

COURSERA

Notes

Neural Networks and Deep Learning

Carlos García González

Summer 2020

Contents

1 Neural Networks and Deep Learning	5
Introduction to Deep Learning	6
Logistic Regression as a Neural Network	9
Python and Vectorization	16
Shallow Neural Network	20
Deep Neural Network	25
2 Improving Deep Neural Networks	31
Setting up Machine Learning Application	32
Regularization	34
Setting Up an Optimization Problem	37
Optimization Algorithms	40
Hyperparameter Tuning	46
Batch Normalization	48
Multi-class Classification	50
Programming frameworks	50
3 Structure of Machine Learning Project	53
ML Strategy	54
Orthogonalization	54
Setting Up The Goal	54
Human-Level Performance	57
Error Analysis	60
Mismatched Training and Dev/Test Set	62
Learning from Multiple Tasks	65
End-to-end Deep Learning	67
4 Convolutional Neural Networks	69
Convolutional Neural Networks	70
Case Studies	76
Practical Advises for ConvNet Implementation	80
Detection Algorithms	81
Face Recognition	85
Neural Style Transfer	86
5 Sequence Models	89
Recurrent Neural Networks	90

6 Heroes of Deep Learning	99
Geoffrey Hinton	100
Pieter Abbeel	102
Ian Goodfellow	103

Chapter 1

Neural Networks and Deep Learning

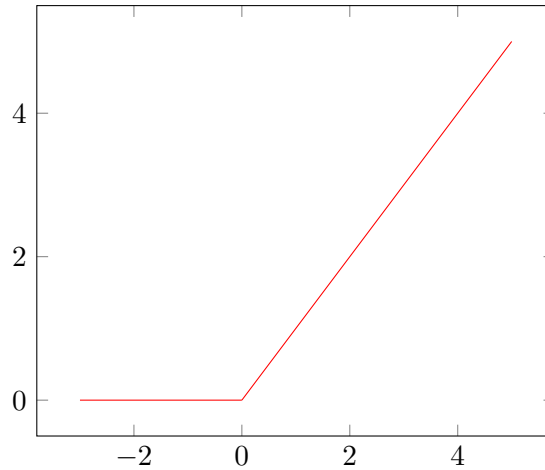


Figure 1.1: Plot of an example RELU function

Introduction to Deep Learning

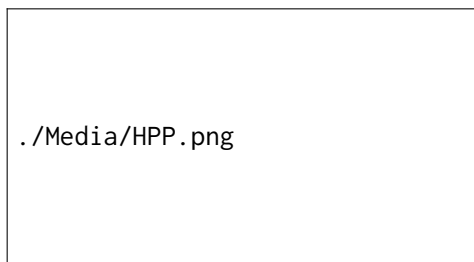
What is a Neural Network?

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

Size x \rightarrow Neuron (does the determined function) \rightarrow 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 let's add more variables for the same problem, the representation of the network would be similar to the following graph:



The inter-connecting nodes 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” nodes represent hidden features, these are the ones created subsequent to the input data and the network itself defines the relation between these.

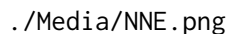
The network processes two different types of data:

- 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 \rightarrow code \rightarrow experiment) that leads to more architecture revisions.

Logistic Regression as a Neural Network

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:

$$X = \begin{pmatrix} \dots & \dots & \dots & \dots \\ x^1 & x^2 & \dots & x^m \\ \dots & \dots & \dots & \dots \end{pmatrix} \quad (1.1)$$

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:

$$Y = \{y^1 \quad y^2 \quad \dots \quad y^m\}, Y \in \mathbb{R}^{1 \times m} \quad (1.2)$$

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 is a modification of $w^T x + b$ so that the range does not exceed 1 or be smaller

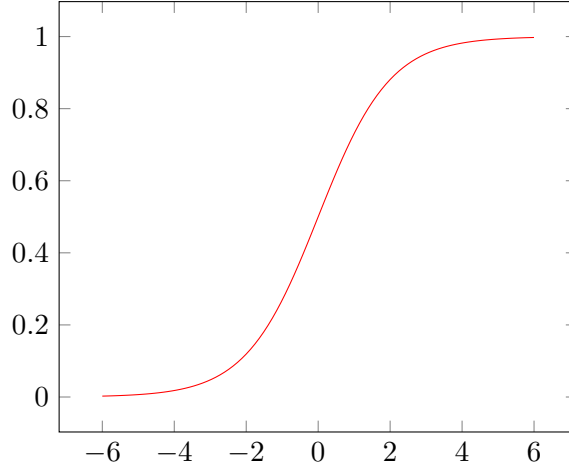


Figure 1.2: Plot of an example sigmoid function

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: $\mathcal{L}(\hat{y}, y) = -(y \log \hat{y} + (1 - y) \log(1 - \hat{y}))$. 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 order to check the complete training set a cost function must be used. The

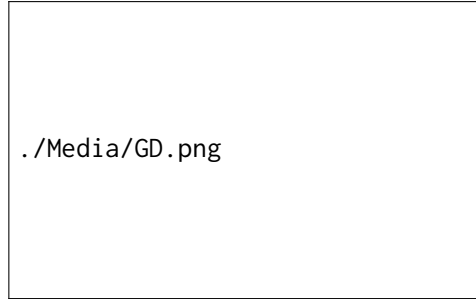


Figure 1.3: Plot of a gradient descent

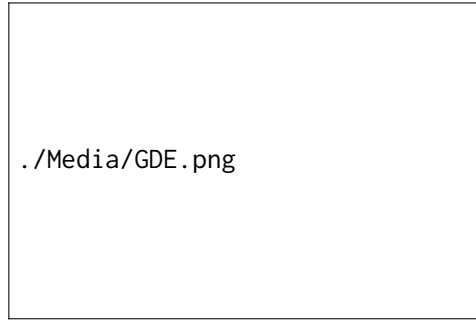


Figure 1.4: Example of a bidimensional gradient descent

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{  
     $w := w - \alpha \frac{dJ(w)}{dw}$   
}
```

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 $J(w, b)$:

$$w := w - \alpha \frac{\partial J(w, b)}{\partial w}, b := b - \alpha \frac{\partial J(w, b)}{\partial b} \quad (1.3)$$

Computation Graph

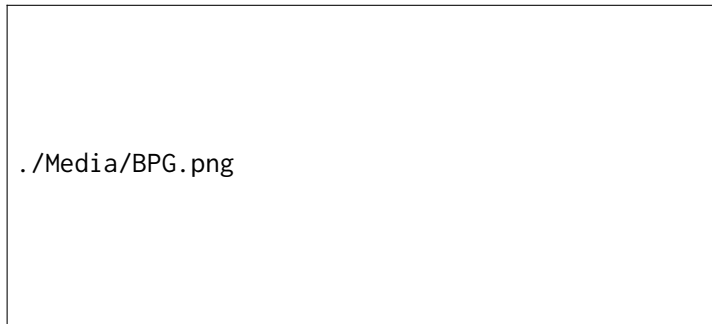
Computation of a NN are organized by forward pass/forward propagation in which the output is calculated, followed by a backward pass/propagation step in which the gradients are computed; the computation graph is a representation in which it's shown why it is done this way. A computation graph is a graphical 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.

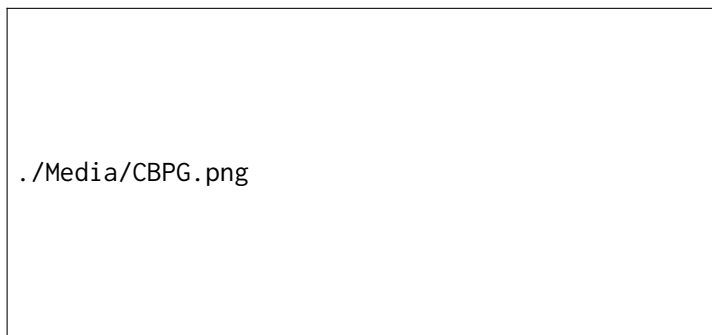
Derivatives with a Computation Graph

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

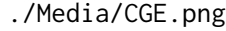
Using the computation graph for gradient descent is rather overkill, but easier to begin to understand the concept. Recap of the functions used by the logistic regression:

$$z = w^T x + b \quad (1.4)$$

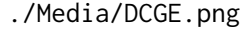
$$\hat{y} = a = \sigma(z) \quad (1.5)$$

$$\mathcal{L}(a, y) = -(y \log(a) + (1 - y) \log(1 - a)) \quad (1.6)$$

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 dw_m in the code), $db = dz$ and by modifying the values in the following fashion:

$$w_m := w_m - \alpha dw_m \quad (1.7)$$

$$b := b - \alpha db \quad (1.8)$$

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)
    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
```

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)=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)
    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

$[z^{(1)}, z^{(2)}, \dots, z^{(m)}]$ and is the result of $w^T X + [b, b, \dots, b]$ that would generate an output $[w^T x^{(1)} + b, w^T x^{(2)} + b, \dots, w^T x^{(m)} + b]$ 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.TX + 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`

Logistic Regression with a Neural Network Mindset

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

`./Media/Broadcast.png`

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 NNs 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).

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.

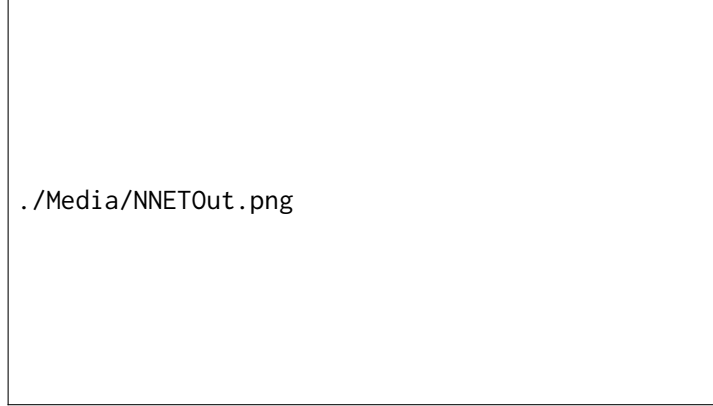
In the previous lessons, we saw that X was a variable to denote the input features; an alternative notation is simply $a^{[0]}$ that denotes the activations on the input features. The hidden layer would further make more activations that would be represented as $a^{[1]}$, and the values of the hidden nodes would be represented as $a_m^{[1]}$; if the Neural Network is a 2-layer network, the output would be classified as $a^{[2]}$. To count the layers of a NN, the input layer is excluded; so a 2 layer NN is really comprised of 3 layers.

In a 2 layer NN, 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: $a_i^{[l]}$, $z_i^{[l]}$, where l is the layer and i is the number of node in the layer. The values for 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 NN 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:

$$\begin{aligned}Z^{[1]} &= \mathbf{W}^{[1]}X + \mathbf{b}^{[1]} \\A^{[1]} &= \sigma(Z^{[1]}) \\Z^{[2]} &= \mathbf{W}^{[2]}A^{[1]} + \mathbf{b}^{[2]} \\A^{[2]} &= \sigma(Z^{[2]})\end{aligned}$$

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 hyperbolic tangent 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 $\sigma(z)$, 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 NN 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.

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

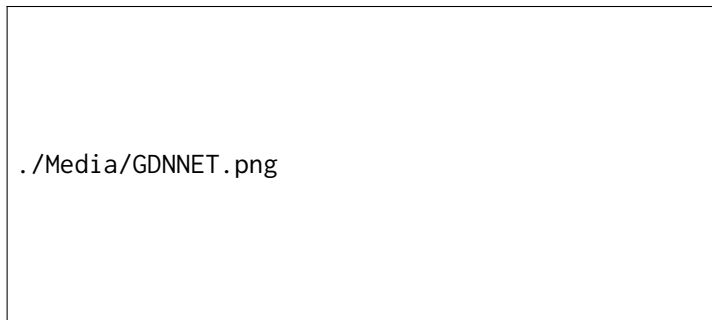
$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \implies \tanh'(z) = 1 - (\tanh(z))^2$$

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

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

Gradient Descent for Neural Networks

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



When training a NN 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:

$$\begin{aligned} Z^{[1]} &= W^{[1]}X + b^{[1]} \\ A^{[1]} &= g^{[1]}(Z^{[1]}) \\ Z^{[2]} &= W^{[2]}A^{[1]} + b^{[2]} \\ \# \text{ Assuming it's binary classification, sigmoid is used} \\ Z^{[1]} &= g^{[2]}(Z^{[2]}) = \sigma(Z^{[2]}) \end{aligned}$$

And for back propagation:

$$\begin{aligned} dZ^{[2]} &= A^{[2]} - Y \\ dW^{[2]} &= \frac{1}{m} dZ^{[2]} A^{[1]T} \\ db^{[2]} &= \frac{1}{m} \text{np.sum}(dZ^{[2]}, \text{axis} = 1, \text{keepdims} = \text{True}) \\ dZ^{[1]} &= W^{[2]T} dZ^{[2]} * g^{[1]'}(Z^{[1]}) \\ dW^{[1]} &= \frac{1}{m} dZ^{[1]} X^T \\ db^{[1]} &= \frac{1}{m} \text{np.sum}(dZ^{[1]}, \text{axis} = 1, \text{keepdims} = \text{True}) \end{aligned}$$

Random Initialization

As mentioned before, it's very important to initialize the values of the NN 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 the hyperbolic tangent activation function.

Deep Neural Network

Deep L-Layer Neural Network

A deep neural network consists of a NN containing many hidden layers. Logistic regression can be seen as a shallow NN and models containing multiple hidden layers are considered to be deep.

On a deep NN there is a similar notation as the previously-seen models, and it consists of the following:

- L : Number of layers.
- $n^{[l]}$: Number of units in the l layer. Also, for the output layer it's also valid to write $n^{[L]}$.
- $a^{[l]}$: Number of activations in the l layer; and of course, the activation on the final layer is the output \hat{y} .

Forward Propagation in a Deep Network

The principle for forward propagation works the same way as the last chapter's example, the only difference is that, in a deep NN, there are more layers (but it's pretty much the same thing). The general rule is the following:

$$\begin{aligned}z^{[l]} &= w^{[l]}a^{[l-1]} + b \\ a^{[l]} &= g^{[l]}(z^{[l]})\end{aligned}$$

And in order to vectorize this for a whole training set, it's the same principle as before:

$$\begin{aligned}Z^{[l]} &= W^{[l]}A^{[l-1]} + b \\ A^{[l]} &= g^{[l]}(Z^{[l]})\end{aligned}$$

There needs to be an explicit for loop in order to compute the activations of each layer.

Correct Matrix Dimensions

It's very important to get the correct dimensions for the matrices, one way to check this out is to "grab a piece of paper" and to execute the following procedure.

Take into account the number of layers L (excluding the input layer), the number of nodes inside each layer $n^{[l]}$ (including input layer). Having this and the notion of forward propagation in mind, the dimensional values for z and a

would be $(n^{[l]}, 1)$, for x it would be $(n^{[0]}, 1)$, and having matrix multiplication in mind it's easy to see that for $W^{[l]}$ the dimensions are $(n^{[l]}, n^{[l-1]})$. For b , it would simply be the broadcast of the value in a $(n^{[l]}, 1)$ vector and the derivatives dW and db would be equal to their respective origin values. For the values of X , Z and A , the dimensions will be modified to $(n^{[0]}, m)$ and $(n^{[l]}, m)$ respectively due to the various training examples.

Why Deep Representations?

In NNs, the first layers compute simple features to then use them and calculate more complex data. Compositional representation is found in many of today's data, so deep NNs are very useful due to the "simple to complex" nature of the learning process.

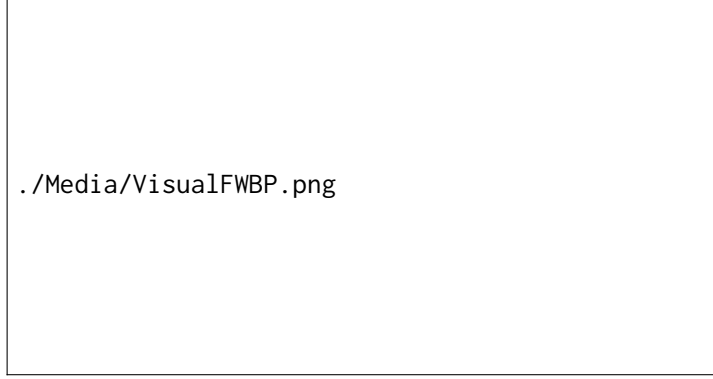
Circuit theory and deep learning have a very peculiar similarity while comparing deep learning and logical gate calculations: there are functions that you can compute with a relatively small L-layer deep NN that shallower networks require exponentially more hidden units to compute (from $O(\log n)$ to roughly 2^n).

Andrew Ng often starts by implementing shallow NNs and by incrementing the number of layers (as a hyperparameter) in order to determine the L "sweet-spot".

Building Blocks

Forward propagation: for layer l , there will be a $w^{[l]}$ and a $b^{[l]}$, there is an input generated by the previous layer $a^{[l-1]}$ and there will be an output $a^{[l]}$. The calculations inside the layer will be $z^{[l]} = w^{[l]}a^{[l-1]} + b^{[l]}$ and $a^{[l]} = g^{[l]}(z^{[l]})$; also, for the back propagation step, the value of $z^{[l]}$ must be cached. *Backward propagation:* for layer l , there will be an input $da^{[l]}$ and an expected output $da^{[l-1]}$. The cached $z^{[l]}$ value is available and the calculation of the values $dw^{[l]}$ and $db^{[l]}$ should also be taken into account.

A simple graphical representation of this process is exemplified in the following graphic:



Forward and Backward Propagation

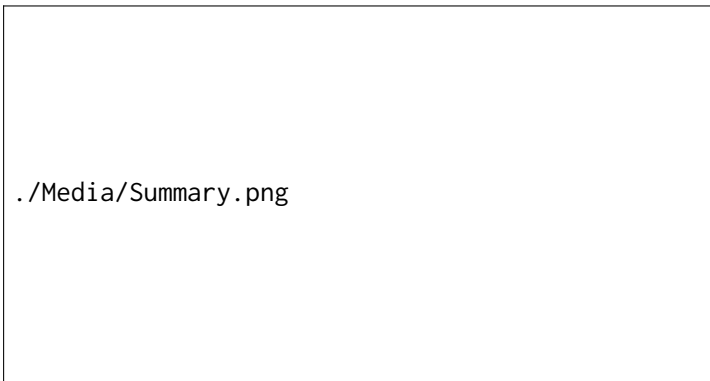
Forward propagation for layer l :

- Input $a^{[l-1]}$
- Output $a^{[l]}$, cache $z^{[l]}, w^{[l]}, b^{[l]}$
- Functions
 - $z^{[l]} = w^{[l]}a^{[l-1]} + b^{[l]}$
 - $a^{[l]} = g^{[l]}(z^{[l]})$

Backward propagation for layer l :

- Input $da^{[l]}$
- Output $da^{[l-1]}, dW^{[l]}, db^{[l]}$
- Functions
 - $dz^{[l]} = da^{[l]} * g^{[l]'}(z^{[l]})$
 - $dW^{[l]} = dz^{[l]}a^{[l-1]T}$
 - $db^{[l]} = dz^{[l]}$
 - $da^{[l-1]} = W^{[l]T}dz^{[l]}$

A visual summary of these processes can be found below:



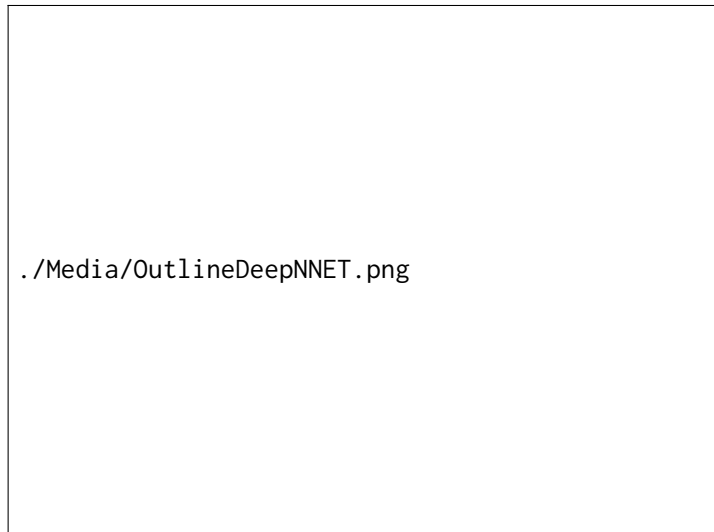
Parameters and Hyperparameters

The parameters in a NN are the values contained in W and the value b and hyperparameters allude to values such as the learning rate α , the number of iterations, the number of hidden layers L and hidden units per layer $n^{[l]}$, and choice of activation function.

A hyperparameter is defined as a value or architectural decision that controls directly the behavior of the system's parameters.

The experimentation on hyperparameters is crucial in order to tune the values and improve the learning rate in the system. Another thing to take into account is that the learning rate may vary from one moment to another due to external factors such as the computer architecture.

Complete Forward and Backward Propagation Visual Aid



Chapter 2

Improving Deep Neural Networks

Setting up Machine Learning Application

Training, Development and Test Sets

In order to develop a good ML algorithm, it is crucial to tune in the best way possible the hyperparameters (number of layers, number of hidden units, learning rates, the activation functions and many other). It's a matter of experimenting with these, there are no particular set of rules that apply to every case so it's really an "Idea -> Code -> Experiment" cycle.

A typical division on datasets has three sections: the training set, the hold-out cross validation or development set, and the test set. The first two are used while developing the initial models, when the most adequate is developed it is tested using the test set values. Before the Big Data era, it was perfectly reasonable to split the sections into 60/20/20 (Train/Dev/Test) but in modern implementations the last two have become smaller when datasets become "too large". It's also important to make sure that the dev and test set data come from the same distribution due to a possible error called the mismatched test/train distribution. Another thing worth noting is that it's possible to not have a test set, in some cases a set divided by training and development subsets would be sufficient.

Bias/Variance

Bias and variance play a very important role in machine learning practices. Essentially, with high bias the functions generated would not likely be representative of the system nature, and with high variance it would generate models difficult to generalize the behavior of the system in an understandable fashion.

In practice, there are techniques that can help identify high variance or high bias.

- Training set error < Dev set error \implies High variance.
- Assuming that we want the error on the training set to be close to 0, if it is a very large number, it indicates that the bias is too high because it's under-fitting the data.
- If both of the previously mentioned conditions are presented, the intuition would point to high variance and high bias values.


These indexes are subjected to the desired optimal error value, this could come handy for examples such as difficult-to-comprehend data such as blurry images.

Basic Recipe for Machine Learning

Simple “recipe” to improve algorithm.

1. *High bias (Training set performance)*: **Try a bigger network**, more hidden units or hidden layers, train it longer or even try a different NN architecture. Do it until the error is gone and the training set fits well with the system.
2. *High variance (Dev set performance)*: **Obtain more data**, regularize the previously processed data to reduce over-fitting or try a different NN architecture.

In the big data era, the concept known as “Bias variance trade-off” has been eliminated due to the bigger network option and the possibility to obtain more data.



./Media/HBHV.png

Regularization

If the network is presenting a high variance issue, one of the most common practices is to regularize the data in order to reduce over-fitting and not requiring new data that can sometimes be limited.

On logistic regression, it is desired to minimize the cost function J and to regularize this function you would have to add some extra values at the end of the function; it would be the following:

$$J(w, b) = \frac{1}{m} \sum_{i=1}^m \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \|w\|_2^2 \quad (2.1)$$

$$\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w \quad (2.2)$$

The previously shown formula refers to the L_2 regularization, this is because it uses the Euclidean norm known as the L_2 norm. In practice it's commonly to regularize the w values but not the b values due to its nature. L_2 is the most common type of regularization, but plenty of others exist such as the L_1 regularization that is the following:

$$\frac{\lambda}{m} \sum_{i=1}^{n_x} w_i = \frac{\lambda}{m} \|w\|_1 \quad (2.3)$$

If L_1 is used, the data would be sparse which means that many zeros will be encountered in the dataset, it's not used a lot in practice.

λ is the regularization parameter which is set using the development set or cross validation, it's a hyperparameter that must be tuned. In programming, it is represented as "lambd" due to "lambda" being a Python reserved call.

Modifying the previous function for J :

$$J(w, b) = \frac{1}{m} \sum_{i=1}^m \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_{l=1}^L \|w^{[l]}\|^2 \quad (2.4)$$

$$\|w^{[l]}\|^2 = \sum_{i=1}^{n^l} \sum_{j=1}^{n^{[l-1]}} (w_{i,j}^{[l]})^2 \quad (2.5)$$

Analytically, the last "loop" makes sense because the dimensions of w are $(n^{[l]}, n^{[l-1]})$ where n corresponds to the hidden units on the respective l layer.

This matrix is known as a "Frobenius Norm" that it's denoted by the superscript on the w values ($\|xyz\|_F^2$).

For back propagation there is a modification required due to the extra terms for the regularization. For this, it is only necessary to add $\frac{\lambda}{m}w^{[l]}$ to the previous definition of dw .

Regularization reduces over-fitting because lambda modifies directly the $W^{[l]}$ values, if we set the weight value of some units to be close to 0 it would rule out the impact of these (resulting in a simpler network), leaving an over-fitting network to become more of a high bias case (of course, calibrating λ to become the “just right” model).

The takeaway of the regularization intuition is the following: If the λ regularization parameter is large, then the $W^{[l]}$ values would output small values due to the penalization; $Z^{[l]}$ in consequence would also be small and if $g(z)$ is the $\tanh(\theta)$ function, every layer would act like a linear function and the NN would practically function as a linear NN.



Dropout Regularization

It works by running through each layer and setting a probability of eliminated a node in a NN. Removing the nodes and their respective relations to other nodes would lead to a smaller NN.

To implement dropout, there is a common technique called the “Inverted Dropout”; a pseudocode for this is the following:

```
# Example for the 3rd layer of a NN
# keep_prob = .8 (prob that node is not removed)

# Boolean array containing the existence of the nodes
d3 = np.random.randn(a3.shape[0], a.shape[1]) < keep_prob

# Zeroes out the corresponding elements of d3
a3 = np.multiply(a3, d3)

# Scale activations, inverted d.o. technique
a3 /= keep_prob
```

Inverted dropout is one of the most common dropout techniques due to its lack of re-scale processes on the latter layers.

At test time there is no dropout calculations because it would generate noise in the output and it would give different results each time. A thing worth noting is that the previously seen `keep_prob` value may vary between each layer.

Understanding Dropout


Intuition: Can't rely on any feature, so have to spread out weights.

The intuition tells us that the calculations on a neuron can't be dependent on a single value, cancelling them in dropout eliminates the codependence and distributes the weights. The downside of dropout is that the J function is no longer well-defined, making the gradient descent revision process quite harder to evaluate; dropout should be "turned off" while checking if the descent is effective and should be "turned on" right after that.

Other Methods

Data augmentation: Modifying the dataset values in a subtle way so that it can be used as a training example as well (flipping images horizontally or random crops). This is not very effective but it's quite inexpensive.

Early stopping: As gradient descent is running, the idea is to plot the number of iterations against the calculated training error or J and at the same time plot the dev set error. Dev set usually starts to go down but suddenly starts to go up again, the idea is to stop training the NN at that point in time.



Setting Up an Optimization Problem

Normalizing Inputs

This process can improve the efficiency on a DL algorithm and consists of two steps; subtracting the mean so the data “goes down” to the horizontal axis, and the second step consists of normalizing the variances.

Mean subtraction...

$$\mu = \frac{1}{m} \sum_{i=1}^m x^{(i)} \quad (2.6)$$

$$x := x - \mu \quad (2.7)$$

Variance normalization...

$$\sigma = \frac{1}{m} \sum_{i=1}^m x^{(i)} * * 2 \text{ (element-wise multiplication)} \quad (2.8)$$

After this process, the variance in both axes will be equal to 1. It’s important to be consistent in the μ and σ values between the training and test set.

If input features are not normalized, the values between training examples can vary immensely due to the range differences, and the cross sections of the gradient descent function would be more of an oval shape rather than a circle, implying that the learning rate should be smaller.

Vanishing / Exploding Gradients

Whenever we encounter a very deep Neural Network, the slope of the function can become extremely big or exponentially slow which makes training really difficult; this is known as exploding gradients. This generates a barrier for the learning process but fortunately there is a partial solution to this problem, that is, the careful process of weight initialization.

Weight Initialization for Deep Networks

For a single neuron that computes an activation function of z , $g(z)$, we have to take into account the number of inputs. If the number of inputs is very large, we would necessarily need to have very small w values for z to not blow up ($z = w_1x_1 + w_2x_2 + \dots w_nx_n + b$).

A sensible thing to implement would be that the variable generation would

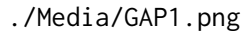
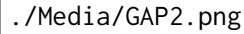
acquire a $\frac{1}{n}$ fashion, the only modification for ReLUs particularly would be to change this to a $\frac{2}{n}$ ratio. This would translate to the following implementation:

```
# np.sqrt (...) == 2/n
W[l] = np.random.randn(slope) * np.sqrt( $\frac{2}{n^{[l-1]}}$ )
```

This doesn't solve the explosive gradient problem but it certainly aids to reduce the impact on the learning rate. As mentioned previously, this is intended for ReLU based functions but there are other implementations for other functions such as $\tanh(\theta)$ that would be represented by the following: $\sqrt{\frac{1}{n^{[l-1]}}}$). Another common practice is to use a hyperparameter inside this function, let it be "mp" in this example:

$$\sqrt{\frac{mp}{n^{[l-1]}}} \quad (2.9)$$

Numerical Approximation of Gradients

Gradient Checking

First step of gradient checking is to take all the W and b values and reshaping them into a big vector θ , first it would reshape them and later concatenate them into the same value; the same can happen for the derivative values dW and db ($d\theta$). The function of $J(W^{[1]}, b^{[1]}, \dots, W^{[L]}, b^{[L]})$ would now be represented as $J(\theta)$.

To implement grad check, it has to go through the following process:

```
for each i:
    # The following equation is an approximate of:
    #  $\approx d\theta[i] = \frac{\partial J}{\partial \theta_i}$ 
     $d\theta[i] = \frac{J(\theta_1, \theta_2, \dots, \theta_i + \epsilon, \dots) - J(\theta_1, \theta_2, \dots, \theta_i - \epsilon, \dots)}{2\epsilon}$ 
```

The crucial section is the one that calculates if these gradients are in fact equivalent, this can be performed by simply calculating the euclidean distance of both values:

$$\frac{||d\theta_{\text{approx}} - d\theta||_2}{||d\theta_{\text{approx}}||_2 + ||d\theta||_2} \quad (2.10)$$

A recommended value for ϵ is 10^{-7} , if the same value is shown in the previous function it's great, 10^{-5} is good enough but 10^{-3} is worrying.

Recommendations:

- This process is not recommended for training, only to debug due to its slow computing time.
- If algorithm fails check, look at components to identify bug.
- Remember regularization term.
- Doesn't work with dropout.
- Run at random initialization; perhaps again after some training.

Optimization Algorithms

Mini-batch Gradient Descent

Having good optimization algorithms is crucial in today's world in order to develop faster solutions.

Vectorization allows the user to efficiently compute on m training examples, but if this number is still very large it would still take a lot of time to process. Mini-batch gradient descent splits the training set in multiple subsets called mini-batches.

To differentiate each batch, the curly braces are used as an identifier as follows: $X^{\{t\}}$ with its respective $A^{\{t\}}$ value (the value within the brackets represents a batch). The generic notation uses the variable t to denote the mini-batch, as follows:

$$X^{\{t\}}, Y^{\{t\}} \quad (2.11)$$

The pseudocode for the process would be the following:

```
for t = 1, ..., T
    # 1 step of descent on X/T

    #Forward prop on  $X^{\{t\}}$ 
     $Z^{[1]} = W^{[1]}X^{\{t\}} + b^{[1]}$ 
     $A^{[1]} = g^{[1]}(Z^{[1]})$ 
     $\vdots$ 
     $A^{[L]} = g^{[L]}(Z^{[L]})$ 
    # There should be vectorization for the previous
    # process, also, this is only for X/T examples

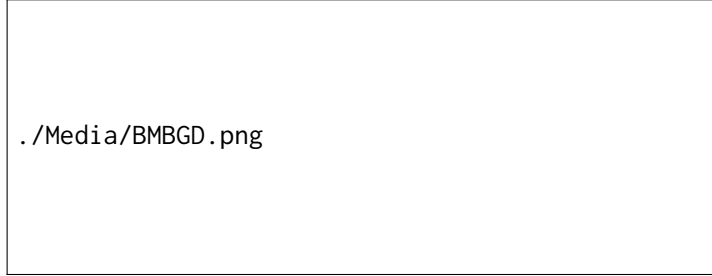
    # Compute cost (only for one mini-batch)
     $J = (X/T) \sum_{i=1}^l \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2 * X/T} \sum_l \|w^{[l]}\|_F^2$ 

    # Backprop
    # Update values of  $w$  and  $b$  with respect to  $\alpha$  and
    # derivatives
```

Essentially it's the same as the already seen examples but with the division on the training examples. An iteration over one mini-batch is also known as an epoch.

A difference between the batch and mini-batch descent is that, if implemented correctly, the cost vs iteration graph would look substantially differ-

ent because in the mini-batch, each epoch computes different sets of data on each iteration; it would look something like the following representation:



The size of the batches is substantial for the algorithm's operation, if it's too high (translates into batch descent) or if it's too small (many mini-batches, becomes a stochastic gradient descent) the learning could become slower. The mini-batch size should be an in-between value from 1 to m that helps the algorithm progress in a better fashion and does not represent a disadvantage in the learning. To choose the value there are some guidelines:

- If small training set, use batch gradient descent ($m > 2000$).
- Else, typical batch sizes go from 64-512 and typically are in the order of powers of 2 due to memory storage. It's important to verify that these fit in the CPU/GPU memory.

Exponentially Weighted Averages

In order to implement faster algorithms than gradient descent, the concept of exponentially weighted averages has to be fully understood. For datasets with very sparse data it's difficult to discover trends until a procedure is performed, it follows the pattern below:

$$\begin{aligned}
 V_0 &= 0 \\
 V_1 &= 0.9V_0 + 0.1\theta_1 \\
 V_2 &= 0.9V_1 + 0.1\theta_2 \\
 &\vdots \\
 V_t &= 0.9V_{t-1} + 0.1\theta_t
 \end{aligned} \tag{2.12}$$

This will create a tendency graph, and the generalization is the following:

$$V_t = \beta V_{t-1} + (1 - \beta)\theta_t \tag{2.13}$$

Where β is a parameter which will affect the number of samples taken into account for the average. Let s be the number of samples, to calculate the impact of β regarding the sample size the following formula will help to get an approximation:

$$n \approx \frac{1}{1 - \beta} \quad (2.14)$$

By varying the β hyperparameter the results can change drastically, that's why it's very important to tune this value adequately. The pseudocode for the mentioned process is the following:

```

Vθ = 0
Repeat
  Get next θt
  Vθ := βVθ + (1 - β)θt

```

Bias Correction for Weighted Averages

When implementing a moving weighted average, the value for V_θ is initialized as 0; so if the formula was maintained as the previously-seen example, the value for V_1 would be extremely low (not very good estimate). A modification is required to develop a better estimate for the initial values, this is called bias correction and follows the modification:

$$\frac{V_t}{1 - \beta^t} \quad (2.15)$$

This removes the bias on the initial stages and later becomes insignificant as the algorithm moves forward.

Gradient Descent with Momentum

Almost every time works faster than gradient descent, the basic idea is to compute an exponentially weighted average of the gradients and then use that gradient to update the weights. For momentum to be implemented, it should calculate the following procedures:

```

On iteration t:
  Compute dW, db on current mini-batch
  VdW = βVdW + (1 - β)dW
  Vdb = βVdb + (1 - β)db
  W := W - αVdW, b := b - αVdb

```

What this process does is that it “smooths out” the gradient descent. In this case, there are two hyperparameters to take into account and are α and β .

RMSprop

Root mean square prop is used to speed up gradient descent. It goes through the following process:

```
On iteration t:
  Compute dW, db on current mini-batch
   $S_{dw} = \beta S_{dw} + (1 - \beta) dW^2$ 
   $S_{db} = \beta S_{db} + (1 - \beta) db^2$ 
  #  $\epsilon$  can have any small value,  $10^{-8}$  recommended
   $W := W - \alpha \frac{dW}{\sqrt{S_{dw} + \epsilon}}$ 
   $b := b - \alpha \frac{db}{\sqrt{S_{db} + \epsilon}}$ 
```

The benefit from using algorithms such as the RMSprop is that the learning rate can be increased due to the smaller gradients on the bias values.

Adam's Optimization

It's one of the few modern algorithms that has proven to be quite good in various NN architectures and scenarios; essentially it combines RMSprop and gradient descent with momentum. The pseudocode is the following:

```
 $V_{dw} = 0, S_{dw} = 0, V_{db} = 0, S_{db} = 0$ 
On iteration t:
  Compute dW, db using current mini-batch

  # Momentum-like procedure
   $V_{dw} = \beta_1 V_{dw} + (1 - \beta_1) dW$ 
   $V_{db} = \beta_1 V_{db} + (1 - \beta_1) db$ 

  # RMSprop
   $S_{dw} = \beta_2 S_{dw} + (1 - \beta_2) dW^2$ 
   $S_{db} = \beta_2 S_{db} + (1 - \beta_2) db^2$ 

  # Bias correction
   $V_{dw}^{corrected} = \frac{V_{dw}}{(1 - \beta_1^t)}$ 
   $V_{db}^{corrected} = \frac{V_{db}}{(1 - \beta_1^t)}$ 
   $S_{dw}^{corrected} = \frac{S_{dw}}{(1 - \beta_2^t)}$ 
   $S_{db}^{corrected} = \frac{S_{db}}{(1 - \beta_2^t)}$ 
   $W := W - \alpha \frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected} + \epsilon}}$ 
   $b := b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected} + \epsilon}}$ 
```

The hyperparameters used are the following:

- α : needs to be tuned to a sensible value.
- β_1 : moving average for dw , 0.9 is a very common value for it.
- β_2 : moving average for dw^2 , 0.999 is a recommended value.
- ϵ : doesn't really matter but it's recommended to be 10^{-8}

Adam stands for Adaptive moment estimation, these moments being the calculations for the β values.

Learning Rate Decay

You might want to reduce the learning rate value in order to better converge to the lower value on the gradient descent.

One thing that we may do is use the following update for the learning rate:

$$\alpha = \frac{1}{1 + \text{decay_rate} * \text{epoch_Number}} \alpha_0 \quad (2.16)$$

This will begin to lower the learning rate value as the algorithm goes through each epoch; the `decay_rate` value is also a hyperparameter that should be tuned.

There are other learning rate decay methods such as the exponential decay that is represented by the following expression:

$$\begin{aligned} n &\in \mathbb{R} | n < 1 \\ \alpha &= n^{\text{epoch-num}} \alpha_0 \end{aligned} \quad (2.17)$$

This will exponentially decay the value of the learning rate.

Problem of Local Optima

Not all points where the gradients are zero are local optimas, they may be saddle points that we have to take into account; this is more common in higher dimensional spaces. On the other hand, in a plateau there can be an error in which the descent “falls” from the saddle and into an erroneous path for the calculations.

Takeaways:

- It is unlikely to get stuck in a bad local optima (in a low-dimension problem).
- Plateaus can make learning slow.
- Optimization algorithms such as momentum help solve this problems.

Hyperparameter Tuning

Tuning Process

It is a painful process while training deepness in a Neural Network due to the amount of hyperparameters, which are:

1. Most important values to tune
 - α
2. Second most important to calibrate
 - β
 - Number of hidden units
 - Mini-batch size
3. Third most relevant values
 - Number of layers
 - Learning rate decay
4. The values for $\beta_1, \beta_2, \epsilon$ are not typically tuned.

A recommendation is to test the hyperparameters initially using random values in order to get an idea of the possible optimal value for a particular network. This is typically done on a tridimensional space where three hyperparameters are plotted into different points in a space, the objective is the find the coordinate values where it's optimal for both hyperparameters. The coarse to fine scheme “zooms in” the nodes with the optimal hyperparameter values so the process can be repeated.

Using Appropriate Scale to Pick Hyperparameters

For values such as α , it is much better to use a logarithmic scale instead of a linear scale due to the great computer resource impact that the latter has. After a log partition on a given range, the samples are selected randomly for each log interval. A sample implementation in Python is the following:

```
r = -4*np.rand() # r is in the range of [-4, 0]
 $\alpha = 10^r$  # Value from  $10^{-4}$  to  $10^0$ 
```

For β it's pretty much the same, we have to explore the sample of the range $[0, \beta]$, and this has to do with the sensitivity of the β value when it gets closer to 1.

Although many concepts of NNs have flowed through many fields and architectures, no problems behaves in the same fashion and the initialization of hyperparameters may vary in a huge way. It is also recommended to re-evaluate these values occasionally because they can get stale after some time.

There are two approaches to modify the values, babysitting one model and training many parameters in parallel. The first one consumes a lot of time but not many resources, and the latter works in the exact opposite fashion. To take a decision it is important to monitor the learning of the network in both cases and to modify the respective values in each scenario (modify hyperparameters for one model vs choosing the model that suits better after tuning hyperparameters in many models running in parallel).

Batch Normalization

Normalizing Activations in a Network

Similar to the concept of input normalization, normalizing the activations of each layer also improves the algorithm's performance significantly. There is a debate in the deep learning world that infers in *where* should the normalization go (before or after the activation is calculated); in practice, the normalization is seen more frequently on the values of Z (before the activation). The implementation is the following:

```
# Given intermediate values of a NN, exists  $Z^{[l](1)}, \dots, Z^{[l](m)}$   
# The following operations are in a specific layer
```

```
# Compute the mean
```

$$\mu = \frac{1}{m} \sum_i Z^{(i)}$$

```
# Compute variance
```

$$\sigma^2 = \frac{1}{m} \sum_i (Z_i - \mu)^2$$

```
# Normalization
```

$$Z_{norm}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

```
#  $\tilde{Z}$  tilde where Gamma and Beta are learnable params
```

$$\tilde{Z}^{(i)} = \gamma Z_{norm}^{(i)} + \beta$$

If the value for these hyperparameters are $\gamma = \sqrt{\sigma^2 + \epsilon}$ and $\beta = \mu$, then $\tilde{Z}^{(i)} = Z^{(i)}$. In the before-mentioned case, the algorithm would essentially compute the identity function. What it does really is to adequate the Z values so that they correspond to the ones calculated for the inputs W .

Batch norm fits in a NN inside of a neuron, before the calculation of the activations and after the computation of the Z values. The hyperparameters that it needs to receive are both β and γ to calculate the \tilde{Z} value so this can be plugged in to the activation function.

This concept also works for mini-batches and the implementation is very similar to the one developed in the previous section, the only difference is that the algorithm now will contain an extra step. Because batch norm zeroes out the mean of the b values in the layers, there is no point on taking it into account, instead, the $\beta^{[L]}$ value will end up affecting the function similar to a bias term.

The procedure to implement gradient descent with the new modifications:

```

for t = 1 \dots num_MiniBatches
  Compute forward propagation on  $X^{\{t\}}$ 
  # In every layer, use Batch Norm
   $Z^{[l]} \Rightarrow \tilde{Z}^{[l]}$ 
  Use back propagation to compute  $dW, d\beta, d\gamma$ 
  Update parameters
# Also works with momentum, RMSprop and Adam

```

Many modern frameworks already have their own implementation of these algorithms but it's very important to understand the reasoning behind them. Batch norm works due to the same principle as the one seen on the input features, but the intuition behind the functionality also extends a bit more. As the NN becomes deeper, the weights on each neuron become quite sturdier, making them harder to modify.

The covariate shift mapping states that if there is an $X \Rightarrow Y$ mapping, but suddenly the distribution of X changes, then the learning algorithm must be re-trained (only black cats training).

Batch norm helps to reduce the impact on the latter layers and prevent covariate shift from heavily influencing the last layers that may not need to change drastically in order to adapt to the new data. The ultimate effect that this process has is to increment the learning rate by lowering the impact on the last layers as the initial ones keep on learning.

While in batch norm a mini-batch is processed at a time, in test time it might be reduced to only one example at a time. The equations for batch norm are the following:

$$\mu = \frac{1}{m} \sum_i z^{(i)} \quad (2.18)$$

$$\sigma^2 = \frac{1}{m} \sum_i (z^{(i)} - \mu)^2 \quad (2.19)$$

$$z_{norm}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}} \quad (2.20)$$

$$\tilde{z}^{(i)} = \gamma z_{norm}^{(i)} + \beta \quad (2.21)$$

At test time we need a new way to come up with μ and σ^2 using exponentially weighted averages as an approximation to keep track on the estimate of both values.

Multi-class Classification

Softmax Regression

Generalization of binary classification that lets a NN predict multiple classes instead of only two. The C value is the number of classes, it has to contain the designated flags for the corresponding desired outputs plus a 0 value that acts as an “unrecognized” value classifier.

The output layer has a number of neurons corresponding to the number of classes in C . Each neuron on this layer contains the probability of $P(c|x)$, where c is a single class; the dimensions of \hat{y} will now be (number of classes, 1). For it to work, there must exist a softmax layer on the output layer.

The softmax layer runs through the following process:

$$Z^{[L]} = W^{[L]}a^{[L-1]} + b^{[L]} \quad (2.22)$$

Activation function:

$$t = e^{(Z^{[L]})}$$
$$a^{[L]} = \frac{e^{Z^{[L]}}}{\sum_{i=1}^{\#classes} t_i}, a^{[L]_i} = \frac{t_i}{\sum_{i=1}^{\#classes} t_i}$$

The “odd” thing on the activation function is that the softmax outputs a (n,1) value that differs from the (1,1) value of the other activation functions. The reason for this is due to the normalization process which indicates the probability on all of the possible outcomes.

In softmax classification, you want to implement a particular loss function, that is the following:

$$\mathbb{L}(\hat{y}, y) = - \sum_{i=1}^{\# \text{ of } C} y_i \log \hat{y}_i \quad (2.23)$$

The cost function is the same as the one implemented in the non-softmax examples. The back propagation it’s important to know the expression for the value of dz , that is the following:

$$dz^{[L]} = \hat{y} - y \quad (2.24)$$

Programming frameworks

Deep Learning Frameworks

Good deep learning software frameworks are the industry standard to implement simple NN functions in order to focus on the other sections of the

algorithms. Some good examples are the following:

- Caffe / Caffe2
- CNTK
- DL4J
- Keras
- Lasagne
- mxnet
- PaddlePaddle
- TensorFlow
- Theano
- Torch

Some criterion to choose the correct framework are the following: ease of programming, running speed and if it is really of an open source nature.

TensorFlow

One of the most famous frameworks, an example of its usage using the cost function $J = w^2 - 10w + 25$ is the following:

```
import numpy as np
import tensorflow as tf

w = tf.Variable(0, dtype=tf.float32)
# cost = tf.add(w**2, tf.add(multiply(-10,w)), 25)
cost = w**2 - 10*w + 25
# Gradient descent algorithm which uses optimizer
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

init = tf.global_variables_initializer()
session = tf.Session()
session.run(init)
# Evaluate w
sess.run(w)
print(session.run(w))
```

```

# Run one step of gradient descent
session.run(train)
print(session.run(w))

# Run through multiple iterations
for i in range(1000):
    session.run(train)
# Value close to 5 (desired value)
print(session.run(w))

```

In order to run the coefficients of the cost function in a dynamic fashion, the following modifications should be done:

```

import numpy as np
import tensorflow as ts

coefficients = np.array([[1.], [-10.], [25.]])

w = tf.Variable(0, dtype=tf.float32)
x = tf.placeholder(tf.float32, [3,1])
cost = x[0][0]*w**2 + x[1][0]*w + x[2][0]
train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

init = tf.global_variables_initializer()
session = tf.Session()
session.run(init)
print(session.run(w))

session.run(train, feed_dict={x: coefficients})
print(session.run(w))

for i in range(1000):
    session.run(train)
print(session.run(w))

```

Chapter 3

Structure of Machine Learning Project

ML Strategy

Quick and effective methods that improve the performance of a Machine Learning project.

Orthogonalization

In deep learning, the process of tuning the hyperparameter values can be one of the most tedious and time consuming steps in the development; the process of consciously tuning parameters to obtain an expected value is called orthogonalization.

Chain assumptions of machine learning:

1. Fit training set well on cost function (bigger NN, perform Adam, ...).
2. Fit dev set well on cost function (regularization, bigger training set).
3. Fit test set well on cost function (bigger dev set).
4. Performs well in real world scenarios (change dev set or cost function).

Setting Up The Goal

Single Number Evaluation Metric

Quick method to evaluate the progress of a NN given a change in one or many of its hyperparameters. The values that need to be attended are the following:

- Precision: Of the examples recognised with a label, what percentage of them are actually the object?
- Recall: What percentage of the actual labeled objects are recognised as such by the algorithm?

With these values, we can obtain a F1 score which can actually help us determine which classifier is actually better; the formula is represented by the harmonic mean which is the following:

$$\frac{2}{\frac{1}{\text{Precision}} + \frac{1}{\text{Recall}}} \quad (3.1)$$

Having a well defined set and a single real number evaluation metric will allow the developers to quickly understand which metric is actually better than the other; it speeds up the iterative process of finding the optimal values.

Another recommendation is that, if training for many labels, that the output of precision and recall is combined into a single real number.

Satisficing and Optimizing Metric

Additional to the accuracy factor playing an important role on the classifier selection process is the running time. Both of the before-mentioned values can also be combined into a single evaluation metric, this can be done with two different approaches:

- With a function that combines both values...

$$\text{cost} = \text{accuracy} - 0.5(\text{runningTime}) \quad (3.2)$$

- With an optimizing-satisficing metric which sets a maximum runtime value (satisficing) which will serve as a reference to pick up the highest accuracy value (optimizing) inside of the range.

The optimizing-satisficing metric can also be used to train the number of accuracy against the number of false positive outputs.

Train/Dev/Test Distributions

It is strongly advised that the data division is done evenly along the multiple distributions in the set. This is done by shuffling a lot the data and dividing it in order to have data from all distributions in the subsets.

Dev and Test Set Size

The “boom” of big data, the distributions of dev and test sets have encountered modifications to the previous 70-30 or 60-20-20 schematics. In today’s world, in big data sets it’s more common to encounter a 98-1-1 (train/dev/test) division; the size of the test set has to be big enough to give high confidence in the overall performance of the system. It is not recommended to “skip” the test set partition but it’s not completely unreasonable for a problem to not have a test set.

When to Change Dev/Test Sets and Metrics

There are cases in which the error percentage is small, but can “leak” data which is unacceptable to show to the end user (e.g. pornographic content). In this case, the second best choice (assuming it does not contain this problem). The error of the algorithms can be written as:

$$\frac{1}{m_{\text{dev}}} \sum_{i=1}^{m_{\text{dev}}} \mathcal{L}\{y_{\text{pred}}^{(i)} \neq y^{(i)}\} \quad (3.3)$$

This basically counts the number of miss-classified examples.

A way to solve the problem of miss-classification can be adding a weight term to the previous function that can filter out unwanted data, such that it acquires a value that makes the example identifiable; the value of w will be 1 if it’s not an unwanted value, and a large number (10, 100, 1000, ...) if it is an undesired output:

$$\frac{1}{m_{\text{dev}}} w^{(i)} \sum_{i=1}^{m_{\text{dev}}} \mathcal{L}\{y_{\text{pred}}^{(i)} \neq y^{(i)}\} \quad (3.4)$$

What this does is that it increments greatly the value for the error so it becomes more evident that the algorithm contains an error.

If the metric and development set indicates that an algorithm should be better but the users needs differ from it, the metric must be re-evaluated to fulfill the established requirements. Another implementation for this is to train another classifier for the undesired value and excluding it from the user’s interaction.

Orthogonalization in ML separates the whole process in two steps:

1. Define an adequate metric for classifiers (place the target).
2. Worry *separately* about how to do well on this metric.

Human-Level Performance

Baseline that gives us an estimate of the error percentage of a human to classify certain objects. It's recommended to use the lowest value possible; if a select group of people can achieve a great level of intuition, then it's very likely that most other humans can, thus, being the optimal value.

Human-Level Performance and ML

After NN reaches "human-level" performance, it can exceed this limit for a bit until it converges into what it's known as the Bayes optimal error (no function that can surpass this implementation).

As long as ML is worse than humans, we can help the system to improve by doing the following:

- Feed it with labeled data
- Manual error analysis (why did a person get something right?)
- Improve on bias and variance

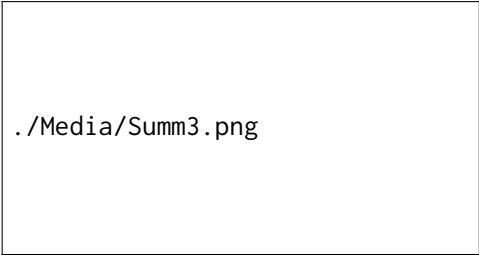
Avoidable Bias

Whatever the human error (or the approximate value of the Bayesian error) is, the decision of focusing on bias or variance correction is changed depending on the scenario; an example is the following:



./Media/Bayesian.png

The difference in percentage of the approximation of the Bayesian error against the training error has been referenced as the Avoidable Bias. The goal of the NN is to perform as close to the Bayesian error as possible, but would be impossible to surpass this without overfitting.



Surpassing Human-Level Performance

After the proxy is passed, the NN will begin to decay in its growth but will keep on learning. If training error becomes smaller than the human performance, it should make sense that humans can no longer help it continue learning. Some common problems where ML performance is significantly better than human-level performance are the following:

- Online adverts
- Product recommendation
- Logistics
- Loan approvals

Also, due to the capability of the computer to process a lot of data nowadays brings a benefit to NN development; where humans shine but computers don't is in natural perception tasks.

Improving Model Performance

The two fundamental assumptions of supervised learning are the following:

1. The training data must fit very well (bias).
2. Training set performance generalizes pretty well to the dev/test set (variance).

Reducing bias and variance:

- Bias: Train bigger model, train longer or better optimization algorithms (RMSProp, Adam, ...), and change NN architecture or parameter search.
- Variance: More data, regularization (L_2 , dropout, ...).
- Training set performance generalizes pretty well to the dev/test set (variance).

Error Analysis

Carrying Out Error Analysis

Manual verification of algorithm's performance that determine what to do next. We need to look at dev examples to get an idea of what is wrong so we can make the necessary adjustments for the classifier to better solve the problems. A recommendation is to get about 100 samples of mislabeled dev set data to analyze them manually and count how many are actually the desired classifier; the percentage of the mislabeled data will lead us into better deciding if it's a good idea to spend time and resources in the adjustment making process.

It is also valid to test many ideas in parallel, using a spreadsheet the ideas should be set as columns and the images as rows and will have a binary value that sets off when an example is recognised as the classifier that's being tested. At the end, we can obtain the error percentage which can be useful to get a sense of the best options to pursue.

Cleaning-up Incorrectly Labeled Data

While mislabeled examples are the ones that the algorithm computes wrong, there are cases in which the human-labeled data contains erroneous labels. As long as the errors are not too far from random, then it's probably okay to leave them as they are, if the dataset is big enough. DL algorithms are quite robust against random errors, but can become vulnerable against systematic errors.

It's recommended to add another column on the error analysis that checks for every time the training example is mislabeled. Three numbers should be looked at to determine if it's a good idea or not to reduce the number of the mislabeled examples:

- Overall dev set error
- Error due to incorrect labels
- Errors due to other causes (overall minus incorrect labels)

If the error due to incorrect labels has a heavy impact on the overall dev set error it's worth while to review the data, in contrast, if it has little impact it's more important to fix the other errors that represent a major impact on the overall error.

Build First System Quickly, Proceed to Iterate

There are many particularities in the NN creation that involve particular cases to improve a specific systems. The recommended procedure is to go from the most general case and then focus on the particular scenarios to make the solution more robust. The simplified steps would be the following:

1. Set up dev/test set and metric.
2. Build initial system quickly.
3. Use Bias/Variance analysis and error analysis to prioritize the next steps.

Mismatched Training and Dev/Test Set

Training and Testing on Different Distributions

In DL era, more and more teams are now training on data that comes from a different distribution than the dev and test sets; of course, there are best practices that should be implemented in order to effectively use the data:

1. Option 1: Merge both datasets uniformly and shuffle the output, and afterwards parsing the data as Train/Dev/Test subsets. The advantage is that now the data would come from the same distribution, but the relevance of some dataset could be overshadowed by the other.
2. Option 2: The data from different distributions should only be allowed in the training partition, while in the train/dev/test subsets, the original distribution values should be allowed.

Bias and Variance with Mismatched Data Distributions

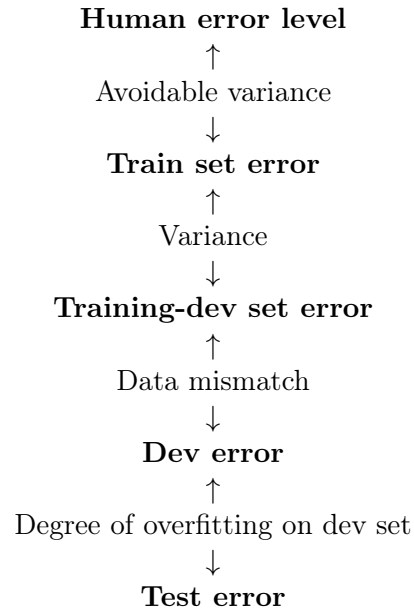
Whenever including different distribution datasets, it's common to encounter that on the dev set, the error is larger than the training error due to the distribution difference of the data in the training set.

To help figure out how much of an impact the different distribution examples have on the error, we have to take into account the Training-Dev set (same dist. as training set, but not used explicitly for training). It is obtained by shuffling all of the train set and selecting a few examples that can be reserved for a training-dev set. To carry out error analysis, what we should do is now look at the error of your classifier on the training set, on the training-dev set, as well as on the dev set.

The outcomes could be the following:

- Training error \ll training-dev error: variance error, not generalizing well on data.
- Training error \approx training-dev error, (training | training-dev) error \neq dev error, means that we have a low variance but with data mismatch problem; somehow the algorithm has learned to work well on another distribution, but not the desired one.
- If the training errors are fairly close together but far away from the desired error, it is considered an avoidable bias error (high bias setting).

The key quantities to look at are:



The numbers of the errors can be represented as a table in order for the developer to have a better visualization of the NN behavior [3.1]. Analyzing

	<i>General Speech recognition</i>	<i>Reverse mirror speech data</i>
<i>Human level</i>	4% (human level)	-
<i>Error on examples trained on</i>	7% (training error)	-
<i>Error on examples not trained on</i>	10% (training-dev error)	6% (dev/test error)

Table 3.1: Example for error representation on a table

the four filled values is considered to be enough for us to modify the values and move towards a good direction.

Addressing Data Mismatch

When encountering mismatch while training data from different distributions, some recommendations would be the following:

- Carry out manual error analysis to try to understand the difference between the training and dev/test sets.
- Make training data more similar, or collect more data similar to dev/test sets.

Artificial data synthesis is the process of combining multiple data sources into one in order to generate a training example for a specific scenario with the data we have (a clear audio + noisy background). If the samples are very unbalanced, a risk that the system has is that it may overfit from the smaller sized sample.

Learning from Multiple Tasks

Transfer Learning

It is the ability of a NN to take knowledge from one task and apply it to a separate task. This is done by removing the output layer (and weights) and adding a new one with another classifier as the output. The dataset also has to be swapped and the last layer has to have initialized its values $w^{[L]}, b^{[L]}$ to then train the model with the new dataset.

There are two steps on the transfer learning process, these are:

1. Pre-training: First run after dataset and output layer are modified.
2. Fine-tuning: Process of tuning the parameters to better fit the new problem.

This allows the new NN to use previously learned data, which will save up some time in similar problems. This is only applicable when the basis of the algorithm is another trained network with a lot of trained examples (bigger than the new one).

Transfer learning make sense when:

- Task A and B have the same input x .
- A lot more data for task A than task B.
- Low level features from A are helpful for B.

Multi Task Learning

The idea of having a NN do several things at the same time, with the hope that these task will help all of the other tasks.

There are problems that require multiple tag processing at the same time (self-driving cars); these will have to compute the $P(\hat{y}_j)$ for each label and the loss function:

$$\frac{1}{m} \sum_{i=1}^m \sum_{j=1}^{\# \text{ of labels}} \mathcal{L}(\hat{y}_j^{(i)}, y_j^{(i)}) \quad (3.5)$$

What this does is that it gives a single example the possibility to have multiple layers in the same object, in contrast, a softmax implementation would only determine the most likely tag of a given example. A thing to notice is that the second sum only will sum up those values with 0 or 1, if the Y vector contains an in-between value, it will not add it up.

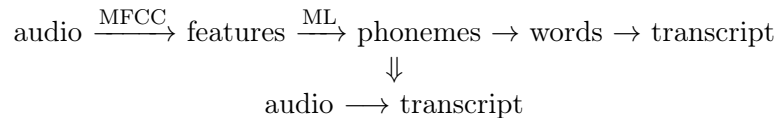
It makes sense in the following scenarios:

- Training on a set of tasks that could benefit from having shared lower-level features.
- Usually, when the amount of data for each task is quite similar.
- When it's possible to train a big enough NN to do well on all tasks.

End-to-end Deep Learning

There are many processing systems that require multiple stages of processing, end-to-end DL can take these and replace them with a single NN.

Traditionally, there are examples such as audio recognition that benefit from this process, for instance, here is the modification of the steps:



The biggest challenge that the DL algorithms faced is that it only works better than the traditional methods whenever a lot of data is available.

In the other hand, researches have found that there are problems which have to be broken down in order to get better results, such as the facial recognition in an access point (capture -> select and crop face -> identify).

The decision of whether you should use an end-to-end approach or a step processing depends on the amount of data formatted in the necessary fashion and the amount of data that needs to be processed at the same time.

End-to-end Learning Pros and Cons

Pros:

- It lets the data speak rather than having it constrained to human pre-conceptions.
- Less hand-designing of components needed.

Cons:

- May need large amount of data.
- Excludes potentially useful hand-designed components.

Chapter 4

Convolutional Neural Networks

Convolutional Neural Networks

Computer Vision

One of the fields where development is advancing really fast in the modern era due to its multiple applications; the main applications are image classification, object detection and neural style transfer. The problem with large images is that the weights between the input vector and the first hidden layer can become insufficient in terms of computational resources; for computer vision it's intended to use these larger images, and it is only possible through convolution.

Edge Detection

For a NN that detects a face, the layers would go from a simple to a more complex task (could be edges -> parts of the face -> faces as a whole).

The edge detection algorithm works by constructing a 3 by 3 matrix that will work as a filter or kernel and applying the "convolution" function over a given image (denotes mathematically with an asterisk "*" symbol).

For the vertical edge detection the filter matrix is the following:

1	0	-1
1	0	-1
1	0	-1

The filter would then go through every subset of pixels and multiply the value of the filter by the intensity of the color, after it has all the values of the 3 by 3 matrix it adds all the values and stores the final output; it is also important to notice that a (m,n) image would output a (m-2, n-1) vector with the output values. What this does, essentially is to obtain the difference between the central values with 0 and the lateral panes in order to determine the existence of a vertical section in an image.

If the results are negative values, it shows a dark to light transition, whereas positive values indicate a light to dark transition; if the nature of the transition is irrelevant, we can take the absolute values for the matrix which will emulate a light to dark transition.

An adaptation of the previously-seen filter can be in order so we can detect horizontal edges:

1	1	1
0	0	0
-1	-1	-1

There have been alterations to the filter in which the values are incremented in the central pixel row, but they work in a pretty similar fashion. With DL advancements, the numbers corresponding to the modified values can be treated as weights and learned through back propagation. This last method has proven to be very useful to figure out the best way to detect the edges from a custom dataset with a particular distribution.

The convolution is the most important function for the learning process, because the back propagation algorithm learns from it.

Padding

A modification to the convolution algorithm which corrects some of the downsides of the previous method. The first downside is that the image ends up shrinking (can be a problem if NN has a lot of layers) as the filter passes, the second is that the reference pixels are inconsistent in terms of their position (the ones on the corner have less impact than the ones on the center because it has more neighboring pixels).

To solve both of the problems is padding the image with an additional border of 1 pixel; this preserves the image dimensions and makes the original border pixels to have the same neighbors as the ones on the center. Padding can also be done with more pixels, and the decision for the p number of padding pixel has a basis on the concepts of “valid” and “same” convolutions.

- Valid: No padding, following the $(n, n) * (f, f) \rightarrow (n - (f + 1), n - (f + 1))$ convention, where “*” is the convolution operator.
- Same: Padding so the output size is the same as the input size, it should follow the principle $p = \frac{f-1}{2}$.

The value for f is most of the time odd because this allows to have a central pixel as a reference and it is extremely recommended to follow this convention.

Strided Convolutions

The stride is the parameter which dictates the number of steps which the filter takes whenever it calculates all the values in a region. The output dimensions will be dictated by the formula:

$$(\lfloor \frac{n + 2p - f}{s} + 1 \rfloor, \lfloor \frac{n + 2p - f}{s} + 1 \rfloor) \quad (4.1)$$

It's has the floor function in case the division is not an integer value, this means tat if the filter steps outside of the image+padding, it will not take into account those calculations. As a note, in some math textbooks, before doing the convolution, the filter is flipped in the horizontal and vertical axes, using the associativity $(A * B) * C = A * (B * C)$ which can be beneficial in some cases such as signal processing, but for NN it simply doesn't matter.

Convolutions Over Volume

In a three dimensional vector, such as an RGB image, the convolution process should be filtered as a three dimensional filter; the output would result in a $(n-2, n-2)$ image (without the 3D factor) if it's a valid implementation. Similar to the process on a two dimensional filter, the three dimensional will go through every layer sequentially until it finishes the computation for the whole image.

Multiple Filters

In order to use multiple filters over one image, it should be processed by every filter separately and by stacking them together, generating a $(n - (f + 1), n - (f + 1), \# \text{ filters})$ using a valid implementation.

One Layer of a Convolutional Network

Let a NN of dimensions $(6, 6, 3)$ have two different filters with dimensions $(3, 3, 3)$ each. The separate calculations for each filter are done, and the first step to "merge" them together is to add a bias value $b|b \in \mathbb{R}$ and add a non-linearity such as ReLU which outputs an equally-sized matrix; after these operations are completed, the results should be stacked in a n-deep representation.

The calculations mentioned above output one layer of a convolutional NN. Remembering the functions for forward propagation $z^{[L]} = w^{[L]}a^{[L-1]} + b^{[L]}$ and $a^{[L]} = g(z^{[L]})$, we can make a comparison with the image processing [4.1].

If you have 10 $(3, 3, 3)$ filters in one layer of NN, how many params does it have? Considering that every filter has $27+1$ (bias) parameters, it will have 280 in total.



Figure 4.1: Representation of one layer of a convolutional network. Contains tags that compare the section with the forward propagation similar value.

The notation for a convolutional layer is the following:

- $[l]$ = reference to the l-th layer.
- $f^{[l]}$ = filter size.
- $p^{[l]}$ = padding.
- $s^{[l]}$ = stride.
- $(n_H^{[l-1]}, n_W^{[l-1]}, n_C^{[l-1]})$ = input dimensions.
- $(n_H^{[l]}, n_W^{[l]}, n_C^{[l]})$ = output dimensions.
- $\lfloor \frac{n^{[l-1]} + 2p^{[l]} f^{[l]}}{s^{[l]}} + 1 \rfloor$ = expanded output formula for n_H and n_w .
- $(f^{[l]}, f^{[l]}, n_c^{[l-1]})$ = filter dimensions.
- $(n_h^{[l]}, n_w^{[l]}, n_c^{[l]})$ = activations dimensions.
- $(f^{[l]}, f^{[l]}, n_c^{[l-1]}, n_c^{[l]})$ = weight dimensions.
- $n_c^{[l]}$ = bias dimensions.

Deep Convolutional Network Example

In the coursera lecture, Andrew Ng developed a multi-layer convolution example [4.2] on a NN. There are tree types of layers in a convolutional NN:

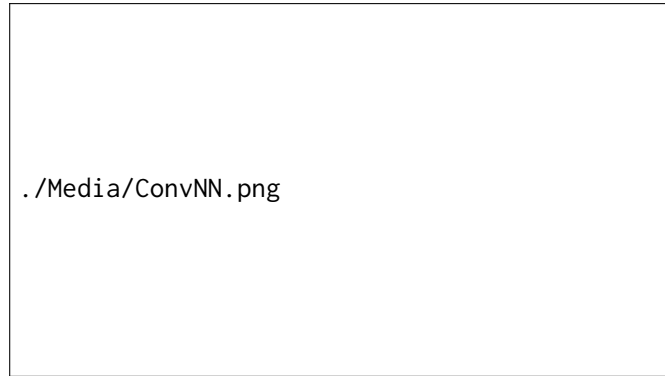


Figure 4.2: Example of a multi-layer convolutional NN.

- Convolution
- Pool
- Fully connected

Pooling Layers

In some cases they can reduce size of representation, speed up the computation, and/or make some of the detected features more robust.

Max pooling: take input and divide it by different regions and then in the output, the region would just be the max value of the contained features (hyperparameters are size of f and s). This can work very well in some cases, and another thing worth noting is that it doesn't change the image on the propagations, it is a static function; if there are many channels, the algorithm will run independently over each channel.

Another less used pooling method is average pooling, which takes the values of the filter, gets the average and outputs it in the corresponding output coordinate; it is mostly used whenever you want to size-down an input image. An extra hyperparameter is the type of pooling that should be used.

Convolutional NN Example

In the Coursera's lecture, Andrew Ng displays a visual representation of a CNN [4.3] to better understand the whole process.

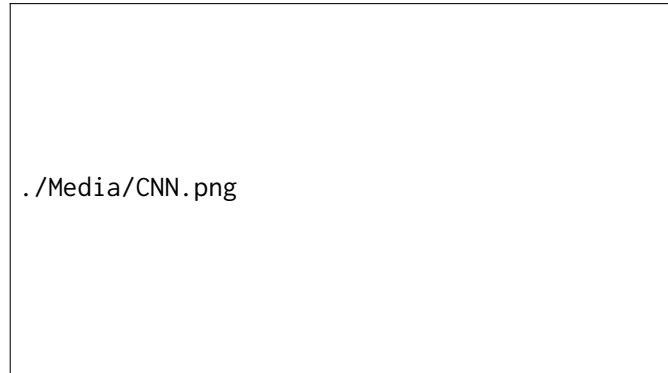


Figure 4.3: Example of a convolutional NN.

Reasoning Behind Convolutions

Two main advantages over convolution layers over fully-connected NN:

- Parameter sharing: A feature detector (such as vertical edge detector) that's useful in one part of the image is probably useful in another part of the image; particularly useful in lower level features, but can also help on higher-level ones.
- Sparsity of connections: Outputs depend directly from the desired inputs, and are not susceptible to the external values from the filter range.

Thanks to these features, these NN can work with smaller parameters due to the nature of its architecture; they are also pretty good on translation and variance in the input examples.

The image is a placeholder for a graphical representation of the LeNet architecture, showing a box with the file path ./Media/LeNet.png.

Figure 4.4: LeNet graphical representation

The image is a placeholder for a graphical representation of the AlexNet architecture, showing a box with the file path ./Media/AlexNet.png.

Figure 4.5: AlexNet graphical representation

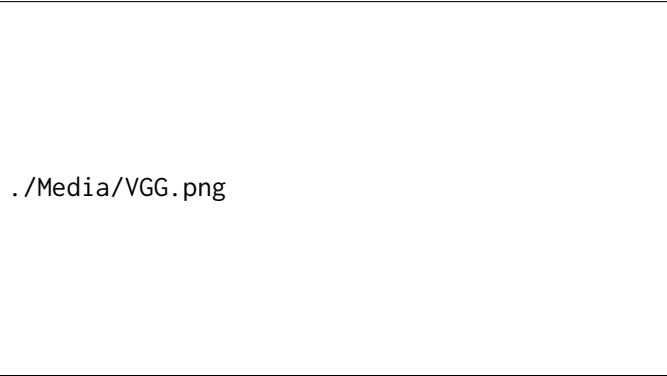
Case Studies

An effective way to learn how to develop convolutional NN is to see and understand some successful examples to gain a better intuition on when to develop such architecture.

Classic Networks

LeNet-5 [4.4], its objective was to recognize hand-written digits from images; trained on gray-scale and with average pool (modern implementations use max pooling).

AlexNet [4.5], developed by Alex Krizhevsky, it is quite similar to LeNet but it's notably bigger; it's more effective than LeNet but has issues on multi-GPU scenarios (not frequently used, but had an immense impact on the DL world and most notably on the computer vision environment).



./Media/VGG.png

Figure 4.6: VGG-16 graphical representation



./Media/ResNet.png

Figure 4.7: ResNet graphical representation

VGG-16 [4.6], gives a general guideline which dictates the size of all of the filter and max-pool dimensions and strides. It's a simpler implementation than the previous ones.

ResNets

When training a very deep NN exploding gradients can be a problem, to avoid these, ResNets can be implemented. ResNets are generated by the stack of multiple residual blocks [4.7], it works basically by making a short-cut in order to pass an activation value to a forward linear calculation. Even if the number of layers grows bigger, the training error would not be affected negatively and would continue to progress normally.



./Media/Inception.png

Figure 4.8: Inception graphical representation

ResNets work well in really deep NN due to the addition of the skipped activation, which will not have a great impact on the learning compared to a “plain” NN; it’s easy to learn the identity function despite the addition of many layers due to the application of the skipped activation.

When going deep on a plain NN, the layers find it difficult to choose parameters to learn even the identity function; the ResNet helps these by adding the skipped activation which makes easier for the layers to learn these identity functions.

Networks in Networks and 1X1 Convolutions

A convolution of a 1X1 filter works in NN with many channels, by doing a convolution it will end up applying a non-linearity which will generate a $(n, m, \text{\#filters})$ output by using a $(1, 1, \text{\#channels})$ filter.

A way to see a 1X1 convolution is that it is basically having a fully connected network that applies for each of the $[\text{\#channels}]$ positions; it is very useful when it is desired to reduce the number of channels in the output.

Inception Network

The reasoning behind an inception layer [4.8] is: instead of choosing what filter size you want on the convolution layer or even whether if you should apply a conv or pool layer, you should do them all at once; both calculations are generated and stacked together.

$$\text{height} \times \text{width} \times f_w \times f_h \times \text{channels} \quad (4.2)$$



Figure 4.9: Inception using a 1X1 filter graphical representation

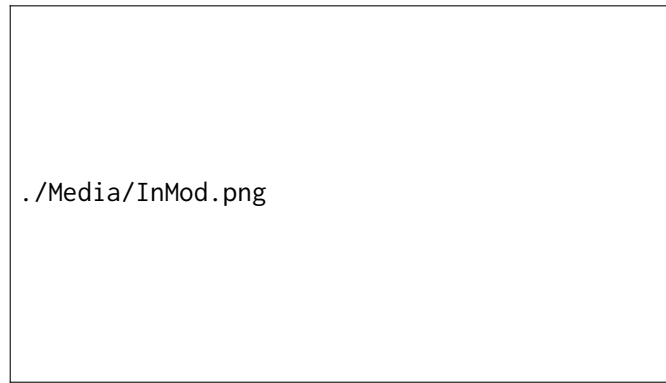


Figure 4.10: Inception module graphical representation

This number can be really large, but implementing 1X1 convolutions can result in improvements of a factor of 10. An alternative architecture would first realize a 1X1 convolution to then apply another custom convolution, the benefit on this operation is that it would work on a smaller object. In the figure [4.9], the middle layer is known as a bottleneck layer because it's the smallest section of this NN. The cost on this implementation will be drastically lower, and the number of multiplications can be calculated through the following expression:

$$(\text{height} \times \text{width} \times \#\text{channels} \times \#\text{filters}) + (\text{height}_{\text{bn}} \times [\dots] \quad (4.3) \\ [\dots] \text{width}_{\text{bn}} \times \#\text{channels}_{\text{bn}} \times \#\text{filters}_{\text{bn}})$$

One thing to remember is that the max-pool layer should always be padded in a “same” fashion so it can be concatenated with the other outputs; an example is shown in figure [4.10]. This figure represents a single module, the network is composed by many of these.

Fun fact: The term “Inception Network” actually came from a meme-related quote from the movie Inception, “We have to go deeper”.

Practical Advises for ConvNet Implementation

Transfer Learning

In Computer Vision application it is faster to download someone else's trained implementation rather than starting from scratch; this works for similar tasks. The recommended process to follow:

1. Find and clone an already-done NN implementation that behaves like the one to be developed with its respective weights.
2. Remove the output and the last softmax or sigmoid function.
3. Create your own output function that outputs the desired values.

The layers before the new softmax or sigmoid function are “frozen” and are recognized by many frameworks with a parameter called “trainableParameter” being equal to 0 and “freeze” being equal to 1.

The notion behind this is that the frozen functions will still be generating the feature outputs adequately, and the final function is the one to be trained for the desired values.

If there is more labeled data to work with, a few layers should be unfrozen (from right to left) to train the deeper layers with the custom data.

Data Augmentation

For the majority of computer vision projects, there is always a feeling that there will never be enough data to properly train the system.

Common data augmentation methods:

- Mirroring: it's quite useful when the process outputs an equivalent result.
- Random cropping: Random sections of the input image, must be big subsets.
- Rotation, shearing, and local warping: Used less but similarly effective.
- Color shifting: Add distortions to the image channels in a small fashion.

Detection Algorithms

Object Localization

Image classification: Determines what an object is through labels.

Classification with localization: determine what and where an object is through labels. While classification refers to a single object recognition problem, detection can identify various objects simultaneously.

To identify the location of an object what we should include are output parameters that enclose the object, these would be something like (b_h, b_y, b_h, b_w) representing what it's known as a bounding box. Another value should be P_c , representing the probability such that an object with a label exists in the picture.

The loss functions for these would be the following:

$$\begin{aligned}\mathcal{L}(\hat{y}, y) &= (\hat{y}_1 - y_1)^2 + (\hat{y}_2 - y_2)^2 + \dots + [\dots] \\ &[\dots](\hat{y}_{\#components} - y_{\#components})^2 | y_1 = 1\end{aligned}\tag{4.4}$$

In the previous description, y_1 is equal to the previously mentioned value P_c which indicates the existence of any possible labeled object.

Landmark Detection

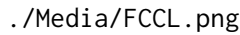
Sometimes it is not necessary to output the bounding box, maybe the output of x and y coordinates are good enough; these act as markers of important points on an image and are called landmarks.

The output would contain these values $l_{1x}, l_{1y}, \dots, l_{nx}, l_{ny}$ generated through a ConvNet. Landmarks are a building block for applications based on facial tracking which is really common in the modern world, it's important to take into account that the first landmark of every training example should represent the same value for it to work.

Sliding Window Algorithm

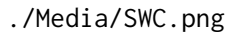
Sliding window detection: Taking a custom size window and going through the image to detect the existence of a labeled object with a ConvNet; after the first pass, a larger window is used and will also run. This process is repeated a couple of times with the hope that if there is an object in the image, the algorithm will detect it.

A huge disadvantage of the process is the computational cost. By cropping and processing multiple times the sections, the program slows down and becomes computationally inefficient.



./Media/FCCL.png

Figure 4.11: FC layer to convolutional layers demonstration



./Media/SWC.png

Figure 4.12: Sliding windows as a convolutional implementation

Convolutional Sliding Window

The first step to implement a convolutional sliding window is to turn the Fully Connected layers into convolutional layers, an example of this process is shown on figure [4.11]. The output volume will be mathematically proven to be equivalent to the corresponding FCL.

The convolutional implementation for sliding windows can be represented by the figure [4.12], this implementation will help to reduce the multiple calculations of the same sectors in an image.

Bounding Box Predictions

A way to get more accurate bounding boxes is through the YOLO (You Only Look Once) algorithm. It places down a grid over an image, then proceeds to run a classification and localization algorithms on each grid cell.

For each grid cell there will be a specific output value containing all the labels necessary; the target output will be a vector of size $(x_{\text{grid}}, y_{\text{grid}}, \text{dim}_{\hat{y}})$. Very efficient and precise algorithm for finding objects in a convolutional fashion, it even works for real-time recognition.

Intersection Over Union

It is computation of the size of an interseccion between objects, in image recognition it is used when comparing the expected output box vs the predicted bounding box. A convention for the IoU threshold has been around 0.5 or greater. The computation for the IoU is the following:

$$\frac{\text{Area of Intersection}}{\text{Area of Union}} \quad (4.5)$$

Non-max Supression

This method is useful for the algorithm to detect an object only once on a grid. What non-max supression does is that it cleans up the multiple detections of an object so at the end only one box remains; it takes the grids with the highest P_C and supresses the ones with low predictive values (using IoU).

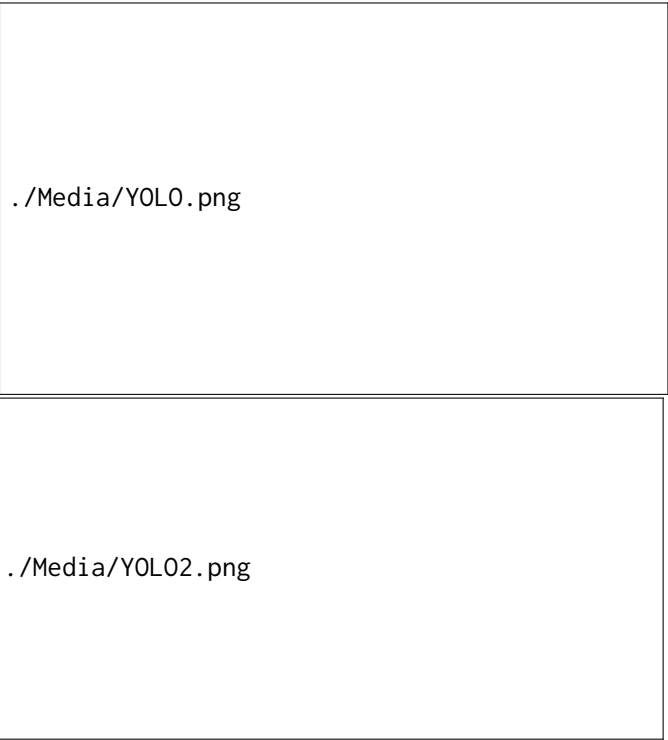
Anchor Objects

Overlapping objects may present an obstacle in image processing, but by pre-defining a box shape (based on multiple grid detections) this can be done easily. The output label will look the same as the previously-seen one but with the same values repeated in ordder to contain two anchor boxes in the same \hat{y} .

Each object in the training image is now assigned to a grid cell which contain the object's midpoint and anchor box for the grid cell with the highest IoU.

YOLO Algorithm

An example for a complete YOLO implementation using two anchor boxes can be found on figure [4.13].



`./Media/YOLO.png`

`./Media/YOLO2.png`

Figure 4.13: YOLO complete implementation

Face Recognition

Verification: Checks if an ID corresponds to an input image.

Recognition: Database of K people, determines input image corresponds to any of the people in the DB.

One Shot Learning

Given one example, the system has to learn to recognize the same object again; to achieve this, a similarity function must be implemented (denote the difference between the images). The comparison is processed throughout the whole dataset to determine if the person is in the database or not.

Siamese Network

This implementation serves as a function to compare an input image with the ones on the database. The network will end up generating an output $f(x^{(n)})$ of an input $x^{(n)}$ and the difference between two examples would look something like:

$$d(x^{(n)}, x^{(n+1)}) = ||f(x^{(n)}) - f(x^{(n+1)})||_2^2 \quad (4.6)$$

Triplet Loss

A way to learn the parameters of the NN so it outputs a good encoding for the pictures is to apply gradient descent on the triplet loss function.

Given multiple images of the same person, it is expected for the difference to be minimal whereas on the opposite scenario it's desired that the difference is big enough. To define the loss function, let's take 3 images A, P, N (anchor or reference, positive and negative) and use the difference formula to define the following:

$$\mathcal{L}(A, P, N) = \max(||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 + \alpha, 0) \implies \quad (4.7)$$

$$\mathcal{J} = \sum_{i=1}^m [||f(A^{(i)}) - f(P^{(i)})||_2^2 - ||f(A^{(i)}) - f(N^{(i)})||_2^2 + \alpha]$$

The second argument for the max function states that the values obtained from the function should be greater or equal to zero.

The examples for A, N, P shouldn't be selected randomly, what it's desired is to choose triplets that are hard to train on, satisfying the following:

$$d(A, P) + \alpha \leq d(A, N) \quad (4.8)$$

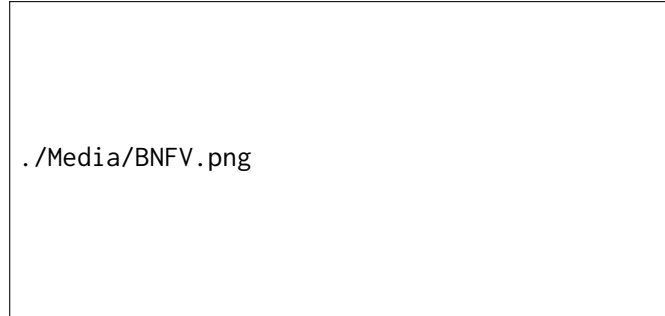


Figure 4.14: Similarity function as a binary classification implementation

Face Verification and Binary Classification

You can treat face verification as a binary function [4.14] by using a feature vector $f(x^{(n)})$ in the database (precomputed) and by comparing it to one obtained by the input.

Neural Style Transfer

Generate new image with a reference style. There are three elements to take into account, a content C , a Style S , and a Generated G image.

Leaning in Deep Layers of ConvNets

The procedure to get to know what the layers are calculated can be done through the following process:

1. Pick a unit in layer 1, find the 9 image patches that maximize the unit's activation.
2. Repeat for other units, looking at these it's easy to understand the nature of the layer and what task it performs.

Cost Function

The cost function for Neural Style Transfer is the following:

$$J(G) = \alpha J_{content}(C, G) + \beta J_{style}(S, G) \quad (4.9)$$

The algorithm process would first initialize the value of G randomly and then use gradient descent to minimize the cost function.

Content Cost Function

The content cost function is the following:

$$J_{context}(C, G) = \frac{1}{2} ||a^{[l][C]} - a^{[l][G]}||^2 \quad (4.10)$$

Where l refers to the layer number and C and G to the respective image. If both activations are similar (low cost), they might be of a similar content.

Style Cost Function

The formula for the style cost is the following:

$$J_{style}(S, G) = \frac{1}{(2n_H^{[l]}n_w^{[l]}n_c^{[l]})} \sum_k \sum_{k'} (G_{kk'}^{[l][S]} - G_{kk'}^{[l][G]})^2 \quad (4.11)$$

1D and 3D Generalizations

Although 2D models are the most common, the idea of convolution can be applied as well in 1-Dimensional and 3-Dimensional examples.

Chapter 5

Sequence Models

Sequence models can be found in many cases on the modern era. It has a wide range impact in fields like speech recognition, natural language processing, music generation, sentiment classification, DNA sequence analysis, machine translation, video activity recognition, name entity recognition, and many others.

Recurrent Neural Networks

Notation

As the most basic example, for each input word (in a NLP scenario) there should be the same number of output words; the output will contain a 1s and 0s mask which will denote if the word on that position is identified.

- The input values are written as $x^{<t>}$ and $y^{<t>}$ as the output, where t is the index of the word (from left to right) of the corresponding value.
- The length of the input sequence is identified as T_x and T_y for the output.
- Different training examples are written as $x^{(i)<t>}$ (t-th element of the i-th training example) and the same will go for y as $y^{(i)<t>}$. As the training examples vary, each one will probably have a different T_x and T_y value.

To represent words in an NLP implementation, the first step is to define a vocabulary. A vocabulary is a list of the used words in a representation, it assigns an index number to each word for ease of use.

When comparing the input sequence, each word is compared to all the words on the vocabulary, after it is found, a one-hot vector is created to reduce the output size. If the word is not recognized, it is labeled as a value with the superscript $< unk >$ for unknown.

Recurrent NN Model

Problems with standard NN:

- Inputs, outputs can be different lengths in different examples.
- Doesn't share features learned across different positions of the text.

What a Recurrent NN [5.1] does is that it inputs the first word into a NN layer and instantly make a prediction, the second and concequent layers

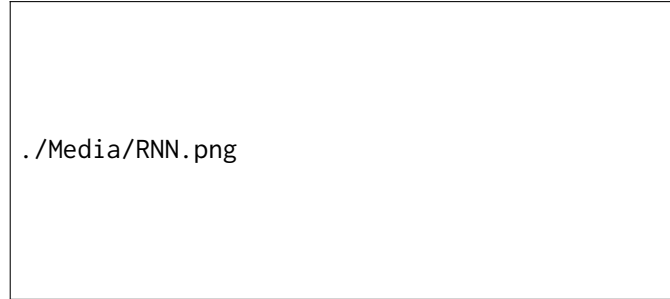


Figure 5.1: Recurrent NN graphical representation

would then receive the corresponding “next word” plus the activation data from the previous layer.

The RNN will go from left to right, having inter-connections between the layers through the parameters W_{ax}, W_{ya}, W_{aa} . One weakness of the model shown on figure [5.1] is that it only takes into account the value that is connected directly before the reference layer, this can become a problem in a NLP algorithm (BRNNs, seen later).

The forward propagation step is made through the following functions:

$$a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a) \quad (5.1)$$

$$y^{<t>} = g_2(W_{ya}a^t + b_y) \quad (5.2)$$

An important thing to notice is that the activation function $g(\theta)$ may vary between the a and y calculations, it is common to see a $\tanh(\theta)$ function as the activation but ReLUs are also used; for the output it will depend directly on the type of problem and the number of output labels needed.

A simpler way to write down the expressions for the functions is the following:

$$a^{<t>} = g_1(W_a[a^{<t-1>}, x^{<t>}] + b_a) \quad (5.3)$$

$$y^{<t>} = g_2(W_y a^t + b_y) \quad (5.4)$$

Back-propagation Through Time

The loss for a sequence NN can be calculated through the following functions:

$$\mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>}) = -y^{<t>} \log \hat{y}^{<t>} - (1 - y^{<t>}) \log(1 - \hat{y}^{<t>}) \quad (5.5)$$

$$\mathcal{L}(\hat{y}, y) = \sum_{t=1}^{T_y} \mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>}) \quad (5.6)$$

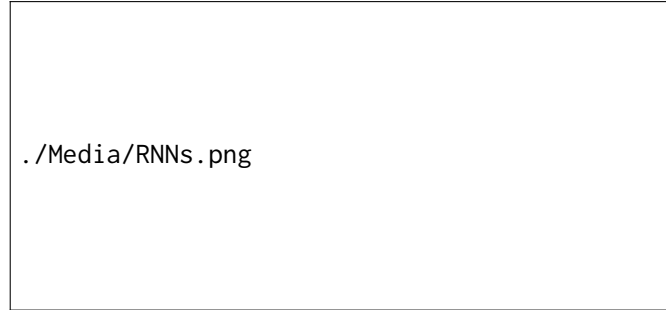


Figure 5.2: RNN types graphical representation

Types of RNNs

Input x and output y can be different types and have completely different sizes, to achieve this, the previously-seen architecture must suffer some changes; the figure [5.2] is a visual representation of the types and the descriptions can be found below:

- *Many-to-many (when input and output length are equal)* is the kind of architecture seen where many T_x values had a related T_y value.
- *Many-to-many (when input and output length are different)* an example can be machine translation, where the translated text might have a different length than the original.
- *Many-to-one* is a scenario where many T_x values have just one output value T_y , an example is Sentiment Classification where a text serves as an input and a prediction of “stars” acts as an output.
- *One-to-many* when the input T_x generates many output nodes T_y , an example can be music generation.
- *One-to-one* when a value T_x has only one T_y related.

Language Modeling and Sequence Generation

Language modelling allows a sequence implementation to take decisions in complex scenarios through probabilities (the apple and pear/pair salad); the processing essentially calculates the probability of an input sentence of being coherent.



Figure 5.3: RNN Language Model graphical representation

Training set: A large corpus of text from a particular language. The words would be then tokenized into one-hot vectors, the end of sentences would also be accounted for with a token called end of sentence $\langle \text{EOS} \rangle$, and the unrecognized words will be tokenized as $\langle \text{UNK} \rangle$.

The RNN model for this case can be found on figure [5.3], where the \hat{y} values are calculated through a softmax function that tries to predict the probability of the word being in the dictionary. The word next to this will do the same probabilistic calculation but given that the previous word is contextually next to it.

The cost function will be based on the probability of the words being associated together. The functions can be written as:

$$\mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>}) = - \sum_i y_i^{<t>} \log \hat{y}_i^{<t>} \quad (5.7)$$

$$\mathcal{L}(\hat{y}, y) = \sum_{t=1}^{T_y} \mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>}) \quad (5.8)$$

Sample Novel Sequences

Given an already-trained Sequence Model, it is the task of generating novel sequences from scratch. The algorithm starts on the first layer where it randomly picks the first word from a batch of probabilistically-limited words, it then passes this selection to the next layer and continue the selection process as the first layer. The sample words can be selected or the $\langle \text{EOS} \rangle$ token can be included. Word based NLP models are more frequent in the modern era, but as the number of resources increases, character based models have become more popular.

Vanishing Gradients

Due to large sequential inputs, vanishing gradients can be an issue due to the weak dependencies that a later layer would have to the earliest layers. Although vanishing gradients are more common, exploding gradients remain as an issue; to counteract these gradient clipping can be a feasible solution, this rescales the vectors on the exploding gradients.

To solve vanishing gradients, more complex methods should be implemented, which will be explained in the following subsections.

Gated Recurrent Unit

Modification of the RNN layer that makes it better to capture long range connections and helps with vanishing gradients.

GRUs work through memory cells c which will provide a space to remember important information from the sequence; it's worth noting that for a value of $c^{<t>}$ there will be an output $a^{<t>}$ that in traditional scenarios are of equal value. At every time step, the c value will be considered to change to a new value, this calculation is performed through the following formulas:

$$\tilde{c}^{<t>} = \tanh(W_c[c^{<t-1>}, x^{<t>}] + b_c) \quad (5.9)$$

$$\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u) \quad (5.10)$$

$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u) * c^{<t-1>} \quad (5.11)$$

The important value on this calculation is the gate Γ , while \tilde{c} controls whether to change the value or not, Γ_u will state when to change the value. The previous formulas correspond to a simpler GRU unit, and to get a full one a value Γ_r corresponding to the relevance is added. The process would now be the following:

$$\tilde{c}^{<t>} = \tanh(W_c[\Gamma_r * c^{<t-1>}, x^{<t>}] + b_c) \quad (5.12)$$

$$\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u) \quad (5.13)$$

$$\Gamma_r = \sigma(W_r[c^{<t-1>}, x^{<t>}] + b_r) \quad (5.14)$$

$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u) * c^{<t-1>} \quad (5.15)$$

The GRU has become the standard in the modern era although there are many other methods such as LSTM which will be seen in the following subsection.

./Media/BRNN.png

Figure 5.4: Bidirectional RNN graphical representation

Long Short Term Memory

More powerful than the GRU and more general, the equations are the following:

$$\tilde{c}^{<t>} = \tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c) \quad (5.16)$$

$$\Gamma_u = \sigma(W_u[a^{<t-1>}, x^{<t>}] + b_u) \quad (5.17)$$

$$\Gamma_f = \sigma(W_f[a^{<t-1>}, x^{<t>}] + b_f) \quad (5.18)$$

$$\Gamma_o = \sigma(W_o[a^{<t-1>}, x^{<t>}] + b_o) \quad (5.19)$$

$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>} \quad (5.20)$$

$$a^{<t>} = \Gamma_o * \tanh(c^{<t>}) \quad (5.21)$$

The Γ_f value represents a gate corresponding to the forget factor, Γ_o corresponds to the output. The separation between the update and forget gate allow the process to be able to keep a previous value while updating on the layer.

Bidirectional RNN

Fixes the forward-only problem with a (as the name suggests) two-way relation between the components. The example diagram is represented on figure [5.4], the addition of backward connections (green) allow this process to compute the backward computations of the activation which will be accounted for the \hat{y} value.

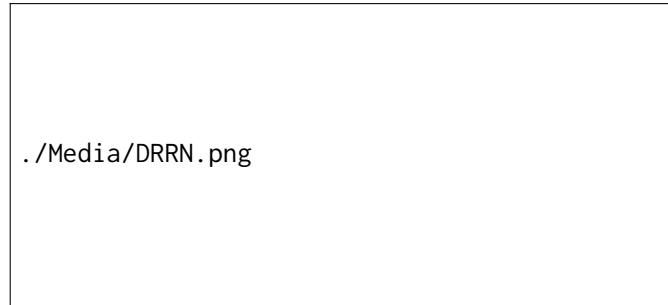


Figure 5.5: Deep RNN graphical representation

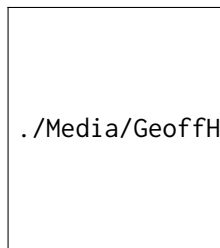
Deep RNNs

The example shown on the course for a Deep RNN is found on figure [5.5]. The indexes for the activations are read as $a^{[l]<t>}$, where l states the layer and t the time.

Chapter 6

Heroes of Deep Learning

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 NNs.

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 rather 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 NNs where only few nodes could be seen and its purpose would be to discover the hidden nodes by 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 NN 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 NNs 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



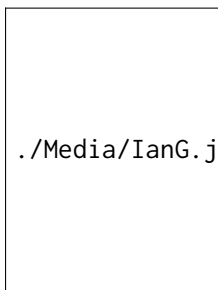
He is a Belgian computer scientist, most of his recent work is related to deep reinforcement learning. Before working on deep reinforcement, he collaborated with Andrew Ng to work on reinforcement learning for autonomous helicopter flight. For all the implementations on this field, a lot of time was 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 raises. 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, Chainer, 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 Learning 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.