a single step for gradient descent, for it to calculate multiple steps there should be a for loop over this process for it to be possible, this should be the only for loop allowed at the moment.

Broadcasting in Python

The example for the following image can be located at:

`0-coursera_notes/python/2-Broadcasting.py`



In python a simple broadcast works by expanding values to fit into equations, such as the following examples:

$$\begin{bmatrix}1\\2\\3\\4\end{bmatrix} + 100 = \begin{bmatrix}1\\2\\3\\4\end{bmatrix} + \begin{bmatrix}100\\100\\100\\100\end{bmatrix}$$

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} + \begin{bmatrix}100 & 200 & 300\end{bmatrix} = \begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} + \begin{bmatrix}100 & 200 & 300\\100 & 200 & 300\end{bmatrix}$$

The general principle for broadcasting in python is that if you have a (m,n) matrix and you add/subtract/divide/multiply a (1,n) or (m,1) matrix it will multiply m or n times the second one and do the operation.
Broadcasting can be seen either as a strength or a weakness. Due to the simplicity of its implementation, there can be many errors that can be difficult to detect due to its logical nature.
A common error is that there can be Rank-1 arrays that have a (m,_) dimension, calculations such as transpose would result in erroneous calculations. It is recommended in NNETs to not use data structures with the dimension in the example above. It's recommended to generate the values as column or row vectors for consistent behavior.
A recommendation is to throw assertion statements around that can verify the nature of a vector, it also has documentation purposes. If a Rank-1 array is needed, the reshape function can also be used.

Row Normalization

Another common technique we use in Machine Learning and Deep Learning is to normalize our data. It often leads to a better performance because gradient descent converges faster after normalization. Here, by normalization we mean changing x to $\frac{x}{\|x\|}$ (dividing each row vector of x by its norm).

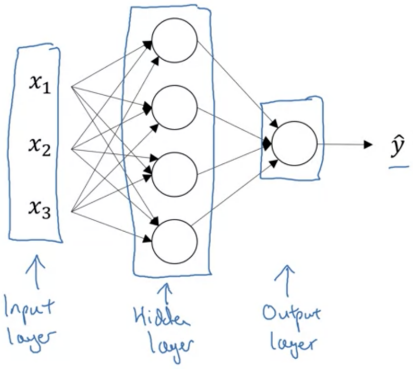
Logistic Regression with a Neural Network Mindset

In the `1-Assignments/1-NNETs_and_DL/W1` folder, there is a PDF with the Jupyter notebook assignment for Week 2 of the Coursera specialization.

Shallow Neural Network

Neural Network Representation

Neural Networks may be represented graphically, in most cases, they are comprised by the following:

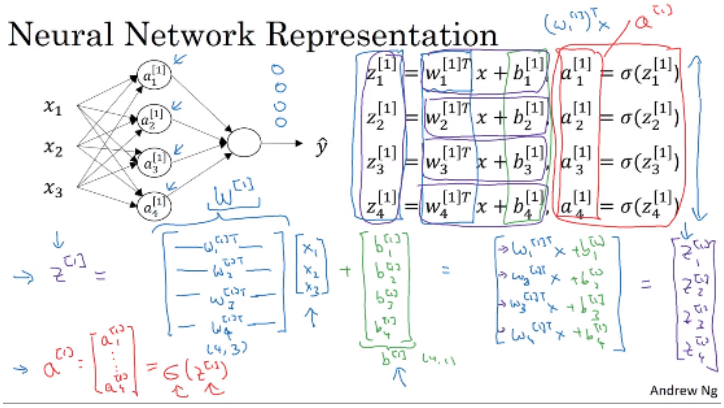


In a Neural network trained with supervised learning, a hidden layer refers to the fact that in the training set the true values for these nodes in the middle are not observed.
In the previous lessons, we saw that X was a variable to denote the input features; an alternative notation is simply $a^{[0]}$ that denotes the activations on the input features. The hidden layer would further make more activations that would be represented as $a^{[1]}$, and the values of the hidden nodes would be represented as $a^{[1]}_m$; if the Neural Network is a 2-layer network, the output would be classified as $a^{[2]}$. To count the layers of a NNET, the input layer is excluded; so a 2 layer NNET is really comprised of 3 layers.
In a 2 layer NNET, in layers 1 and 2 there are parameters associated with each node; for the hidden layer these are $w^{[1]}$ and $b^{[1]}$ and for the output $w^{[2]}$ and $b^{[2]}$.

Computing a Neural Network's Output

It's similar to logistic regression, but repeated multiple times. A Neural Network is very similar to Logistic Regression but computing z and the activations multiple times. The nodes on the hidden layer are composed by both operations mentioned before; these can be identified by the following notation: $a^{[l]}_i$, $z^{[l]}_i$, where l is the layer and i is the number of node in the layer. z and a must be calculated for all the nodes on the hidden layer of

the graph, thus, vectorization is required; the visual representation is the following:



The values of z and a depend directly on the values obtained from the previous layer, an example using layer 1 and 2 is represented as follows:
$z^{[1]} = W^{[1]}x + b^{[1]}$
$a^{[1]} = \sigma(z^{[1]})$

$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$
$a^{[2]} = \sigma(z^{[2]})$

Vectorizing Across Multiple Examples

Unvectorized version of the pseudocode for two layer NNET calculation:

```
for i = 1 to m:
    z[1](i) = W[1]x(i) + b[1]
    a[1](i) = σ(z[1](i))
    z[2](i) = W[2]a[1](i) + b[2]
    a[2](i) = σ(z[2](i))
```

Respecting the convention established where the capital letters correspond to the agglomeration of the respective "lowercase" values, the vectorization process can be developed. One thing to take into account is that A is a vector that goes horizontally from the first to the last training examples, and vertically goes from the first to the last hidden unit; another is that for Z is practically the same, but vertically it goes from the first to the last obtained feature.

The pseudocode for the vectorized process is the following:

```
Z[1] = W[1]X + b[1]
A[1] = σ(Z[1])
Z[2] = W[2]A[1] + b[2]
A[2] = σ(Z[2])
```

Activation Functions

In the previous lessons, the activation function was obtained through a sigmoid function, and in some cases there are other types of functions that can work better.
An activation function that works better in most cases is the hyperbolic tangent function. The formula for the tanh function is $\frac{e^z - e^{-z}}{e^z + e^{-z}}$ and it is a shifted version of the sigmoid function that goes from -1 to 1 instead of 0 to 1. This is useful because it "centers" the data and the mean could become closer to 0 instead of .5 which is particularly useful for the next layer.
The one exception where sigmoid function is better is the output layer because y is either 0 or 1. When using a different function than sigma for the activations, it's represented as g(z) instead of σ(z), and if there are more than one activation function, the convention is to difference them with the $g^{[i]}$notation.
The problem with these functions if z is too big or too small has to do with the derivative on both extremes; gradient descent would begin to slow down drastically due to the slope. In these cases another popular option is presented, the Rectifier Linear Unit which has the formula ($a = max(0, z)$). It has the advantage that the slope is always 1 when the z values are positive and 0 when it is negative. If binary classification is intended to be used, then the sigmoid function is recommended for the output layer, but for all other units the ReLU is increasingly becoming the default activation function.
The only concern with the ReLU function is that when the values are negative, the derivative is equal to 0; in practice this is not an issue but there are implementations called the "leaky" ReLUs that give the negative values a slope (although they are not as used in practice).
Recap of which functions to use and where/when to use them:

- Sigmoid: Never, except for the output layer when doing binary classification.
- ReLU: Most used function due to its constant gradient.
- Leaky ReLU: Maybe, but it's not common in practice.

For a NNET to compute an adequate result, it must use a non-linear function. If linear activation functions were used, the output would be a linear activation function of the input. The only place where linear activation functions could be seen is if the expected output is a non-binary value (house pricing prediction) and it would be included in the output calculation.

Derivatives of Activation Functions

For every activation function, we must know its respective derivative in order to implement descent.
$\sigma(z) = \frac{1}{1+e^{-z}} \implies \sigma'(z) = a(1-a)$

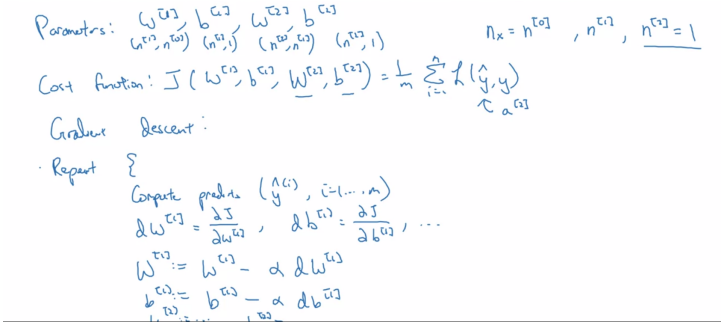
$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \implies tanh'(z) = 1 - (tanh(z))^2$

$R(z) = max(0, z), R'(Z) = z < 0 \implies 0, z > 0 \implies 1, z = 0 \implies$ UNDEF

$LR(z) = max(nz, z), LR'(z) = z < 0 \implies n, z > 0 \implies 1, z = 0 \implies$ UNDEF

Gradient Descent for Neural Networks

An iteration of the gradient descent on a two layer NNET example using binary classification can be the following:



When training a NNET it is important to initialize the variables in a random fashion rather than just 0s.
The trick in this exercise is to implement the formulas for the distinct values generated in the layers.

Assuming again that it's a binary classification problem, the formulas for forward propagation would be the following:

```
Z[1] = w[1]X + b[1]
A[1] = g[1] z([1])
Z[2] = w[2]A[1] + b[2]
// Assuming it's binary classification, sigmoid is used
Z[1] = g[2] z([2]) = σ(z[2])
```

And for back propagation:

```
dZ[2] = A[2] - Y
dW[2] = 1/m dZ[2]A[1]T
db[2] = 1/m np.sum(dZ[2], axis = 1, keepdims = True)

dZ[1] = W[2]TdZ[2] * g[1]'(z[1])
dW[1] = 1/m dZ[1]XT
db[1] = 1/m np.sum(dZ[1], axis = 1, keepdims = True)
```

Random Initialization

As mentioned before, it's very important to initialize the values of the NNET with random values rather than with 0s. If it's initialized as the logistic regression exercise, gradient descent would not be possible due to similar back and forward prop values.
The numpy function random.rand should be used for the $W^{[1]}$ values and the zero function can be used by both b values and by $W^{[2]}$.
It's recommended to initialize the random values with very small numbers because it's typically faster for the learning rate inside the sigmoid or tanh activation function.

Deep Neural Network

Chapter 2

Improving Deep Neural Networks

Chapter 3

Structure of Machine Learning Project

Chapter 4

Convolutional Neural Networks

Chapter 5

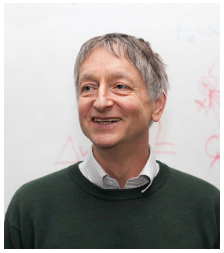
Natural Language Processing

Chapter 6

Heroes of Deep Learning

Additional to the course's materials, there were multiple encounters with some of the most famous deep learning "heroes" that have a great influence in today's world. The notes below are for em to remember these scientists and some of their work.

Geoffrey Hinton



Known as *The Godfather of Deep learning*, Geoffrey was inspired by indulging in how the brain stored memories. After attempting to understand it by studying it through physiological, psychological and philosophical fields, he found it lacking a complete explanation.
After some time he studied AI in Edinburgh where he studied Langer Higgin's theses on neural networks which he found really interesting. While in Great Britain he had a lot of trouble finding a job, in California he encountered more open-minded scientists that were interested in NNETs.
Although it's common belief that him and David Rumelhart invented the back propagation algorithm in 1982, many other scientists had already developed their own implementation but were not recognised as much. This paper combined two completely different strands on how the functionality of the brain which made it rater interesting. Stuart Sunderland was particularly impressed because the back propagation algorithm had the capability of generating feature vectors from information; this was rather useful because with the output features, it could generate new information (with the graph you can get a feature vector, and with this same vector you could derive new consistent graph-like structures). One of the most impressive examples consisted of an English text as an input that led to the generation of English words.
His proudest creation was the Boltzmann machine with Terry Sejnowski, these were discovered by improving an algorithm on big density connected NNETs where only few nodes could be seen and its purpose would be to discover the hidden nodes bu forward and backward passes. Further, in 2007, he found out that the features obtained by using Boltzmann machines could be used as input data, so this hidden layer could help discover the others effectively. It was discovered that by combining Boltzmann machines and using sigmoid belief nets, the process could be immensely improved in its efficiency.Also, thanks to the development based on ReLUs the algorithms developed by Hinton's group became very used and are now common in everyday NNET development.
"If it turns out the back prop is a really good algorithm for doing learning. Then for sure evolution could've figured out how to implement it. I mean you have cells that could turn into either eyeballs or teeth. Now, if cells can do that, they can for sure implement back propagation and presumably this huge

selective pressure for it." In 1987 he had an idea that the information inside the brain is sent through a recirculation algorithm that takes an idea goes through a loop where the information stays the same as it circles around, this process was assimilated to synaptic functions inside of the brain. Some time later, neuroscientists implemented the same algorithm but the other way around (where new memory is good and old memory is bad).

Geoffrey Hinton has been known to come with ideas that nobody believes in, he develops papers that most of the times get rejected and does not stop until he gets a publication out. For instance, he proposed that the neurons of a hidden layer could be grouped by some criterion when commonly in NNETs the hidden neurons are contained in the same layer without relation. Hinton proposes an extra structure for the relations to be represented with capsules that contain various neurons with similar characteristics. *"So let's suppose you want to do segmentation and you have something that might be a mouth and something else that might be a nose. And you want to know if you should put them together to make one thing. So the idea should have a capsule for a mouth that has the parameters of the mouth. And you have a capsule for a nose that has the parameters of the nose. And then to decipher whether to put them together or not, you get each of them to vote for what the parameters should be for a face."*
Geoffrey mentions that unsupervised learning is crucial in the long run, but you must face reality with the technological advancements we have today (almost anyone has any idea of how to develop unsupervised systems).
"Most people say you should spend several years reading the literature and then you should start working on your own ideas. And that may be true for some researchers, but for creative researchers I think what you want to do is read a little bit of the literature. And notice something that you think everybody is doing wrong, I'm contrary in that sense. You look at it and it just doesn't feel right. And then figure out how to do it right."
"When you have what you think is a good idea and other people think is complete rubbish, that's the sign of a really good idea."
"And so I think thoughts are just these great big vectors, and that big vectors have causal powers. They cause other big vectors, and that's utterly unlike the standard AI view that thoughts are symbolic expressions. "

Pieter Abbeel

He is a Belgian computer scientist, most of is recent work is related to deep reinforcement learning. Before working on deep reinforcement, he collaborated with Andrew Ng to work on reinforcement learning for autonomous helicopter flight. For all the implementations on this field, a lot of time was required to successfully implement a solution; it was until 2012 with Geoffrey Hinton's breakthrough on supervised learning that made him realize the benefits of using deep learning in his projects.
He thinks that the future of Deep Reinforcement Learning is vast due to the questions that it rises. While in supervised learning it's more about the IO mapping, reinforcement learning there is the notion of "Where does the data even come from?". The biggest challenge nowadays is how to get the systems to reason over long time horizons, a 5 second skill is very different to a 5 day skill; "How do you learn safely and also how do you keep learning once you're already pretty good".
He came up with an interesting idea to help the programs on the exploration steps: "Imagine, you have a reinforcement learning program, whatever it is, and you throw it out some problem and then you see how long it takes to learn. And then you say, well, that took a while. Now, let another program modify this reinforcement learning program. After the modification, see how fast it learns. If it learns more quickly, that was a good modification and maybe keep it and improve from there."
He recommends to try building things using Tensorflow, Channer, Pyano, Pytorch or other frameworks rather than just reading and following steps.

Ian Goodfellow

With Ethan Dreifuss he built one of the first GPU CUDA-based machines at Stanford in order to run Watson machines. In the development he figured out that the capabilities of Deep Learning were immense and that it was "the way to go".
His invention of GANs (Generative Adversarial Networks) have been influential to many modern industries and developments. GANs are a way of doing generative modeling, it's used when there is a lot of training data and it's desired to produce more examples resembling the original input datum. With a vast knowledge on Boltzmann machines and other frameworks, he worked on one that could avoid the disadvantages of the established models.
GANs are used nowadays to populate other models with generated data and even simulating scientific developments. Many other models can work in these cases, but this is due to the early stages that GANs are in. If GANs become as reliable as Deep Leaning has become, Ian hopes that people will use GANs more commonly and in a more successful fashion. Most of his work today consists of stabilizing the Networks to achieve this goal.
His book [1] has helped many initiates to understand the basic intuition of probability and linear algebra.

Bibliography

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. The MIT Press, 2016.