

Python Deep Learning - Guía rápida

Python Deep Learning - Introducción

El aprendizaje estructurado profundo o el aprendizaje jerárquico o el aprendizaje profundo, en resumen, es parte de la familia de métodos de aprendizaje automático que son en sí mismos un subconjunto del campo más amplio de Inteligencia Artificial.

El aprendizaje profundo es una clase de algoritmos de aprendizaje automático que utilizan varias capas de unidades de procesamiento no lineales para la extracción y transformación de características. Cada capa sucesiva utiliza la salida de la capa anterior como entrada.

Las redes neuronales profundas, las redes de creencias profundas y las redes neuronales recurrentes se han aplicado a campos como la visión por computadora, el reconocimiento del habla, el procesamiento del lenguaje natural, el reconocimiento de audio, el filtrado de redes sociales, la traducción automática y la bioinformática donde produjeron resultados comparables y en algunos casos mejor que los expertos humanos.

Algoritmos y redes de aprendizaje profundo -

- se basan en el aprendizaje no supervisado de múltiples niveles de características o representaciones de los datos. Las características de nivel superior se derivan de las características de nivel inferior para formar una representación jerárquica.
- usa alguna forma de descenso en gradiente para entrenar.

Python Deep Learning - Medio ambiente

En este capítulo, aprenderemos sobre el entorno configurado para Python Deep Learning. Tenemos que instalar el siguiente software para hacer algoritmos de aprendizaje profundo.

- Python 2.7+
- Scipy con Numpy
- Matplotlib
- Theano
- Keras
- TensorFlow

Se recomienda encarecidamente que Python, NumPy, SciPy y Matplotlib se instalen a través de la distribución Anaconda. Viene con todos esos paquetes.

Necesitamos asegurarnos de que los diferentes tipos de software estén instalados correctamente.

Vayamos a nuestro programa de línea de comandos y escriba el siguiente comando:

```
$ python  
Python 3.6.3 |Anaconda custom (32-bit)| (default, Oct 13 2017, 14:21:34)  
[GCC 7.2.0] on linux
```

A continuación, podemos importar las bibliotecas necesarias e imprimir sus versiones.

```
import numpy  
print numpy.__version__
```

Salida

```
1.14.2
```

Instalación de Theano, TensorFlow y Keras

Antes de comenzar con la instalación de los paquetes: Theano, TensorFlow y Keras, debemos confirmar si el **pip** está instalado. El sistema de gestión de paquetes en Anaconda se llama pip.

Para confirmar la instalación de pip, escriba lo siguiente en la línea de comando:

```
$ pip
```

Una vez que se confirma la instalación de pip, podemos instalar TensorFlow y Keras ejecutando el siguiente comando:

```
$pip install theano  
$pip install tensorflow  
$pip install keras
```

Confirme la instalación de Theano ejecutando la siguiente línea de código:

```
$python -c “import theano: print (theano.__version__)”
```

Salida

```
1.0.1
```

Confirme la instalación de Tensorflow ejecutando la siguiente línea de código:

```
$python -c “import tensorflow: print tensorflow.__version__”
```

Salida

```
1.7.0
```

Confirme la instalación de Keras ejecutando la siguiente línea de código:

```
$python -c "import keras: print keras.__version__"
Using TensorFlow backend
```

Salida

2.1.5

Aprendizaje automático básico de Python Deep

Artificial Intelligence (AI) is any code, algorithm or technique that enables a computer to mimic human cognitive behaviour or intelligence. Machine Learning (ML) is a subset of AI that uses statistical methods to enable machines to learn and improve with experience. Deep Learning is a subset of Machine Learning, which makes the computation of multi-layer neural networks feasible. Machine Learning is seen as shallow learning while Deep Learning is seen as hierarchical learning with abstraction.

Machine learning deals with a wide range of concepts. The concepts are listed below –

- supervised
- unsupervised
- reinforcement learning
- linear regression
- cost functions
- overfitting
- under-fitting
- hyper-parameter, etc.

In supervised learning, we learn to predict values from labelled data. One ML technique that helps here is classification, where target values are discrete values; for example, cats and dogs. Another technique in machine learning that could come of help is regression. Regression works on the target values. The target values are continuous values; for example, the stock market data can be analysed using Regression.

In unsupervised learning, we make inferences from the input data that is not labelled or structured. If we have a million medical records and we have to make sense of it, find the underlying structure, outliers or detect anomalies, we use clustering technique to divide data into broad clusters.

Data sets are divided into training sets, testing sets, validation sets and so on.

A breakthrough in 2012 brought the concept of Deep Learning into prominence. An algorithm classified 1 million images into 1000 categories successfully using 2 GPUs and latest technologies like Big Data.

Relating Deep Learning and Traditional Machine Learning

One of the major challenges encountered in traditional machine learning models is a process called feature extraction. The programmer needs to be specific and tell the computer the features to be looked out for. These features will help in making decisions.

Entering raw data into the algorithm rarely works, so feature extraction is a critical part of the traditional machine learning workflow.

This places a huge responsibility on the programmer, and the algorithm's efficiency relies heavily on how inventive the programmer is. For complex problems such as object recognition or handwriting recognition, this is a huge issue.

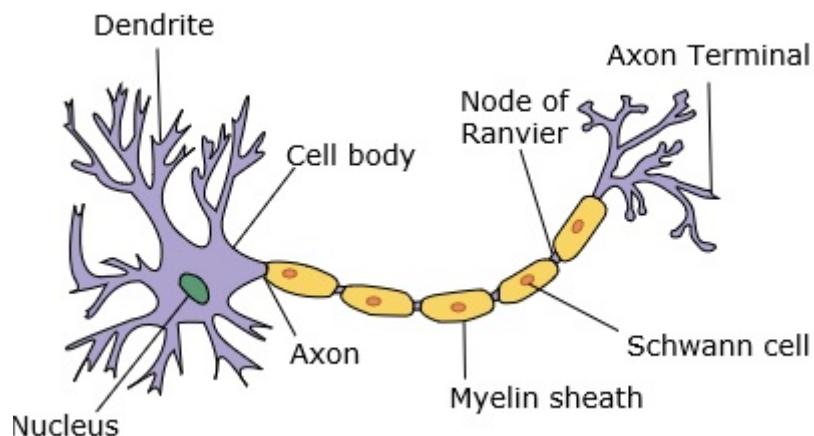
Deep learning, with the ability to learn multiple layers of representation, is one of the few methods that has helped us with automatic feature extraction. The lower layers can be assumed to be performing automatic feature extraction, requiring little or no guidance from the programmer.

Artificial Neural Networks

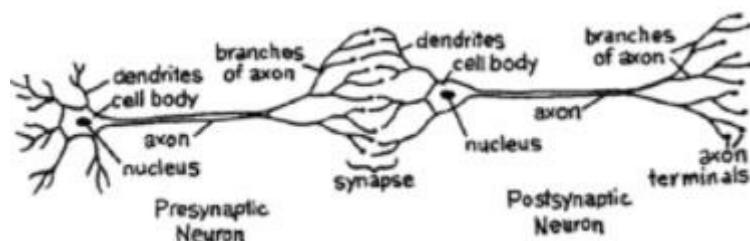
The Artificial Neural Network, or just neural network for short, is not a new idea. It has been around for about 80 years.

It was not until 2011, when Deep Neural Networks became popular with the use of new techniques, huge dataset availability, and powerful computers.

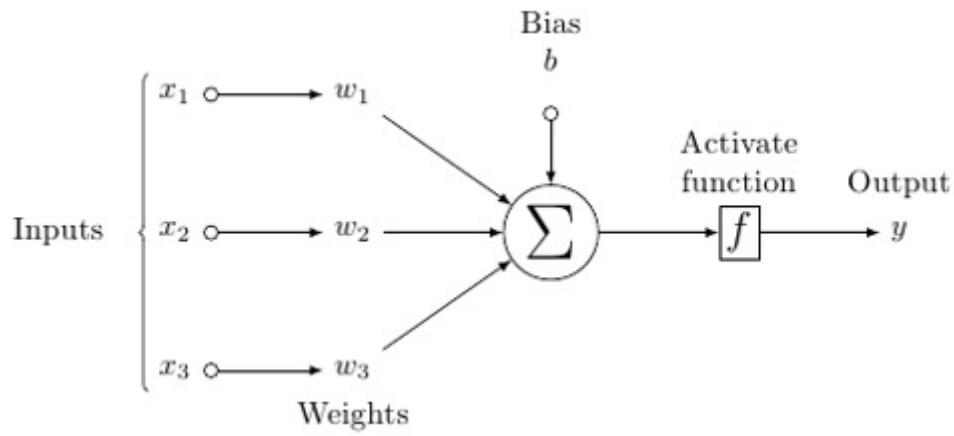
A neural network mimics a neuron, which has dendrites, a nucleus, axon, and terminal axon.



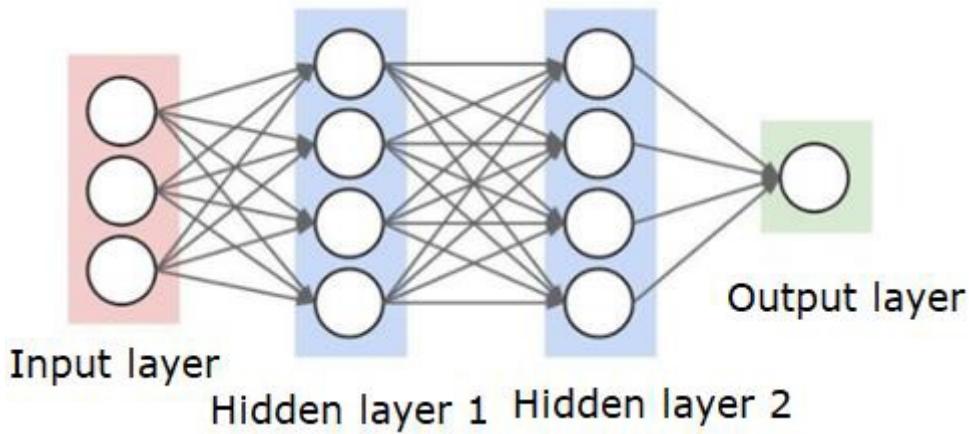
For a network, we need two neurons. These neurons transfer information via synapse between the dendrites of one and the terminal axon of another.



A probable model of an artificial neuron looks like this –



A neural network will look like as shown below –



The circles are neurons or nodes, with their functions on the data and the lines/edges connecting them are the weights/information being passed along.

Each column is a layer. The first layer of your data is the input layer. Then, all the layers between the input layer and the output layer are the hidden layers.

If you have one or a few hidden layers, then you have a shallow neural network. If you have many hidden layers, then you have a deep neural network.

In this model, you have input data, you weight it, and pass it through the function in the neuron that is called threshold function or activation function.

Basically, it is the sum of all of the values after comparing it with a certain value. If you fire a signal, then the result is (1) out, or nothing is fired out, then (0). That is then weighted and passed along to the next neuron, and the same sort of function is run.

We can have a sigmoid (s-shape) function as the activation function.

As for the weights, they are just random to start, and they are unique per input into the node/neuron.

In a typical "feed forward", the most basic type of neural network, you have your information pass straight through the network you created, and you compare the output to what you hoped the output would have been using your sample data.

From here, you need to adjust the weights to help you get your output to match your desired output.

The act of sending data straight through a neural network is called a **feed forward neural network**.

Our data goes from input, to the layers, in order, then to the output.

When we go backwards and begin adjusting weights to minimize loss/cost, this is called **back propagation**.

This is an **optimization problem**. With the neural network, in real practice, we have to deal with hundreds of thousands of variables, or millions, or more.

The first solution was to use stochastic gradient descent as optimization method. Now, there are options like AdaGrad, Adam Optimizer and so on. Either way, this is a massive computational operation. That is why Neural Networks were mostly left on the shelf for over half a century. It was only very recently that we even had the power and architecture in our machines to even consider doing these operations, and the properly sized datasets to match.

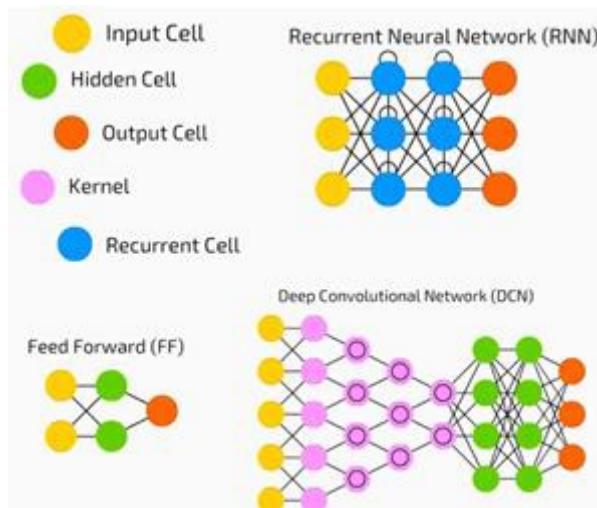
For simple classification tasks, the neural network is relatively close in performance to other simple algorithms like K Nearest Neighbors. The real utility of neural networks is realized when we have much larger data, and much more complex questions, both of which outperform other machine learning models.

Deep Neural Networks

A deep neural network (DNN) is an ANN with multiple hidden layers between the input and output layers. Similar to shallow ANNs, DNNs can model complex non-linear relationships.

The main purpose of a neural network is to receive a set of inputs, perform progressively complex calculations on them, and give output to solve real world problems like classification. We restrict ourselves to feed forward neural networks.

We have an input, an output, and a flow of sequential data in a deep network.



Neural networks are widely used in supervised learning and reinforcement learning problems. These networks are based on a set of layers connected to each other.

In deep learning, the number of hidden layers, mostly non-linear, can be large; say about 1000 layers.

DL models produce much better results than normal ML networks.

We mostly use the gradient descent method for optimizing the network and minimising the loss function.

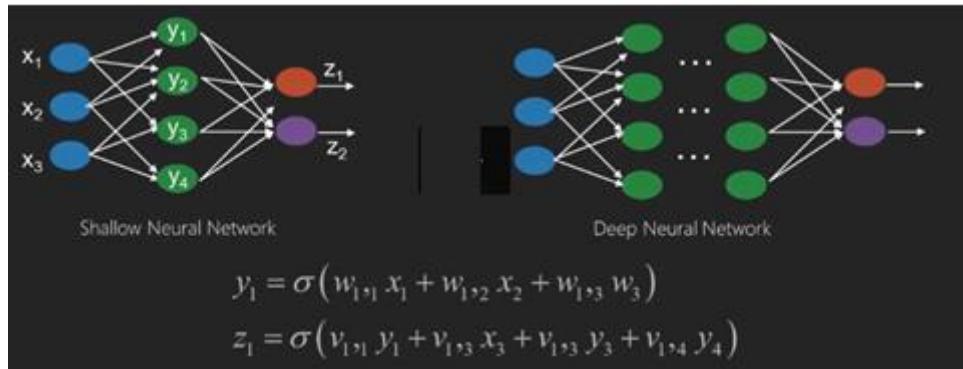
We can use the **Imagenet**, a repository of millions of digital images to classify a dataset into categories like cats and dogs. DL nets are increasingly used for dynamic images apart from static

ones and for time series and text analysis.

Training the data sets forms an important part of Deep Learning models. In addition, Backpropagation is the main algorithm in training DL models.

DL deals with training large neural networks with complex input output transformations.

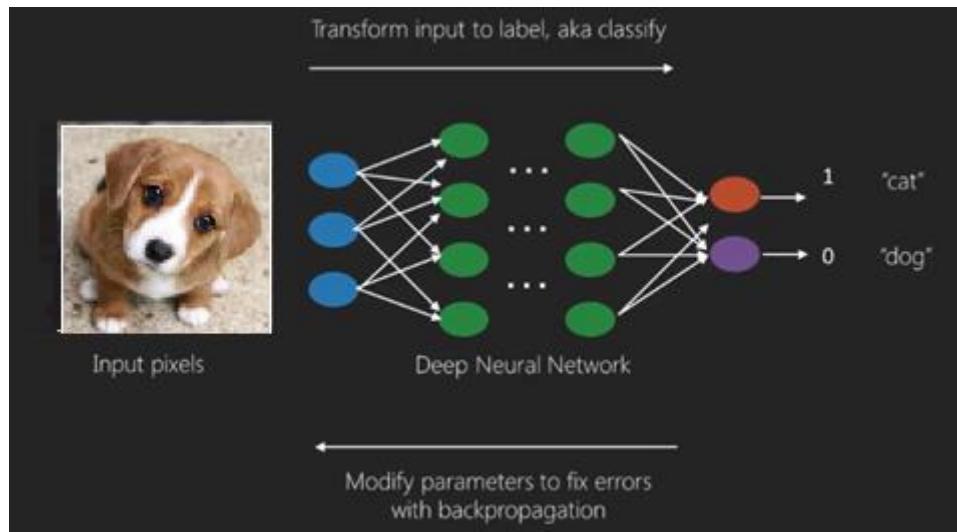
One example of DL is the mapping of a photo to the name of the person(s) in photo as they do on social networks and describing a picture with a phrase is another recent application of DL.



Neural networks are functions that have inputs like $x_1, x_2, x_3 \dots$ that are transformed to outputs like z_1, z_2, z_3 and so on in two (shallow networks) or several intermediate operations also called layers (deep networks).

The weights and biases change from layer to layer. 'w' and 'v' are the weights or synapses of layers of the neural networks.

The best use case of deep learning is the supervised learning problem. Here, we have a large set of data inputs with a desired set of outputs.



Here we apply back propagation algorithm to get correct output prediction.

The most basic data set of deep learning is the MNIST, a dataset of handwritten digits.

We can train a deep Convolutional Neural Network with Keras to classify images of handwritten digits from this dataset.

The firing or activation of a neural net classifier produces a score. For example, to classify patients as sick and healthy, we consider parameters such as height, weight and body temperature, blood pressure etc.

A high score means patient is sick and a low score means he is healthy.

Each node in output and hidden layers has its own classifiers. The input layer takes inputs and passes on its scores to the next hidden layer for further activation and this goes on till the output is reached.

This progress from input to output from left to right in the forward direction is called **forward propagation**.

Credit assignment path (CAP) in a neural network is the series of transformations starting from the input to the output. CAPs elaborate probable causal connections between the input and the output.

CAP depth for a given feed forward neural network or the CAP depth is the number of hidden layers plus one as the output layer is included. For recurrent neural networks, where a signal may propagate through a layer several times, the CAP depth can be potentially limitless.

Deep Nets and Shallow Nets

There is no clear threshold of depth that divides shallow learning from deep learning; but it is mostly agreed that for deep learning which has multiple non-linear layers, CAP must be greater than two.

Basic node in a neural net is a perception mimicking a neuron in a biological neural network. Then we have multi-layered Perception or MLP. Each set of inputs is modified by a set of weights and biases; each edge has a unique weight and each node has a unique bias.

La **precisión** de predicción de una red neuronal depende de sus **pesos y sesgos**.

El proceso de mejorar la precisión de la red neuronal se llama **entrenamiento**. La salida de una red de propulsión directa se compara con el valor que se sabe que es correcto.

La **función de costo o la función de pérdida** es la diferencia entre el producto generado y el producto real.

El objetivo de la capacitación es hacer que el costo de la capacitación sea lo más pequeño posible en millones de ejemplos de capacitación. Para hacer esto, la red ajusta los pesos y los sesgos hasta que la predicción coincide con la salida correcta.

Una vez que se entrena bien, una red neuronal tiene el potencial de hacer una predicción precisa cada vez.

Cuando el patrón se vuelve complejo y desea que su computadora los reconozca, debe buscar redes neuronales. En estos escenarios de patrones complejos, la red neuronal supera a todos los demás algoritmos competidores.

Ahora hay GPU que pueden entrenarlos más rápido que nunca. Las redes neuronales profundas ya están revolucionando el campo de la IA

Las computadoras han demostrado ser buenas para realizar cálculos repetitivos y seguir instrucciones detalladas, pero no han sido tan buenas para reconocer patrones complejos.

Si existe el problema del reconocimiento de patrones simples, una máquina de vectores de soporte (svm) o un clasificador de regresión logística pueden hacer bien el trabajo, pero a medida que aumenta la complejidad del patrón, no hay más remedio que buscar redes neuronales profundas.

Por lo tanto, para patrones complejos como un rostro humano, las redes neuronales superficiales fallan y no tienen otra alternativa que buscar redes neuronales profundas con más capas. Las redes profundas pueden hacer su trabajo al dividir los patrones complejos en otros más simples. Por ejemplo, rostro humano; una red profunda usaría bordes para detectar partes como labios, nariz, ojos, oídos, etc., y luego volvería a combinarlos para formar un rostro humano

La precisión de la predicción correcta se ha vuelto tan precisa que recientemente, en un Google Pattern Recognition Challenge, una red profunda venció a un humano.

Esta idea de una red de perceptrones en capas ha existido por algún tiempo; En esta área, las redes profundas imitan el cerebro humano. Pero una desventaja de esto es que tardan mucho tiempo en entrenarse, una restricción de hardware

Sin embargo, las GPU recientes de alto rendimiento han podido entrenar redes tan profundas en menos de una semana; mientras que los cpus rápidos podrían haber tomado semanas o quizás meses para hacer lo mismo.

Elegir una red profunda

¿Cómo elegir una red profunda? Tenemos que decidir si estamos construyendo un clasificador o si estamos tratando de encontrar patrones en los datos y si vamos a utilizar el aprendizaje no supervisado. Para extraer patrones de un conjunto de datos no etiquetados, utilizamos una máquina de Boltzman restringida o un codificador automático.

Considere los siguientes puntos al elegir una red profunda:

- Para el procesamiento de texto, análisis de sentimientos, análisis y reconocimiento de entidades de nombre, utilizamos una red de tensor neural recurrente neta o recursiva o RNTN;
- Para cualquier modelo de lenguaje que opera a nivel de caracteres, usamos la red recurrente.
- Para el reconocimiento de imágenes, utilizamos la red de creencias profundas DBN o red convolucional.
- Para el reconocimiento de objetos, usamos un RNTN o una red convolucional.
- Para el reconocimiento de voz, utilizamos la red recurrente.

En general, las redes de creencias profundas y los perceptrones multicapa con unidades lineales rectificadas o RELU son buenas opciones para la clasificación.

Para el análisis de series de tiempo, siempre se recomienda utilizar la red recurrente.

Neural nets have been around for more than 50 years; but only now they have risen into prominence. The reason is that they are hard to train; when we try to train them with a method called back propagation, we run into a problem called vanishing or exploding gradients. When that happens, training takes a longer time and accuracy takes a back-seat. When training a data set, we are constantly calculating the cost function, which is the difference between predicted output and the actual output from a set of labelled training data. The cost function is then minimized by adjusting the weights and biases values until the lowest value is obtained. The training process uses a gradient, which is the rate at which the cost will change with respect to change in weight or bias values.

Restricted Boltzman Networks or Autoencoders - RBNs

In 2006, a breakthrough was achieved in tackling the issue of vanishing gradients. Geoff Hinton devised a novel strategy that led to the development of **Restricted Boltzman Machine - RBM**, a shallow two layer net.

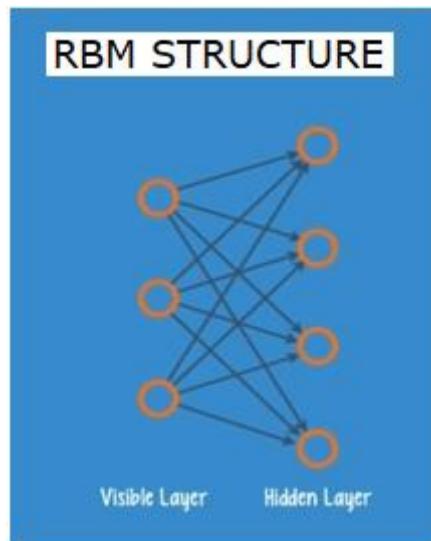
The first layer is the **visible** layer and the second layer is the **hidden** layer. Each node in the visible layer is connected to every node in the hidden layer. The network is known as restricted as no two

layers within the same layer are allowed to share a connection.

Autoencoders are networks that encode input data as vectors. They create a hidden, or compressed, representation of the raw data. The vectors are useful in dimensionality reduction; the vector compresses the raw data into smaller number of essential dimensions. Autoencoders are paired with decoders, which allows the reconstruction of input data based on its hidden representation.

RBM is the mathematical equivalent of a two-way translator. A forward pass takes inputs and translates them into a set of numbers that encodes the inputs. A backward pass meanwhile takes this set of numbers and translates them back into reconstructed inputs. A well-trained net performs back prop with a high degree of accuracy.

In either steps, the weights and the biases have a critical role; they help the RBM in decoding the interrelationships between the inputs and in deciding which inputs are essential in detecting patterns. Through forward and backward passes, the RBM is trained to re-construct the input with different weights and biases until the input and there-construction are as close as possible. An interesting aspect of RBM is that data need not be labelled. This turns out to be very important for real world data sets like photos, videos, voices and sensor data, all of which tend to be unlabelled. Instead of manually labelling data by humans, RBM automatically sorts through data; by properly adjusting the weights and biases, an RBM is able to extract important features and reconstruct the input. RBM is a part of family of feature extractor neural nets, which are designed to recognize inherent patterns in data. These are also called auto-encoders because they have to encode their own structure.



Redes de creencias profundas - DBN

Las redes de creencias profundas (DBN) se forman combinando RBM e introduciendo un método de entrenamiento inteligente. Tenemos un nuevo modelo que finalmente resuelve el problema de la desaparición del gradiente. Geoff Hinton inventó los RBM y también las Redes de creencias profundas como alternativa a la propagación inversa.

Un DBN es similar en estructura a un MLP (perceptrón multicapa), pero muy diferente cuando se trata de entrenamiento. Es la capacitación que permite a los DBN superar a sus homólogos poco profundos.

Un DBN se puede visualizar como una pila de RBM donde la capa oculta de un RBM es la capa visible del RBM por encima de él. El primer RBM está entrenado para reconstruir su entrada con la mayor precisión posible.

La capa oculta del primer RBM se toma como la capa visible del segundo RBM y el segundo RBM se entrena utilizando las salidas del primer RBM. Este proceso se repite hasta que se entrena cada capa de la red.

En un DBN, cada RBM aprende la entrada completa. Un DBN funciona globalmente ajustando la entrada completa en sucesión a medida que el modelo mejora lentamente como una lente de cámara que enfoca lentamente una imagen. Una pila de RBM supera a un solo RBM, ya que un MLP de perceptrón multicapa supera a un solo perceptrón.

En esta etapa, los RBM han detectado patrones inherentes en los datos pero sin ningún nombre o etiqueta. Para finalizar la capacitación de la DBN, tenemos que introducir etiquetas en los patrones y ajustar la red con aprendizaje supervisado.

Necesitamos un conjunto muy pequeño de muestras etiquetadas para que las características y los patrones puedan asociarse con un nombre. Este conjunto de datos con etiqueta pequeña se utiliza para el entrenamiento. Este conjunto de datos etiquetados puede ser muy pequeño en comparación con el conjunto de datos original.

Los pesos y sesgos se alteran ligeramente, lo que resulta en un pequeño cambio en la percepción de la red de los patrones y, a menudo, un pequeño aumento en la precisión total.

La capacitación también se puede completar en un período de tiempo razonable mediante el uso de GPU que brindan resultados muy precisos en comparación con las redes poco profundas y también vemos una solución para el problema del gradiente de fuga.

Redes Adversarias Generativas - GANs

Las redes de confrontación generativas son redes neuronales profundas que comprenden dos redes, enfrentadas una contra la otra, de ahí el nombre de "confrontación".

Las GAN se introdujeron en un artículo publicado por investigadores de la Universidad de Montreal en 2014. El experto en inteligencia artificial de Facebook, Yann LeCun, refiriéndose a las GAN, calificó el entrenamiento de confrontación como "la idea más interesante en los últimos 10 años en ML".

El potencial de GAN es enorme, ya que el escaneo de red aprende a imitar cualquier distribución de datos. Se puede enseñar a las GAN a crear mundos paralelos sorprendentemente similares al nuestro en cualquier dominio: imágenes, música, discurso, prosa. En cierto modo, son artistas de robots, y su producción es bastante impresionante.

En una GAN, una red neuronal, conocida como el generador, genera nuevas instancias de datos, mientras que la otra, el discriminador, las evalúa para verificar su autenticidad.

Digamos que estamos tratando de generar números escritos a mano como los que se encuentran en el conjunto de datos MNIST, que se toma del mundo real. El trabajo del discriminador, cuando se muestra una instancia del verdadero conjunto de datos MNIST, es reconocerlos como auténticos.

Ahora considere los siguientes pasos de la GAN:

- La red del generador toma la entrada en forma de números aleatorios y devuelve una imagen.
- Esta imagen generada se proporciona como entrada a la red discriminadora junto con un flujo de imágenes tomadas del conjunto de datos real.
- El discriminador toma imágenes reales y falsas y devuelve probabilidades, un número entre 0 y 1, donde 1 representa una predicción de autenticidad y 0 representa falso.

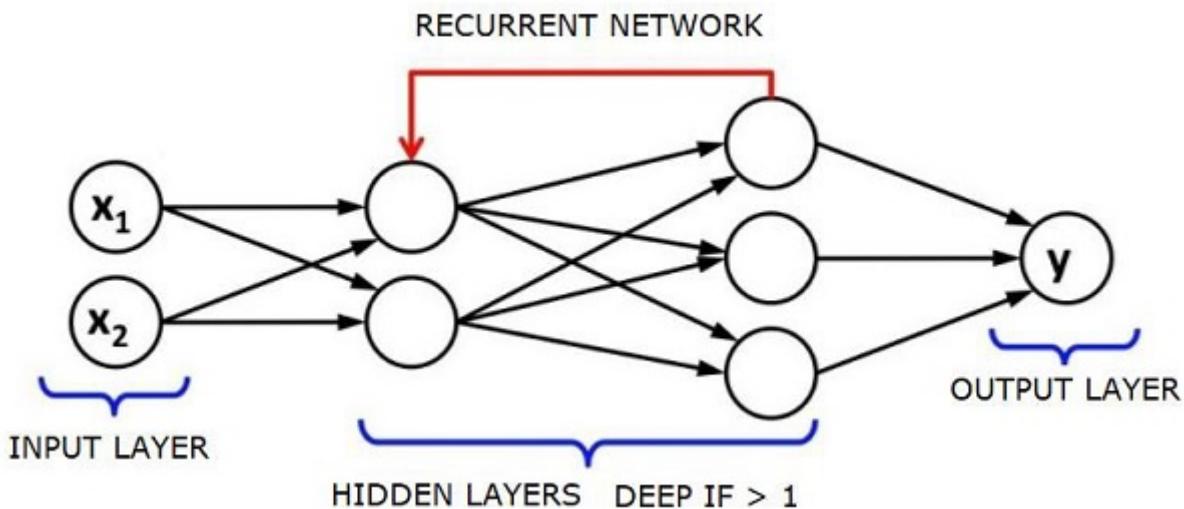
- Entonces tienes un circuito de retroalimentación doble:
 - El discriminador está en un circuito de retroalimentación con la verdad básica de las imágenes, que sabemos.
 - El generador está en un circuito de retroalimentación con el discriminador.

Redes neuronales recurrentes - RNN

RNN Son redes neuronales en las que los datos pueden fluir en cualquier dirección. Estas redes se utilizan para aplicaciones como modelado de lenguaje o procesamiento de lenguaje natural (PNL).

El concepto básico subyacente a los RNN es utilizar información secuencial. En una red neuronal normal se supone que todas las entradas y salidas son independientes entre sí. Si queremos predecir la siguiente palabra en una oración, tenemos que saber qué palabras vinieron antes.

Los RNN se denominan recurrentes ya que repiten la misma tarea para cada elemento de una secuencia, y la salida se basa en los cálculos anteriores. Por lo tanto, se puede decir que los RNN tienen una "memoria" que captura información sobre lo que se ha calculado previamente. En teoría, los RNN pueden usar información en secuencias muy largas, pero en realidad, solo pueden mirar hacia atrás unos pocos pasos.



Las redes de memoria a largo plazo (LSTM) son las RNN más utilizadas.

Junto con las redes neuronales convolucionales, los RNN se han utilizado como parte de un modelo para generar descripciones de imágenes no etiquetadas. Es bastante sorprendente lo bien que parece funcionar.

Redes neuronales profundas convolucionales - CNN

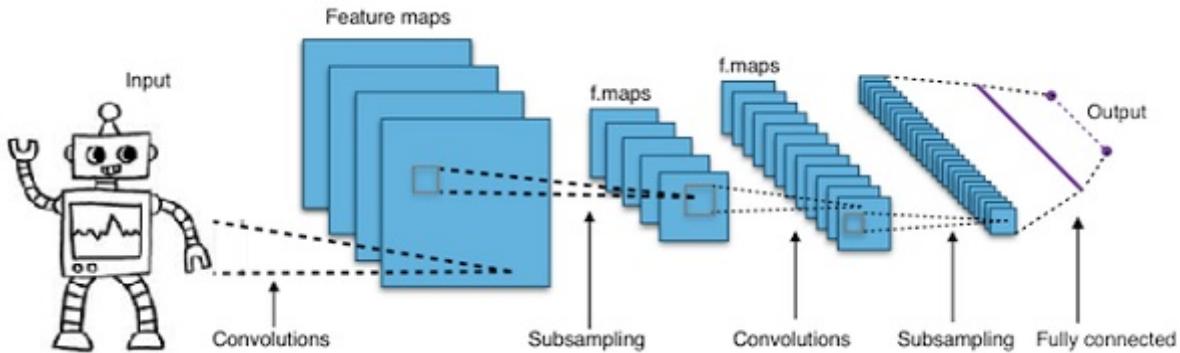
Si aumentamos el número de capas en una red neuronal para profundizarla, aumenta la complejidad de la red y nos permite modelar funciones que son más complicadas. Sin embargo, el número de pesos y sesgos aumentará exponencialmente. De hecho, aprender problemas tan difíciles puede volverse imposible para las redes neuronales normales. Esto lleva a una solución, las redes neuronales convolucionales.

Las CNN se usan ampliamente en visión artificial; se han aplicado también en modelado acústico para reconocimiento automático de voz.

La idea detrás de las redes neuronales convolucionales es la idea de un "filtro en movimiento" que pasa a través de la imagen. Este filtro móvil, o convolución, se aplica a una determinada vecindad de nodos que, por ejemplo, pueden ser píxeles, donde el filtro aplicado es $0.5 \times$ el valor del nodo -

Noted researcher Yann LeCun pioneered convolutional neural networks. Facebook as facial recognition software uses these nets. CNN have been the go to solution for machine vision projects. There are many layers to a convolutional network. In Imagenet challenge, a machine was able to beat a human at object recognition in 2015.

In a nutshell, Convolutional Neural Networks (CNNs) are multi-layer neural networks. The layers are sometimes up to 17 or more and assume the input data to be images.



CNNs drastically reduce the number of parameters that need to be tuned. So, CNNs efficiently handle the high dimensionality of raw images.

Python Deep Learning - Fundamentals

In this chapter, we will look into the fundamentals of Python Deep Learning.

Deep learning models/algorithms

Let us now learn about the different deep learning models/ algorithms.

Some of the popular models within deep learning are as follows –

- Convolutional neural networks
- Recurrent neural networks
- Deep belief networks
- Generative adversarial networks
- Auto-encoders and so on

The inputs and outputs are represented as vectors or tensors. For example, a neural network may have the inputs where individual pixel RGB values in an image are represented as vectors.

The layers of neurons that lie between the input layer and the output layer are called hidden layers. This is where most of the work happens when the neural net tries to solve problems. Taking a closer look at the hidden layers can reveal a lot about the features the network has learned to extract from the data.

Different architectures of neural networks are formed by choosing which neurons to connect to the other neurons in the next layer.

Pseudocode for calculating output

Following is the pseudocode for calculating output of **Forward-propagating Neural Network** –

- # node[] := array of topologically sorted nodes
- # An edge from a to b means a is to the left of b
- # If the Neural Network has R inputs and S outputs,
- # then first R nodes are input nodes and last S nodes are output nodes.
- # incoming[x] := nodes connected to node x
- # weight[x] := weights of incoming edges to x

For each neuron x, from left to right –

- if $x \leq R$: do nothing # its an input node
- $\text{inputs}[x] = [\text{output}[i] \text{ for } i \text{ in } \text{incoming}[x]]$
- $\text{weighted_sum} = \text{dot_product}(\text{weights}[x], \text{inputs}[x])$
- $\text{output}[x] = \text{Activation_function}(\text{weighted_sum})$

Training a Neural Network

We will now learn how to train a neural network. We will also learn back propagation algorithm and backward pass in Python Deep Learning.

We have to find the optimal values of the weights of a neural network to get the desired output. To train a neural network, we use the iterative gradient descent method. We start initially with random initialization of the weights. After random initialization, we make predictions on some subset of the data with forward-propagation process, compute the corresponding cost function C, and update each weight w by an amount proportional to dC/dw , i.e., the derivative of the cost functions w.r.t. the weight. The proportionality constant is known as the learning rate.

The gradients can be calculated efficiently using the back-propagation algorithm. The key observation of backward propagation or backward prop is that because of the chain rule of differentiation, the gradient at each neuron in the neural network can be calculated using the gradient at the neurons, it has outgoing edges to. Hence, we calculate the gradients backwards, i.e., first calculate the gradients of the output layer, then the top-most hidden layer, followed by the preceding hidden layer, and so on, ending at the input layer.

The back-propagation algorithm is implemented mostly using the idea of a computational graph, where each neuron is expanded to many nodes in the computational graph and performs a simple mathematical operation like addition, multiplication. The computational graph does not have any weights on the edges; all weights are assigned to the nodes, so the weights become their own nodes. The backward propagation algorithm is then run on the computational graph. Once the calculation is complete, only the gradients of the weight nodes are required for update. The rest of the gradients can be discarded.

Gradient Descent Optimization Technique

One commonly used optimization function that adjusts weights according to the error they caused is called the “gradient descent.”

Gradiente es otro nombre para pendiente, y la pendiente, en un gráfico xy, representa cómo se relacionan dos variables entre sí: el aumento sobre la carrera, el cambio de distancia sobre el cambio de tiempo, etc. En este caso, la pendiente es la relación entre el error de la red y un solo peso; es decir, cómo cambia el error a medida que varía el peso.

Para decirlo con mayor precisión, queremos encontrar qué peso produce el menor error. Queremos encontrar el peso que representa correctamente las señales contenidas en los datos de entrada y las traduce a una clasificación correcta.

A medida que una red neuronal aprende, ajusta lentamente muchos pesos para que puedan asignar la señal al significado correctamente. La relación entre el error de red y cada uno de esos pesos es una derivada, dE / dw , que calcula el grado en que un ligero cambio en un peso causa un ligero cambio en el error.

Cada peso es solo un factor en una red profunda que involucra muchas transformaciones; la señal del peso pasa a través de activaciones y sumas en varias capas, por lo que utilizamos la regla de cálculo de la cadena para trabajar de nuevo a través de las activaciones y salidas de la red. Esto nos lleva al peso en cuestión y su relación con el error general.

Dadas dos variables, error y peso, están mediadas por una tercera variable, la **activación**, a través de la cual se pasa el peso. Podemos calcular cómo un cambio en el peso afecta un cambio en el error calculando primero cómo un cambio en la activación afecta un cambio en el Error, y cómo un cambio en el peso afecta un cambio en la activación.

La idea básica en el aprendizaje profundo no es más que eso: ajustar los pesos de un modelo en respuesta al error que produce, hasta que ya no pueda reducir el error.

La red profunda se entrena lentamente si el valor del gradiente es pequeño y rápido si el valor es alto. Cualquier imprecisión en el entrenamiento conduce a resultados imprecisos. El proceso de entrenar las redes desde la salida hasta la entrada se llama propagación hacia atrás o apoyo hacia atrás. Sabemos que la propagación hacia adelante comienza con la entrada y funciona hacia adelante. Back prop hace el inverso / opuesto calculando el gradiente de derecha a izquierda.

Cada vez que calculamos un gradiente, utilizamos todos los gradientes anteriores hasta ese punto.

Comencemos en un nodo en la capa de salida. El borde usa el gradiente en ese nodo. A medida que volvemos a las capas ocultas, se vuelve más complejo. El producto de dos números entre 0 y 1 le da un número menor. El valor del gradiente se vuelve cada vez más pequeño y, como resultado, el apoyo de respaldo toma mucho tiempo para entrenar y la precisión se ve afectada.

Desafíos en los algoritmos de aprendizaje profundo

Existen ciertos desafíos para las redes neuronales poco profundas y las redes neuronales profundas, como el sobreajuste y el tiempo de cálculo. Los DNN se ven afectados por el sobreajuste debido al uso de capas adicionales de abstracción que les permiten modelar dependencias raras en los datos de entrenamiento.

Los métodos de **regularización**, como la deserción, la detención temprana, el aumento de datos, el aprendizaje de transferencia se aplican durante el entrenamiento para combatir el sobreajuste. La regularización de abandono omite aleatoriamente unidades de las capas ocultas durante el entrenamiento, lo que ayuda a evitar dependencias raras. Los DNN tienen en cuenta varios parámetros de entrenamiento, como el tamaño, es decir, el número de capas y el número de unidades por capa, la tasa de aprendizaje y los pesos iniciales. Encontrar parámetros óptimos no siempre es práctico debido al alto costo en tiempo y recursos computacionales. Varios hacks, como el procesamiento por lotes, pueden acelerar el cálculo. El gran poder de procesamiento de las GPU ha

ayudado significativamente al proceso de capacitación, ya que la matriz y los cálculos vectoriales requeridos están bien ejecutados en las GPU.

Abandonar

La deserción es una técnica de regularización popular para redes neuronales. Las redes neuronales profundas son particularmente propensas al sobreajuste.

Veamos ahora qué es la deserción y cómo funciona.

En palabras de Geoffrey Hinton, uno de los pioneros de Deep Learning, 'Si tienes una red neuronal profunda y no está sobreajustada, probablemente deberías estar usando una más grande y abandonada'.

La deserción es una técnica en la que durante cada iteración de descenso de gradiente, dejamos caer un conjunto de nodos seleccionados al azar. Esto significa que ignoramos algunos nodos al azar como si no existieran.

Cada neurona se mantiene con una probabilidad de q y se cae al azar con probabilidad $1-q$. El valor q puede ser diferente para cada capa en la red neuronal. Un valor de 0.5 para las capas ocultas y 0 para la capa de entrada funciona bien en una amplia gama de tareas.

Durante la evaluación y predicción, no se utiliza abandono. La salida de cada neurona se multiplica por q para que la entrada a la siguiente capa tenga el mismo valor esperado.

La idea detrás del abandono es la siguiente: en una red neuronal sin regularización del abandono, las neuronas desarrollan una codependencia entre ellas que conduce al sobreajuste.

Truco de implementación

El abandono se implementa en bibliotecas como TensorFlow y Pytorch manteniendo la salida de las neuronas seleccionadas al azar como 0. Es decir, aunque la neurona existe, su salida se sobrescribe como 0.

Parar temprano

Entrenamos redes neuronales usando un algoritmo iterativo llamado descenso de gradiente.

La idea detrás de la detención temprana es intuitiva; Dejamos de entrenar cuando el error comienza a aumentar. Aquí, por error, nos referimos al error medido en los datos de validación, que es la parte de los datos de entrenamiento utilizados para ajustar los hiperparámetros. En este caso, el hiperparámetro es el criterio de detención.

Aumento de datos

El proceso en el que aumentamos la cantidad de datos que tenemos o los aumentamos utilizando los datos existentes y aplicando algunas transformaciones. Las transformaciones exactas utilizadas dependen de la tarea que pretendemos lograr. Además, las transformaciones que ayudan a la red neuronal dependen de su arquitectura.

Por ejemplo, en muchas tareas de visión por computadora, como la clasificación de objetos, una técnica efectiva de aumento de datos está agregando nuevos puntos de datos que son versiones recortadas o traducidas de datos originales.

Cuando una computadora acepta una imagen como entrada, toma una matriz de valores de píxeles. Digamos que toda la imagen se desplaza 15 píxeles hacia la izquierda. Aplicamos muchos cambios diferentes en diferentes direcciones, lo que resulta en un conjunto de datos aumentado muchas veces el tamaño del conjunto de datos original.

Transferir aprendizaje

El proceso de tomar un modelo pre-entrenado y "ajustar" el modelo con nuestro propio conjunto de datos se llama aprendizaje de transferencia. Hay varias formas de hacer esto. Algunas se describen a continuación:

- Entrenamos el modelo pre-entrenado en un gran conjunto de datos. Luego, eliminamos la última capa de la red y la reemplazamos con una nueva capa con pesos aleatorios.
- Luego congelamos los pesos de todas las otras capas y entrenamos la red normalmente. Aquí congelar las capas no está cambiando los pesos durante el descenso o la optimización del gradiente.

El concepto detrás de esto es que el modelo pre-entrenado actuará como un extractor de características, y solo la última capa será entrenada en la tarea actual.

Gráficos computacionales

La retropropagación se implementa en marcos de aprendizaje profundo como Tensorflow, Torch, Theano, etc., mediante el uso de gráficos computacionales. Más significativamente, la comprensión de la propagación hacia atrás en los gráficos computacionales combina varios algoritmos diferentes y sus variaciones, como backprop a través del tiempo y backprop con pesos compartidos. Una vez que todo se convierte en un gráfico computacional, siguen siendo el mismo algoritmo: solo retroceden la propagación en los gráficos computacionales.

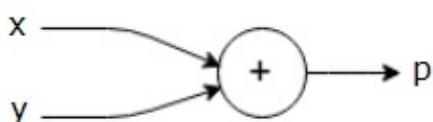
¿Qué es el gráfico computacional?

Un gráfico computacional se define como un gráfico dirigido donde los nodos corresponden a operaciones matemáticas. Los gráficos computacionales son una forma de expresar y evaluar una expresión matemática.

Por ejemplo, aquí hay una ecuación matemática simple:

$$p = x + y$$

Podemos dibujar un gráfico computacional de la ecuación anterior de la siguiente manera.

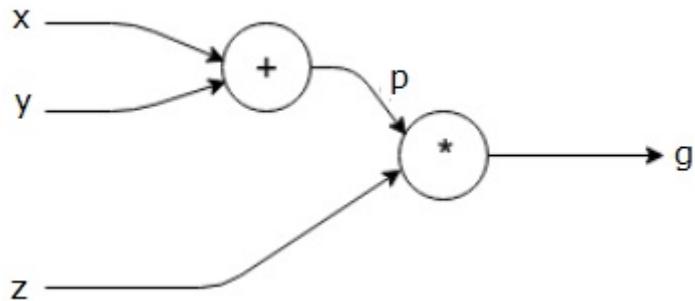


El gráfico computacional anterior tiene un nodo de suma (nodo con signo "+") con dos variables de entrada x e y y una salida p .

Tomemos otro ejemplo, un poco más complejo. Tenemos la siguiente ecuación.

$$sol = (x + y) * z$$

La ecuación anterior está representada por el siguiente gráfico computacional.



Gráficos computacionales y retropropagación

Los gráficos computacionales y la propagación hacia atrás, ambos son conceptos centrales importantes en el aprendizaje profundo para el entrenamiento de redes neuronales.

Pase adelantado

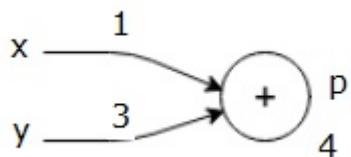
El pase directo es el procedimiento para evaluar el valor de la expresión matemática representada por gráficos computacionales. Hacer pasar adelante significa que estamos pasando el valor de las variables en dirección hacia adelante desde la izquierda (entrada) a la derecha donde está la salida.

Consideremos un ejemplo al dar algún valor a todas las entradas. Supongamos que se dan los siguientes valores a todas las entradas.

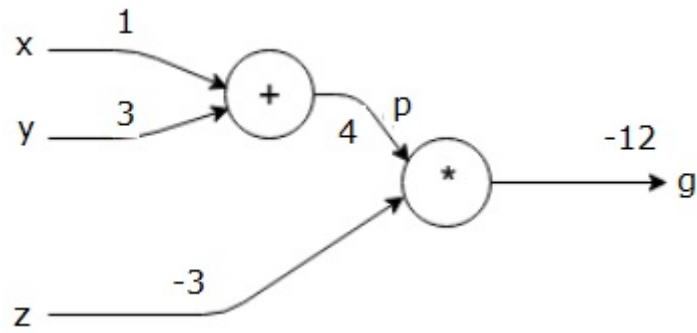
$$x = 1, y = 3, z = -3$$

Al dar estos valores a las entradas, podemos realizar un pase directo y obtener los siguientes valores para las salidas en cada nodo.

Primero, usamos el valor de $x = 1$ e $y = 3$, para obtener $p = 4$.



Luego usamos $p = 4$ y $z = -3$ para obtener $g = -12$. Vamos de izquierda a derecha, hacia adelante.



Objetivos del pase hacia atrás

En el paso hacia atrás, nuestra intención es calcular los gradientes para cada entrada con respecto a la salida final. Estos gradientes son esenciales para entrenar la red neuronal mediante el descenso de gradiente.

Por ejemplo, deseamos los siguientes gradientes.

Gradientes deseados

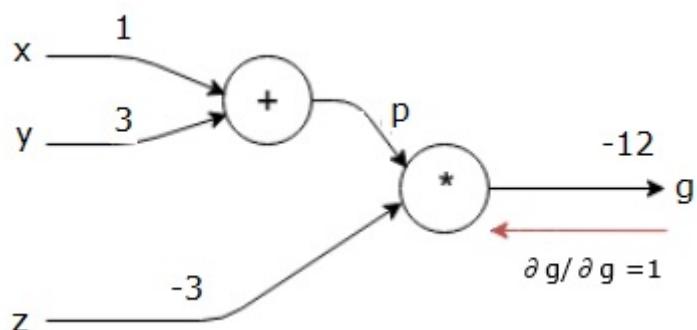
$$\frac{\partial X}{\partial F}, \frac{\partial y}{\partial F}, \frac{\partial z}{\partial F}$$

Pase hacia atrás (propagación hacia atrás)

Comenzamos el paso hacia atrás al encontrar la derivada de la salida final con respecto a la salida final (¡sí mismo!). Por lo tanto, dará como resultado la derivación de identidad y el valor es igual a uno.

$$\frac{\partial sol}{\partial sol} = 1$$

Nuestro gráfico computacional ahora se ve como se muestra a continuación:



A continuación, haremos el paso hacia atrás a través de la operación "=". Calcularemos los gradientes en py z. Como $g = p * z$, sabemos que -

$$\frac{\partial sol}{\partial z} = p$$

$$\frac{\partial sol}{\partial pag} = z$$

Ya conocemos los valores de z y p del paso directo. Por lo tanto, obtenemos -

$$\frac{\partial sol}{\partial z} = p = 4$$

y

$$\frac{\partial sol}{\partial pag} = z = -3$$

Queremos calcular los gradientes en x e y -

$$\frac{\partial sol}{\partial X} \frac{\partial sol}{\partial y}$$

Sin embargo, queremos hacer esto de manera eficiente (aunque x y g están a solo dos saltos en este gráfico, imagínense que están muy lejos el uno del otro). Para calcular estos valores de manera eficiente, utilizaremos la regla de la cadena de diferenciación. De la regla de la cadena, tenemos:

$$\frac{\partial sol}{\partial X} = \frac{\partial sol}{\partial pag} * \frac{\partial pag}{\partial X}$$

$$\frac{\partial sol}{\partial y} = \frac{\partial sol}{\partial pag} * \frac{\partial pag}{\partial y}$$

Pero ya sabemos que dg / dp = -3, dp / dx y dp / dy son fáciles ya que p depende directamente de x e y. Tenemos -

$$p = x + y \Rightarrow \frac{\partial X}{\partial pag} = 1, \frac{\partial y}{\partial pag} = 1$$

Por lo tanto, obtenemos -

$$\frac{\partial sol}{\partial F} = \frac{\partial sol}{\partial pag} * \frac{\partial pag}{\partial X} = (-3) . 1 = -3$$

Además, para la entrada y -

$$\frac{\partial sol}{\partial y} = \frac{\partial sol}{\partial pag} * \frac{\partial pag}{\partial y} = (-3) . 1 = -3$$

La razón principal para hacer esto al revés es que cuando tuvimos que calcular el gradiente en x, solo usamos valores ya calculados y dq / dx (derivada de la salida del nodo con respecto a la entrada del mismo nodo). Utilizamos información local para calcular un valor global.

Pasos para entrenar una red neuronal

Siga estos pasos para entrenar una red neuronal:

- Para el punto de datos x en el conjunto de datos, hacemos pasar con x como entrada, y calculamos el costo c como salida.
- Hacemos un paso hacia atrás comenzando en c, y calculamos gradientes para todos los nodos en el gráfico. Esto incluye nodos que representan los pesos de la red neuronal.
- Luego actualizamos los pesos haciendo $W = W - \text{gradiente de tasa de aprendizaje} *$.
- Repetimos este proceso hasta que se cumplan los criterios de detención.

Python Deep Learning - Aplicaciones

El aprendizaje profundo ha producido buenos resultados para algunas aplicaciones, como visión por computadora, traducción de idiomas, subtítulos de imágenes, transcripción de audio, biología molecular, reconocimiento de voz, procesamiento de lenguaje natural, automóviles autónomos, detección de tumores cerebrales, traducción de voz en tiempo real, música composición, juego automático, etc.

El aprendizaje profundo es el próximo gran salto después del aprendizaje automático con una implementación más avanzada. En la actualidad, se dirige a convertirse en un estándar de la industria que promete cambiar las reglas del juego cuando se trata de datos no estructurados sin procesar.

El aprendizaje profundo es actualmente uno de los mejores proveedores de soluciones para una amplia gama de problemas del mundo real. Los desarrolladores están creando programas de inteligencia artificial que, en lugar de usar reglas dadas previamente, aprenden de ejemplos para resolver tareas complicadas. Con el aprendizaje profundo utilizado por muchos científicos de datos, las redes neuronales más profundas están entregando resultados cada vez más precisos.

La idea es desarrollar redes neuronales profundas aumentando el número de capas de entrenamiento para cada red; la máquina aprende más sobre los datos hasta que sean lo más precisos posible. Los desarrolladores pueden usar técnicas de aprendizaje profundo para implementar tareas complejas de aprendizaje automático y entrenar redes de inteligencia artificial para tener altos niveles de reconocimiento perceptivo.

El aprendizaje profundo encuentra su popularidad en la visión por computadora. Aquí una de las tareas logradas es la clasificación de imágenes donde las imágenes de entrada dadas se clasifican como gato, perro, etc. o como una clase o etiqueta que mejor describa la imagen. Nosotros, como humanos, aprendemos cómo hacer esta tarea muy temprano en nuestras vidas y tenemos estas habilidades para reconocer rápidamente patrones, generalizar a partir de conocimientos previos y adaptarnos a diferentes entornos de imagen.

Bibliotecas y marcos

En este capítulo, relacionaremos el aprendizaje profundo con las diferentes bibliotecas y marcos.

Aprendizaje profundo y Theano

Si queremos comenzar a codificar una red neuronal profunda, es mejor que tengamos una idea de cómo funcionan los diferentes marcos como Theano, TensorFlow, Keras, PyTorch, etc.

Theano es una biblioteca de Python que proporciona un conjunto de funciones para construir redes profundas que entran rápidamente en nuestra máquina.

Theano se desarrolló en la Universidad de Montreal, Canadá, bajo el liderazgo de Yoshua Bengio, un pionero de la red profunda.

Theano nos permite definir y evaluar expresiones matemáticas con vectores y matrices que son matrices rectangulares de números.

Técnicamente hablando, tanto las redes neuronales como los datos de entrada pueden representarse como matrices y todas las operaciones netas estándar pueden redefinirse como operaciones matriciales. Esto es importante ya que las computadoras pueden realizar operaciones matriciales muy rápidamente.

Podemos procesar múltiples valores de matriz en paralelo y si construimos una red neuronal con esta estructura subyacente, podemos usar una sola máquina con una GPU para entrenar redes enormes en una ventana de tiempo razonable.

Sin embargo, si usamos Theano, tenemos que construir la red profunda desde cero. La biblioteca no proporciona una funcionalidad completa para crear un tipo específico de red profunda.

En cambio, tenemos que codificar cada aspecto de la red profunda como el modelo, las capas, la activación, el método de entrenamiento y cualquier método especial para detener el sobreajuste.

Sin embargo, la buena noticia es que Theano permite construir nuestra implementación sobre una parte superior de las funciones vectorizadas, proporcionándonos una solución altamente optimizada.

Hay muchas otras bibliotecas que amplían la funcionalidad de Theano. TensorFlow y Keras se pueden usar con Theano como backend.

Aprendizaje profundo con TensorFlow

Google TensorFlow es una biblioteca de Python. Esta biblioteca es una excelente opción para crear aplicaciones de aprendizaje profundo de grado comercial.

TensorFlow surgió de otra biblioteca DistBelief V2 que era parte de Google Brain Project. Esta biblioteca tiene como objetivo extender la portabilidad del aprendizaje automático para que los modelos de investigación puedan aplicarse a aplicaciones de grado comercial.

Al igual que la biblioteca Theano, TensorFlow se basa en gráficos computacionales en los que un nodo representa datos persistentes u operaciones matemáticas y los bordes representan el flujo de datos entre nodos, que es una matriz o tensor multidimensional; de ahí el nombre de TensorFlow

La salida de una operación o un conjunto de operaciones se alimenta como entrada en la siguiente.

Aunque TensorFlow fue diseñado para redes neuronales, funciona bien para otras redes en las que el cálculo puede modelarse como un gráfico de flujo de datos.

TensorFlow también utiliza varias características de Theano, como la eliminación común y de subexpresión, la diferenciación automática, las variables compartidas y simbólicas.

Se pueden construir diferentes tipos de redes profundas usando TensorFlow como redes convolucionales, Autoencoders, RNTN, RNN, RBM, DBM / MLP, etc.

Sin embargo, no hay soporte para la configuración de hiperparámetros en TensorFlow. Para esta funcionalidad, podemos usar Keras.

Aprendizaje profundo y Keras

Keras es una biblioteca de Python potente y fácil de usar para desarrollar y evaluar modelos de aprendizaje profundo.

Tiene un diseño minimalista que nos permite construir una red capa por capa; entrenarlo y ejecutarlo.

Envuelve las eficientes bibliotecas de computación numérica Theano y TensorFlow y nos permite definir y entrenar modelos de redes neuronales en unas pocas líneas cortas de código.

Es una API de red neuronal de alto nivel, que ayuda a hacer un amplio uso del aprendizaje profundo y la inteligencia artificial. Se ejecuta sobre una serie de bibliotecas de nivel inferior, incluidas TensorFlow, Theano, etc. El código Keras es portátil; podemos implementar una red neuronal en Keras usando Theano o TensorFlow como back-end sin ningún cambio en el código.

Python Deep Learning - Implementaciones

En esta implementación del aprendizaje profundo, nuestro objetivo es predecir el desgaste de los clientes o la rotación de datos para un determinado banco, que es probable que los clientes abandonen este servicio bancario. El conjunto de datos utilizado es relativamente pequeño y contiene 10000 filas con 14 columnas. Estamos utilizando la distribución Anaconda y marcos como Theano, TensorFlow y Keras. Keras está construido sobre Tensorflow y Theano, que funcionan como backends.

```
# Artificial Neural Network
# Installing Theano
pip install --upgrade theano

# Installing Tensorflow
pip install -upgrade tensorflow

# Installing Keras
pip install --upgrade keras
```

Paso 1: preprocessamiento de datos

```
In[]:  
  
# Importing the libraries  
import numpy as np  
import matplotlib.pyplot as plt  
import pandas as pd  
  
# Importing the database  
dataset = pd.read_csv('Churn_Modelling.csv')
```

Paso 2

Creamos matrices de las características del conjunto de datos y la variable objetivo, que es la columna 14, etiquetada como "Salida".

El aspecto inicial de los datos es como se muestra a continuación:

```
In[]:  
X = dataset.iloc[:, 3:13].values  
Y = dataset.iloc[:, 13].values  
X
```

Salida

```
array([[619, 'France', 'Female', ..., 1, 1, 101348.88],  
       [608, 'Spain', 'Female', ..., 0, 1, 112542.58],  
       [502, 'France', 'Female', ..., 1, 0, 113931.57],  
       ...,  
       [709, 'France', 'Female', ..., 0, 1, 42085.58],  
       [772, 'Germany', 'Male', ..., 1, 0, 92888.52],  
       [792, 'France', 'Female', ..., 1, 0, 38190.78]], dtype=object)
```

Paso 3

```
Y
```

Salida

```
array([1, 0, 1, ..., 1, 1, 0], dtype = int64)
```

Etapa 4

Simplificamos el análisis codificando variables de cadena. Estamos utilizando la función ScikitLearn 'LabelEncoder' para codificar automáticamente las diferentes etiquetas en las columnas con valores entre 0 y n_classes-1.

```

from sklearn.preprocessing import LabelEncoder, OneHotEncoder
labelencoder_X_1 = LabelEncoder()
X[:,1] = labelencoder_X_1.fit_transform(X[:,1])
labelencoder_X_2 = LabelEncoder()
X[:, 2] = labelencoder_X_2.fit_transform(X[:, 2])
X

```

Salida

```

array([[619, 0, 0, ..., 1, 1, 101348.88],
       [608, 2, 0, ..., 0, 1, 112542.58],
       [502, 0, 0, ..., 1, 0, 113931.57],
       ...,
       [709, 0, 0, ..., 0, 1, 42085.58],
       [772, 1, 1, ..., 1, 0, 92888.52],
       [792, 0, 0, ..., 1, 0, 38190.78]], dtype=object)

```

En el resultado anterior, los nombres de los países se reemplazan por 0, 1 y 2; mientras que masculino y femenino se reemplazan por 0 y 1.

Paso 5

Etiquetado de datos codificados

Usamos la misma biblioteca **ScikitLearn** y otra función llamada **OneHotEncoder** para simplemente pasar el número de columna creando una variable ficticia.

```

onehotencoder = OneHotEncoder(categorical_features = [1])
X = onehotencoder.fit_transform(X).toarray()
X = X[:, 1:]
X

```

Ahora, las primeras 2 columnas representan el país y la cuarta columna representa el género.

Salida

```

array([[0.0000000e+00, 0.0000000e+00, 6.1900000e+02, ..., 1.0000000e+00,
       1.0000000e+00, 1.0134888e+05],
       [0.0000000e+00, 1.0000000e+00, 6.0800000e+02, ..., 0.0000000e+00,
       1.0000000e+00, 1.1254258e+05],
       [0.0000000e+00, 0.0000000e+00, 5.0200000e+02, ..., 1.0000000e+00,
       0.0000000e+00, 1.1393157e+05],
       ...,
       [0.0000000e+00, 0.0000000e+00, 7.0900000e+02, ..., 0.0000000e+00,
       1.0000000e+00, 4.2085580e+04],
       [1.0000000e+00, 0.0000000e+00, 7.7200000e+02, ..., 1.0000000e+00,
       0.0000000e+00, 9.2888520e+04],
       [0.0000000e+00, 0.0000000e+00, 7.9200000e+02, ..., 1.0000000e+00,
       0.0000000e+00, 3.8190780e+04]])

```

Siempre dividimos nuestros datos en parte de capacitación y prueba; entrenamos nuestro modelo en datos de entrenamiento y luego verificamos la precisión de un modelo en datos de prueba que ayuda a evaluar la eficiencia del modelo.

Paso 6

Estamos utilizando la función `train_test_split` de **ScikitLearn** para dividir nuestros datos en conjunto de entrenamiento y conjunto de prueba. Mantenemos la relación de división entre el tren y la prueba como 80:20.

```
#Splitting the dataset into the Training set and the Test Set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2)
```

Algunas variables tienen valores en miles, mientras que otras tienen valores en decenas o unidades. Escalamos los datos para que sean más representativos.

Paso 7

En este código, estamos ajustando y transformando los datos de entrenamiento usando la función **StandardScaler**. Estandarizamos nuestra escala para que usemos el mismo método ajustado para transformar / escalar datos de prueba.

```
# Feature Scaling
```

```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

Salida

```
array([[-0.5698444 ,  1.74309049,  0.16958176, ...,  0.64259497,
       -1.03227043,  1.10643166],
      [ 1.75486502, -0.57369368, -2.30455945, ...,  0.64259497,
       0.9687384 , -0.74866447],
      [-0.5698444 , -0.57369368, -1.19119591, ...,  0.64259497,
       -1.03227043,  1.48533467],
      ...,
      [-0.5698444 , -0.57369368,  0.9015152 , ...,  0.64259497,
       -1.03227043,  1.41231994],
      [-0.5698444 ,  1.74309049, -0.62420521, ...,  0.64259497,
       0.9687384 ,  0.84432121],
      [ 1.75486502, -0.57369368, -0.28401079, ...,  0.64259497,
       -1.03227043,  0.32472465]])
```

Los datos ahora se escalan correctamente. Finalmente, hemos terminado con nuestro preprocesamiento de datos. Ahora, comenzaremos con nuestro modelo.

Paso 8

Importamos los módulos necesarios aquí. Necesitamos el módulo secuencial para inicializar la red neuronal y el módulo denso para agregar las capas ocultas.

```
# Importing the Keras libraries and packages
import keras
from keras.models import Sequential
from keras.layers import Dense
```

Paso 9

Denominaremos el modelo como Clasificador ya que nuestro objetivo es clasificar la rotación de clientes. Luego usamos el módulo secuencial para la inicialización.

```
#Initializing Neural Network
classifier = Sequential()
```

Paso 10

Agregamos las capas ocultas una por una usando la función densa. En el siguiente código, veremos muchos argumentos.

Nuestro primer parámetro es **output_dim**. Es la cantidad de nodos que agregamos a esta capa. **init** es la inicialización del gradiente estocástico decente. En una red neuronal asignamos pesos a cada nodo. En la inicialización, los pesos deben estar cerca de cero y los inicializamos aleatoriamente usando la función uniforme. El parámetro **input_dim** es necesario solo para la primera capa, ya que el modelo no conoce el número de nuestras variables de entrada. Aquí el número total de variables de entrada es 11. En la segunda capa, el modelo conoce automáticamente el número de variables de entrada de la primera capa oculta.

Ejecute la siguiente línea de código para agregar la capa de entrada y la primera capa oculta:

```
classifier.add(Dense(units = 6, kernel_initializer = 'uniform',
activation = 'relu', input_dim = 11))
```

Ejecute la siguiente línea de código para agregar la segunda capa oculta:

```
classifier.add(Dense(units = 6, kernel_initializer = 'uniform',
activation = 'relu'))
```

Ejecute la siguiente línea de código para agregar la capa de salida:

```
classifier.add(Dense(units = 1, kernel_initializer = 'uniform',
activation = 'sigmoid'))
```

Paso 11

Compilando el ANN

Hemos agregado varias capas a nuestro clasificador hasta ahora. Ahora los compilaremos utilizando el método de **compilación**. Los argumentos agregados en el control de compilación final completan la red neuronal, por lo que debemos ser cuidadosos en este paso.

Aquí hay una breve explicación de los argumentos.

El primer argumento es **Optimizer**. Este es un algoritmo utilizado para encontrar el conjunto óptimo de pesos. Este algoritmo se llama **Descenso de gradiente estocástico (SGD)**. Aquí estamos usando uno entre varios tipos, llamado 'Adam optimizer'. El SGD depende de la pérdida, por lo que nuestro segundo parámetro es la pérdida. Si nuestra variable dependiente es binaria, usamos la función de pérdida logarítmica llamada '**binary_crossentropy**' , y si nuestra variable dependiente tiene más de dos categorías en la salida, entonces usamos '**categorical_crossentropy**' . Queremos mejorar el rendimiento de nuestra red neuronal en función de la **precisión** , por lo que agregamos **métricas** como precisión.

```
# Compiling Neural Network  
classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy'])
```

Paso 12

Se deben ejecutar varios códigos en este paso.

Ajuste de la ANN al conjunto de entrenamiento

Ahora entrenamos nuestro modelo en los datos de entrenamiento. Usamos el **ajuste** método para adaptarse a nuestro modelo. También optimizamos los pesos para mejorar la eficiencia del modelo. Para esto, tenemos que actualizar los pesos. **El tamaño del lote** es el número de observaciones después de las cuales actualizamos los pesos. **La época** es el número total de iteraciones. Los valores de tamaño de lote y época se eligen por el método de prueba y error.

```
classifier.fit(X_train, y_train, batch_size = 10, epochs = 50)
```

Hacer predicciones y evaluar el modelo.

```
# Predicting the Test set results  
y_pred = classifier.predict(X_test)  
y_pred = (y_pred > 0.5)
```

Predecir una sola observación nueva

```
# Predicting a single new observation  
"""Our goal is to predict if the customer with the following data will leave the bank:  
Geography: Spain  
Credit Score: 500  
Gender: Female  
Age: 40  
Tenure: 3  
Balance: 50000  
Number of Products: 2  
Has Credit Card: Yes  
Is Active Member: Yes
```

Paso 13

Predecir el resultado del conjunto de prueba

El resultado de la predicción le dará la probabilidad de que el cliente abandone la empresa. Convertiremos esa probabilidad en binario 0 y 1.

```
# Predicting the Test set results  
y_pred = classifier.predict(X_test)  
y_pred = (y_pred > 0.5)
```

```
new_prediction = classifier.predict(sc.transform  
(np.array([[0.0, 0, 500, 1, 40, 3, 50000, 2, 1, 1, 40000]])))  
new_prediction = (new_prediction > 0.5)
```

Paso 14

Este es el último paso donde evaluamos el rendimiento de nuestro modelo. Ya tenemos resultados originales y, por lo tanto, podemos construir una matriz de confusión para verificar la precisión de nuestro modelo.

Haciendo la matriz de confusión

```
from sklearn.metrics import confusion_matrix  
cm = confusion_matrix(y_test, y_pred)  
print (cm)
```

Salida

```
loss: 0.3384 acc: 0.8605  
[ [1541 54]  
[230 175] ]
```

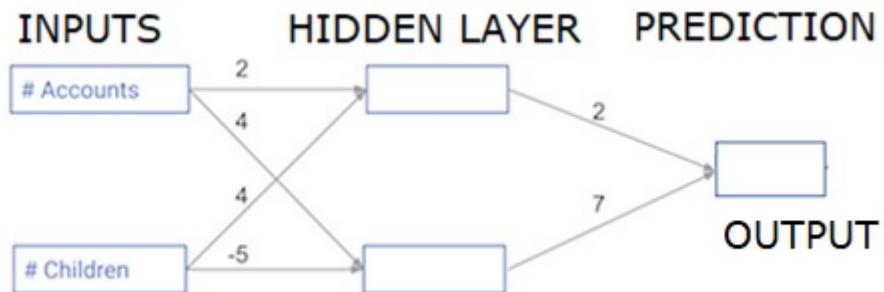
A partir de la matriz de confusión, la precisión de nuestro modelo se puede calcular como:

```
Accuracy = 1541+175/2000=0.858
```

Logramos una precisión del 85.8% , lo cual es bueno.

El algoritmo de propagación directa

En esta sección, aprenderemos cómo escribir código para hacer propagación hacia adelante (predicción) para una red neuronal simple:



Cada punto de datos es un cliente. La primera entrada es cuántas cuentas tienen, y la segunda entrada es cuántos hijos tienen. El modelo predecirá cuántas transacciones realizará el usuario en el próximo año.

Los datos de entrada se cargan previamente como datos de entrada, y los pesos están en un diccionario llamado pesos. La matriz de pesos para el primer nodo en la capa oculta está en pesos ['nodo_0'], y para el segundo nodo en la capa oculta está en pesos ['nodo_1'] respectivamente.

Los pesos que se alimentan al nodo de salida están disponibles en pesos.

La función de activación lineal rectificada

Una "función de activación" es una función que funciona en cada nodo. Transforma la entrada del nodo en alguna salida.

La función de activación lineal rectificada (llamada *ReLU*) se usa ampliamente en redes de muy alto rendimiento. Esta función toma un solo número como entrada, devuelve 0 si la entrada es negativa, y la entrada como salida si la entrada es positiva.

Aquí hay algunos ejemplos:

- $\text{relu}(4) = 4$
- $\text{relu}(-2) = 0$

Completamos la definición de la función *relu* ()

- Usamos la función *max* () para calcular el valor de la salida de *relu* ().
- Aplicamos la función *relu* () a *node_0_input* para calcular *node_0_output*.
- Aplicamos la función *relu* () a *node_1_input* para calcular *node_1_output*.

```

import numpy as np
input_data = np.array([-1, 2])
weights = {
    'node_0': np.array([3, 3]),
    'node_1': np.array([1, 5]),
    'output': np.array([2, -1])
}
node_0_input = (input_data * weights['node_0']).sum()
node_0_output = np.tanh(node_0_input)
node_1_input = (input_data * weights['node_1']).sum()
node_1_output = np.tanh(node_1_input)
hidden_layer_output = np.array(node_0_output, node_1_output)
output =(hidden_layer_output * weights['output']).sum()
print(output)
  
```

```

def relu(input):
    '''Define your relu activation function here'''
    # Calculate the value for the output of the relu function: output
    output = max(input,0)
    # Return the value just calculated
    return(output)

# Calculate node 0 value: node_0_output
node_0_input = (input_data * weights['node_0']).sum()
node_0_output = relu(node_0_input)

# Calculate node 1 value: node_1_output
node_1_input = (input_data * weights['node_1']).sum()
node_1_output = relu(node_1_input)

# Put node values into array: hidden_layer_outputs
hidden_layer_outputs = np.array([node_0_output, node_1_output])

# Calculate model output (do not apply relu)
model_output = (hidden_layer_outputs * weights['output']).sum()
print(model_output)# Print model output

```

Salida

```

0.9950547536867305
-3

```

Aplicando la red a muchas observaciones / filas de datos

En esta sección, aprenderemos cómo definir una función llamada `predict_with_network()`. Esta función generará predicciones para múltiples observaciones de datos, tomadas de la red anterior tomada como `input_data`. Se están utilizando los pesos dados en la red anterior. La definición de la función `relu()` también se está utilizando.

Definamos una función llamada `predict_with_network()` que acepta dos argumentos - `input_data_row` y `weights` - y devuelve una predicción de la red como salida.

Calculamos los valores de entrada y salida para cada nodo, almacenándolos como: `node_0_input`, `node_0_output`, `node_1_input` y `node_1_output`.

Para calcular el valor de entrada de un nodo, multiplicamos las matrices relevantes y calculamos su suma.

Para calcular el valor de salida de un nodo, aplicamos la función `relu()` al valor de entrada del nodo. Usamos un 'bucle for' para iterar sobre `input_data` -

También usamos nuestro `predict_with_network()` para generar predicciones para cada fila de `input_data` - `input_data_row`. También agregamos cada predicción a los resultados.

```

# Define predict_with_network()
def predict_with_network(input_data_row, weights):
    # Calculate node 0 value
    node_0_input = (input_data_row * weights['node_0']).sum()
    node_0_output = relu(node_0_input)

```

```

# Calculate node 1 value
node_1_input = (input_data_row * weights['node_1']).sum()
node_1_output = relu(node_1_input)

# Put node values into array: hidden_layer_outputs
hidden_layer_outputs = np.array([node_0_output, node_1_output])

# Calculate model output
input_to_final_layer = (hidden_layer_outputs*weights['output']).sum()
model_output = relu(input_to_final_layer)

# Return model output
return(model_output)

# Create empty List to store prediction results
results = []
for input_data_row in input_data:
    # Append prediction to results
    results.append(predict_with_network(input_data_row, weights))
print(results) # Print results

```

Salida

```
[0, 12]
```

Aquí hemos usado la función relu donde $\text{relu}(26) = 26$ y $\text{relu}(-13) = 0$ y así sucesivamente.

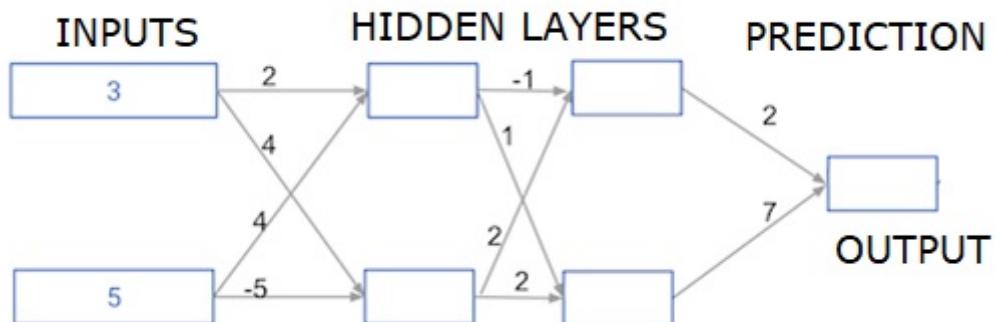
Redes neuronales profundas multicapa

Aquí estamos escribiendo código para hacer propagación hacia adelante para una red neuronal con dos capas ocultas. Cada capa oculta tiene dos nodos. Los datos de entrada se han precargado como **input_data**. Los nodos en la primera capa oculta se denominan **node_0_0** y **node_0_1**.

Sus pesos están precargados como pesos **['node_0_0']** y pesos **['node_0_1']** respectivamente.

Los nodos en la segunda capa oculta se llaman **nodo_1_0** y **nodo_1_1**. Sus pesos están precargados como **pesos ['nodo_1_0']** y **pesos ['nodo_1_1']** respectivamente.

Luego creamos un modelo de salida de los nodos ocultos usando pesos precargados como **pesos ['salida']**.



Calculamos node_0_0_input usando sus pesos ['node_0_0'] y los input_data dados. Luego aplique la función relu () para obtener node_0_0_output.

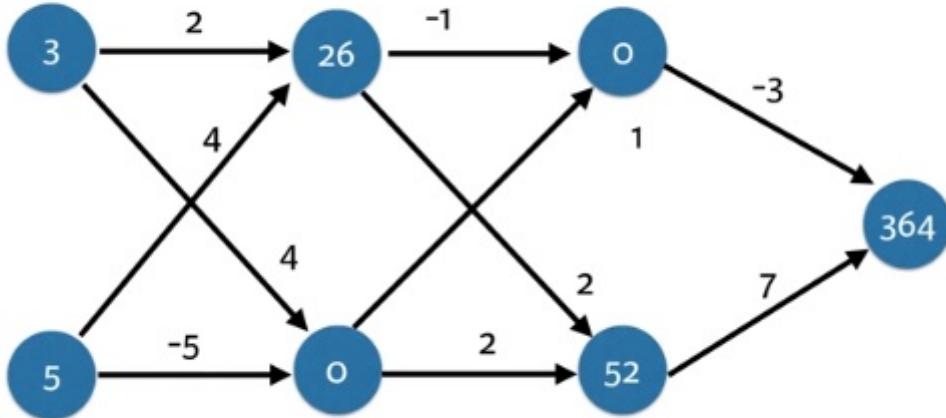
Hacemos lo mismo que anteriormente para node_0_1_input para obtener node_0_1_output.

Calculamos node_1_0_input usando sus pesos ponderaciones ['node_1_0'] y las salidas de la primera capa oculta: hidden_0_outputs. Luego aplicamos la función relu () para obtener node_1_0_output.

Hacemos lo mismo que anteriormente para node_1_1_input para obtener node_1_1_output.

Calculamos model_output usando pesos ['output'] y los resultados de la segunda capa oculta array hidden_1_outputs. No aplicamos la función relu () a esta salida.

MULTIPLE HIDDEN LAYERS



CALCULATING WITH ReLU ACTIVATION FUNCTION

```
import numpy as np
input_data = np.array([3, 5])
weights = {
    'node_0_0': np.array([2, 4]),
    'node_0_1': np.array([4, -5]),
    'node_1_0': np.array([-1, 1]),
    'node_1_1': np.array([2, 2]),
    'output': np.array([2, 7])
}
def predict_with_network(input_data):
    # Calculate node 0 in the first hidden Layer
    node_0_0_input = (input_data * weights['node_0_0']).sum()
    node_0_0_output = relu(node_0_0_input)

    # Calculate node 1 in the first hidden Layer
    node_0_1_input = (input_data*weights['node_0_1']).sum()
    node_0_1_output = relu(node_0_1_input)

    # Put node values into array: hidden_0_outputs
    hidden_0_outputs = np.array([node_0_0_output, node_0_1_output])

    # Calculate node 0 in the second hidden Layer
    node_1_0_input = (hidden_0_outputs*weights['node_1_0']).sum()
    node_1_0_output = relu(node_1_0_input)
```

```
# Calculate node 1 in the second hidden layer
node_1_1_input = (hidden_0_outputs*weights['node_1_1']).sum()
node_1_1_output = relu(node_1_1_input)

# Put node values into array: hidden_1_outputs
hidden_1_outputs = np.array([node_1_0_output, node_1_1_output])

# Calculate model output: model_output
model_output = (hidden_1_outputs*weights['output']).sum()
    # Return model_output
return(model_output)
output = predict_with_network(input_data)
print(output)
```

Salida

```
364
```