Redes Neuronales

Las redes neuronales son un framework de machine learning que intenta imitar el patrón de aprendizaje de las neuronas biológicas naturales.

Las neuronas biológicas naturales tienen neuronas conectadas mediante dendritas que reciben las entradas, luego basado en esas entradas producen una señal de salida a través de un axón hacia otra neurona.

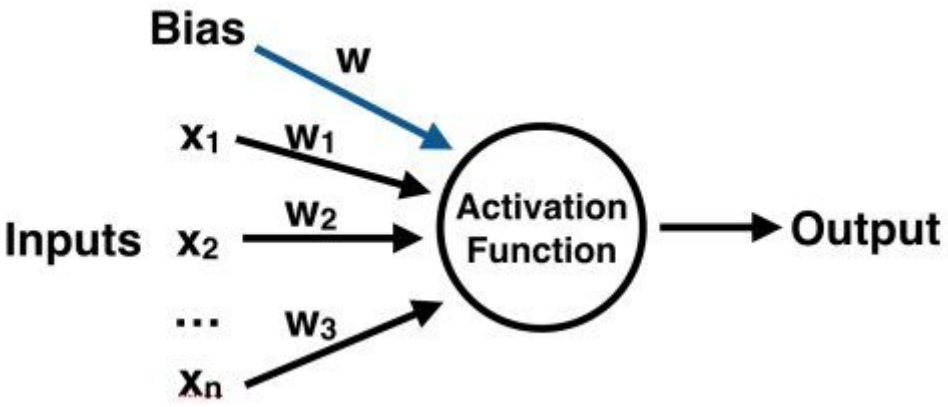
Vamos a tratar de imitar este proceso a través del uso de las Redes Neuronales Artificiales (RNA), a las cuales sólo llamaremos Redes Neuronales a partir de este momento. El proceso para crear una red neuronal empieza con la forma más básica, un sólo perceptrón.

Perceptron

Empecemos hablando del perceptrón, un perceptrón tiene una o más entradas, un bias, una función de activación y una única salida.

El perceptrón recibe entradas, las multiplica por algún peso, y luego las pasa a través de una función de activación para producir una salida. Hay muchas funciones de activación posibles para escoger, como la función logística, una función trigonométrica, una step function etc.

También nos aseguramos de agregar un bias al perceptrón, esto evita problemas donde todas las entradas tiendan a cero (que ningún peso tenga un efecto).

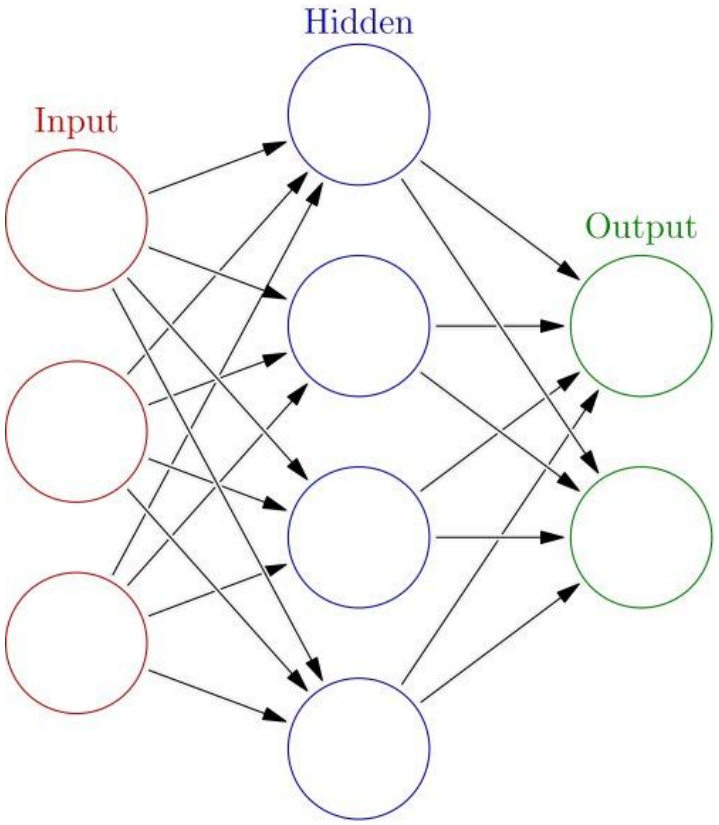


Una vez que ya tenemos la salida podemos compararla con un label definido y real lo que nos permitirá ajustar los pesos de acuerdo a lo necesario (los pesos normalmente se inician con valores al azar).

Repetimos este proceso hasta que hayamos alcanzado el máximo número de iteraciones posibles o un tasa de error aceptable.

Para crear una red neuronal, simplemente empezamos a agregar capas de perceptrones, creando una red neuronal modelo multicapa de perceptrones (Multilayer perceptron model of a neural network).

Se tendrá una capa de entrada que tomará directamente los features de entrada y una capa de salida que creará las salidas resultantes. Cualquier capa en medio de estas es conocida como capa escondida por que no “ve” directamente los features de entrada o las salidas.



Ahora crearemos una red neuronal con Python.

Datos:

Usaremos “Breast Cancer Data Set” que es un dataset que viene incluido en la librería de SciKit Learn, este tiene diferente features de tumores con una “labeled class” que indica si el tumor es Maligno o Benigno. Trataremos de crear una red neuronal que pueda tomar estos features e intentar predecir la label ya sea maligna o benigna para los tumores que no ha visto.

Obtengamos los datos:

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
```

El objeto “`cancer`” se comporta como un diccionario, contiene la descripción de los datos, los features y los targets o labels.

```
cancer.keys()
```

```
dict_keys(['DESCR', 'feature_names', 'target_names', 'target',
'data'])
```

Podemos ver la descripción del objeto mediante la siguiente línea de código:

```
print(cancer['DESCR'])
```

Podemos ver la dimensión de la matriz de datos:

```
cancer['data'].shape
```

```
(569, 30)
```

Acomodemos nuestros datos y labels:

```
X = cancer['data']
y = cancer['target']
```

Ahora dividamos nuestros datos en set de entrenamiento y set de prueba, esto se hace fácil usando la función `train_test_split` dentro de `model_selection` que es parte de SciKit Learn:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y)
```

Ahora trabajaremos en el preprocesamiento de los datos, lo necesitamos porque la red neuronal puede tener un poco de problemas llegando al valor esperado antes del máximo número de iteraciones permitidas, si los datos no están normalizados.

El modelo Multi-layer perceptron es sensible a las escalas de los features, por lo tanto es recomendable escalar los datos. Hay diferentes métodos para hacerlo, pero en este ejemplo usaremos una función incluida, llamada `StandardScaler` para poder estandarizar.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
```

Uusamos solo la matriz de entrenamiento

```
scaler.fit(X_train)
```

```
StandardScaler(copy=True, with_mean=True, with_std=True)
```

Ahora aplicamos la transformación a todos los datos:

```
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

Es momento de entrenar nuestro modelo. SciKit Learn nos hará esto sumamente sencillo, usando un algoritmo definido. En este caso nuestro algoritmo importamos `"Multi-layer Perceptron Classifier"` de la librería `"neural_network"` de SciKit Learn.

```
from sklearn.neural_network import MLPClassifier
```

Lo que sigue es crear una instancia de este modelo, hay varios parámetros que podemos modificar pero en este caso solo modificaremos `"hidden_layer_sizes"`. Para este parámetro se pasa una tupla que consiste en el número de neuronas que se busca en cada capa, donde la entrada n en la tupla representa el número de neuronas en la n capa del modelo. Hay varias maneras de escoger esta medida, por simplicidad escogeremos 3 capas con el mismo número de neuronas como features que tiene el data set:

```
mlp = MLPClassifier(hidden_layer_sizes=(30,30,30))
```

Ahora que el modelo ya se creó, podemos entrenarlo con los datos de entrenamiento, recordando que estos datos ya fueron procesados y escalados.

```
mlp.fit(X_train,y_train)
```

```
MLPClassifier(activation='relu', alpha=0.0001, batch_size='auto', beta_1=0.9,
       beta_2=0.999, early_stopping=False, epsilon=1e-08,
       hidden_layer_sizes=(30, 30, 30), learning_rate='constant',
       learning_rate_init=0.001, max_iter=200, momentum=0.9,
       nesterovs_momentum=True, power_t=0.5, random_state=None,
       shuffle=True, solver='adam', tol=0.0001, validation_fraction=0.1,
       verbose=False, warm_start=False)
```

Podemos ver en la salida que nos muestra los valores por default de los otros parámetros en el modelo. Les recomiendo jugar con estos valores para ver lo que produce en el modelo.

Ahora que ya tenemos un modelo es momento de usarlo para obtener predicciones, Podemos hacer esto con el método `predict()` de nuestro modelo entrenado:

```
predictions = mlp.predict(X_test)
```

Ya con las predicciones podemos utilizar las métricas incluidas en SciKit Learn como el reporte de clasificación y la matriz de confusión para evaluar qué tan bueno es el desempeño de nuestro modelo:

```
from sklearn.metrics import classification_report,confusion_matrix
print(confusion_matrix(y_test,predictions))
[[50  3]
 [ 0 90]]


print(classification_report(y_test,predictions))

             precision    recall  f1-score   support

          0       1.00      0.94      0.97        53
          1       0.97      1.00      0.98        90

avg / total       0.98      0.98      0.98       143
```

Al parecer solo clasificamos mal 3 tumores, eso nos da un 98% de “accuracy” así como 98% de precisión y recall. Estos valores son muy buenos considerando la cantidad de líneas de código que utilizamos. El lado no tan bueno es lo difícil que es interpretar el modelo por si mismo. Los pesos y el bias no pueden ser interpretados de manera sencilla en relación a cuál de los features son importantes para el modelo en sí.

Sin embargo, si queremos extraer los pesos y bias después de entrenar el modelo, podemos usar los atributos `coefs_` e `intercepts_`.

`coefs_` es una lista de matrices de pesos, donde la matriz en el índice i representa los pesos entre la capa i y la capa i+1.

`intercepts_` es una lista de vectores de bias, donde el vector en el índice i representa el valor del bias en la capa i+1.

```
len(mlp.coefs_)
4

len(mlp.coefs_[0])
30

len(mlp.intercepts_[0])
30
```