4-11-2021

SISTEMAS INTELIGENTES I

Entrega Práctica 1 .- Arquitecturas Supervisadas (BPNN) con Python&Keras

4ºCurso

Alejandro Daniel Herrera Cardenes
Martin Van Puffelen López
UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

Indice

Introducción

Se nos presenta la siguiente práctica que consiste en diseñar una Arquitectura Neuronal Supervisada para la Clasificación de Patrones de un dataset de Transporte de Mercancías Peligrosas y comparar dos modelos:

- Modelo 1: Desarrollo de esta arquitectura solo usando Python
- Modelo 2: Desarrollo de esta arquitectura usando Keras

Descripción

Dataset

Para empezar, observamos como está organizada el dataset y vemos que contiene 65536 filas y 14 columnas organizadas como:

- Fecha hora
- Longitud vehículo
- Carril circulación
- Velocidad vehículo
- Peso vehículo
- Numero ejes
- Temperatura aire
- Humedad relativa
- Tipo precipitación
- Intensidad precipitación
- Dirección viento
- Velocidad viento
- Estado carretera
- Accidente

Partimos de la base de que hay que tener en cuenta que el factor humano supone el 94% de los accidentes de tráfico (datos de National Highway Traffic Safety Administration (NHTSA)). Estos fallos incluyen poca atención a la carretera y no visualizar a los demás coches u objetos. Sabiendo esto, las variables que tenemos en nuestro dataset son todas relacionadas al aspecto técnico de los automóviles, condiciones meteorológicas en cada momento. La próxima decisión humana relevante es la velocidad de circulación justo en una zona comprometida, como una curva o carretera con un estado poco óptimo o en un momento donde las condiciones sean desfavorables. En las características mencionadas anteriormente se aprecia que no hay ninguna de carácter humano directo pero creemos que es importante tenerlo en cuenta.

A la hora del preprocesado del dataset transformamos los datos tanto de String como los Object a Float64 para que de menos problemas a la red. También vimos que había columnas con datos vacíos pero la librería Numpy nos ayuda a eliminarlos.

Hemos decidido eliminar la columna de Fecha y hora porque nos resultó irrelevante a la hora de entrenar la red (podría ser interesante establecer una zona horaria del día ya sea mañana tarde y noche para mejor clasificación).

También cabe añadir que normalizamos los datos para que mejore el entrenamiento, esto quiere decir que las muestras con valores continuos estarán dentro del rango 0 a 1. Esto se ha conseguido dividiendo cada columna entre el máximo.

Arquitectura

Referente a la arquitectura de la red, deberá tener 12 neuronas en la capa de entrada, 1 capa oculta con 4 neuronas y la correspondiente capa de salida con 1 neurona.

Nuestra capa de salida en esta red será la última columna que es la representación de la predicción de accidentes, siendo la salida 0 como 'No' accidente y 1 como 'Si' accidente.

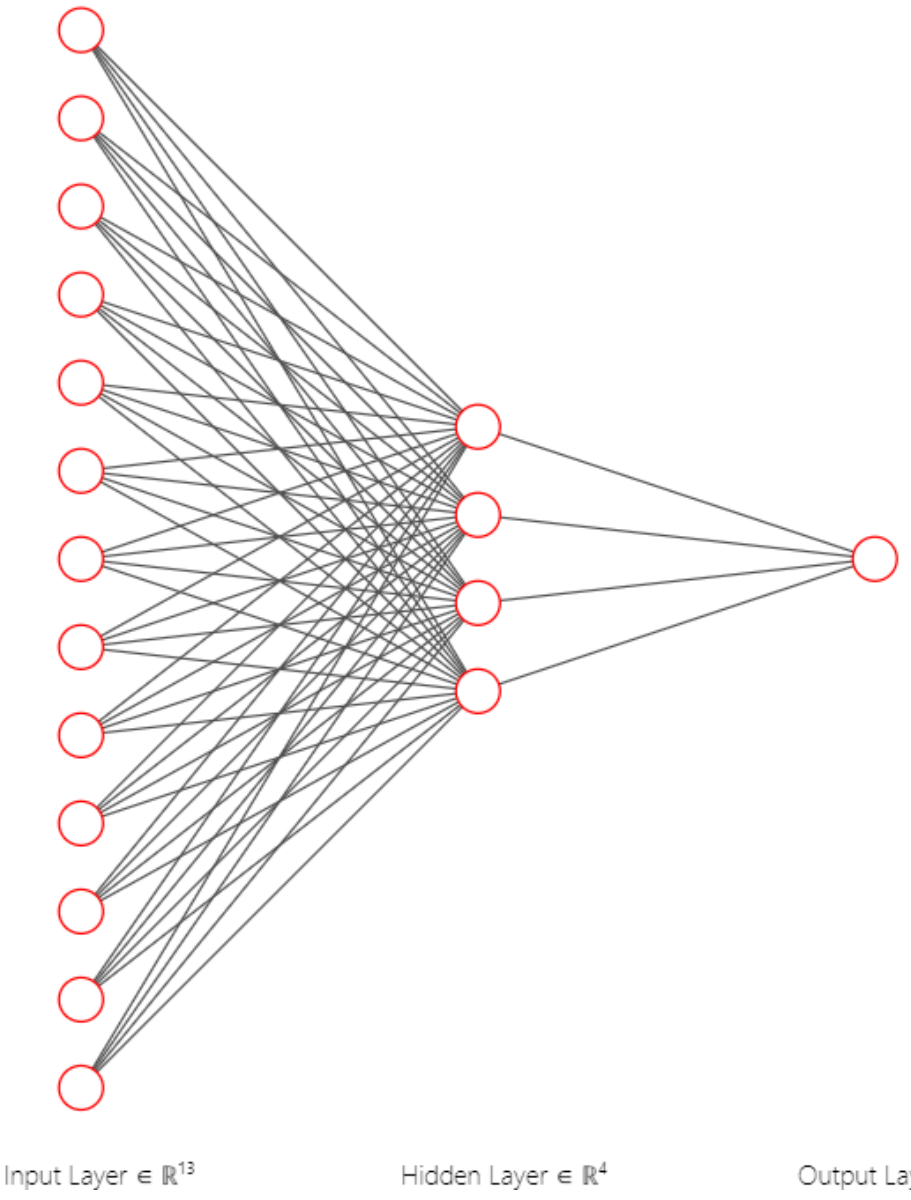


Diagrama de la Arquitectura

Función costo

Función de costo (C): entropía cruzada. Es una función con múltiples aplicaciones en la estadística pero en el área de la inteligencia artificial se puede interpretar como la relación entre la probabilidad de clasificación de una etiqueta real y nuestro valor precedido.

$$CrossEntropyLoss = -(y_i log(\hat{y}_i) + (1 - y_i) log(1 - \hat{y}_i))$$

Pérdida de entropía cruzada.

Esta función de costo funciona muy bien en problemas de clasificación con el rango de valores entre 0 y 1 y a la hora de la práctica se puede definir como (para dos clases):

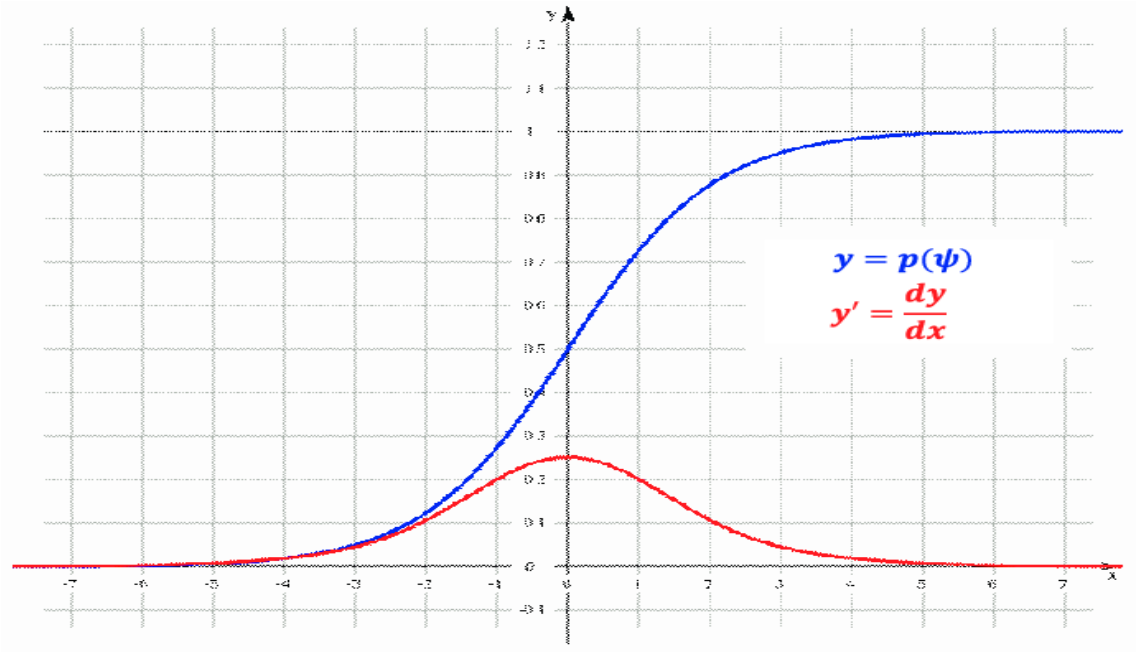
$$-\log(o)$$

Algoritmo del descenso del gradiente (estocástico)

Teóricamente, para aplicar la regla de aprendizaje mediante el descenso del gradiente, se debe computar muestra por muestra y hacer una media de la función costo. Esto supone, que si el dataset es muy grande, el caso común hoy en día, la fase de entrenamiento sería muy costosa o imposible computacionalmente. Por ello existe el descenso del gradiente estocástico que agiliza el aprendizaje estimando el gradiente computando en base a un mini conjunto de muestras aleatorios del conjunto de datos (mini batch).

Neurona sigmoidal

Para nuestra arquitectura hemos decidido establecer como neuronas la neurona sigmoidal, ya que nos permitirá aplicar el descenso por el gradiente y backpropagation siendo derivable. Además, la derivada es muy sencilla de computar y la salida estará en el rango de 0 y 1.



Representación de la función sigmoidal

Métricas de evaluación (Accuracy y AUC)

El **accuracy** simplemente calcula la frecuencia de las predicciones sean iguales al valor real de salida (target). Es la métrica mas usada y muy sencilla de interpretar.

AUC mide la capacidad del modelo a clasificar entre clases. En este caso cuanto mayor sea el área debajo de la curva (ROC) mejor será el rendimiento en la clasificación.

Backpropagation

La propagación hacia atrás de errores o retro propagación (del inglés **backpropagation**) es un método de cálculo del gradiente utilizado en algoritmos de aprendizaje supervisado mediante la regla de la cadena.

En nuestra implementación nos hemos focalizado en la facilidad del proceso para comprender como se retro propagan los errores y se ejecuta el proceso de aprendizaje.

Aceptara una tupla de características – etiquetas reales (target) para seleccionar una serie de muestras aleatorias con el tamaño del mini batch que se establece al crear la red.

Tras eso se hará el feed forward que es el sumatorio de los pesos por la multiplicación de las entradas mas el bias y pasara por la función de activación que en nuestro caso es la sigmoide. Tras esto se calculará el error y comenzará la retro propagación.

Notar que a la hora de actualizar los pesos se usara la función de actualización mediante el descenso por el gradiente estocástico:

$$\nabla C \approx \frac{1}{m} \sum_{j=1}^{m} \nabla C_{X_j},$$

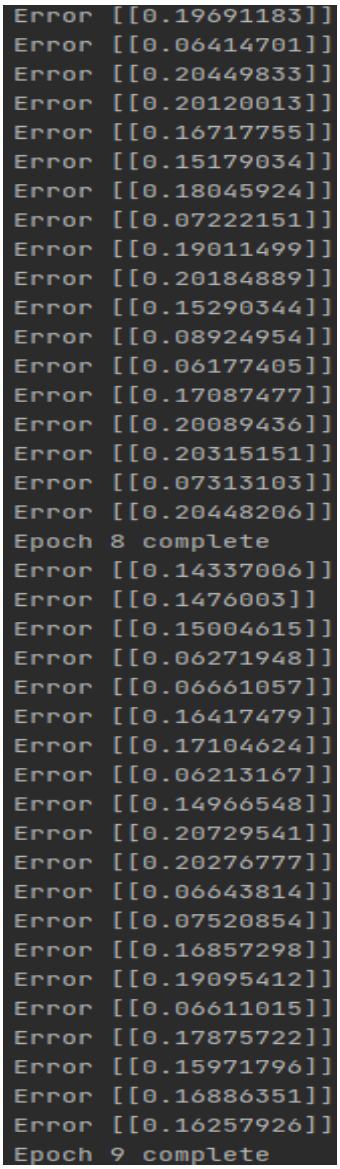
Representación del gradiente del costo con mini batch

Resultados

No hemos sabido interpretar las salidas de nuestra implementación ya que el proceso de limpieza del dataset nos ha tomado bastante tiempo y el desarrollar cada uno de los conceptos detrás del backpropagation nos ha impedido tener una conclusión concreta en esta parte. Debido a esto, no implementamos la entropía cruzada si no el Error cuadrático.

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

Demostración error cuadrático



Conjunto de errores Modelo 1

Como podemos apreciar en cada epoch (se define como una iteración en el aprendizaje que recorre todas las muestras en el conjunto de entrenamiento), tenemos 20 errores que es el tamaño de nuestro mini batch y estos mismos no siguen un descenso continuo si no varían al alza o a la baja según suponemos por la muestra evaluada en sí. Todo esto dificulta nuestra interpretación.

Estaríamos interesados en como arreglar el algoritmo para que den errores lógicos.

Backpropagation con Keras

TensorFlow es una API de Google de bajo nivel que hace muy eficiente la computación orientada al aprendizaje automático y redes neuronales.

Keras es un framework implementado en TensorFlow que permite una aplicación a alto nivel de forma más sencilla.

Por motivos académicos y de comparación hemos construido una red análoga a la anterior implementación sin ninguna librería (nuevamente 12 entradas, 4 neuronas en 1a capa oculta y una neurona en la capa de salida).

Hemos usado un modelo secuencial con dos conexiones 'Dense'. Como optimizador hemos usado el 'SGD' (descenso gradiente estocástico).

La función costo será la entropía cruzada binaria (apartado costo).

Y las métricas serán el 'Accuracy' y el 'AUC' (apartado de métricas).

Resultados

La facilidad de la implementación nos ha sorprendido y nos ha permitido evaluar con un test estableciendo el mini batch de tamaño 20 muestras y realizando un costoso descenso por el gradiente 'online'.

```
Epoch 45/50
2294/2294 [==============================] - 1s 390us/step - loss: 0.1262 - accuracy: 0.9355 - auc: 0.9586
Epoch 46/50
2294/2294 [==============================] - 1s 390us/step - loss: 0.1259 - accuracy: 0.9358 - auc: 0.9588
Epoch 47/50
2294/2294 [==============================] - 1s 391us/step - loss: 0.1257 - accuracy: 0.9363 - auc: 0.9590
Epoch 48/50
2294/2294 [==============================] - 1s 390us/step - loss: 0.1254 - accuracy: 0.9360 - auc: 0.9592
Epoch 49/50
2294/2294 [==============================] - 1s 389us/step - loss: 0.1250 - accuracy: 0.9364 - auc: 0.9594
Epoch 50/50
2294/2294 [==============================] - 1s 389us/step - loss: 0.1247 - accuracy: 0.9365 - auc: 0.9596
615/615 [==============================] - 0s 443us/step - loss: 0.0353 - accuracy: 0.9843 - auc: 0.9896
```

Batch size=20, 50 epochs

```
Epoch 44/50
45874/45874 [==============================] - 20s 442us/step - loss: 0.0931 - accuracy: 0.9529 - auc: 0.9773
Epoch 45/50
45874/45874 [==============================] - 20s 437us/step - loss: 0.0930 - accuracy: 0.9529 - auc: 0.9771
Epoch 46/50
45874/45874 [==============================] - 20s 446us/step - loss: 0.0920 - accuracy: 0.9537 - auc: 0.9778
Epoch 47/50
45874/45874 [==============================] - 20s 444us/step - loss: 0.0925 - accuracy: 0.9533 - auc: 0.9772
Epoch 48/50
45874/45874 [==============================] - 20s 433us/step - loss: 0.0916 - accuracy: 0.9540 - auc: 0.9781
Epoch 49/50
45874/45874 [==============================] - 20s 442us/step - loss: 0.0920 - accuracy: 0.9538 - auc: 0.9777
Epoch 50/50
45874/45874 [==============================] - 20s 433us/step - loss: 0.0915 - accuracy: 0.9540 - auc: 0.9783
615/615 [==============================] - 0s 395us/step - loss: 0.0182 - accuracy: 0.9913 - auc: 0.9966
```

Batch size=1, 50 epochs

Descenso por el gradiente en serie o en línea ("online gradient descent"), actualiza la configuración de la red para cada muestra, es decir, cada epoch recorrerá la red para cada una de las muestras.

Conclusión

En el extremadamente lento y costoso descenso por el gradiente en serie o en línea, es muy interesante apreciar la diferencia entre ambos resultados teniendo en cuenta que el descenso por el gradiente estocástico con tamaño del batch a 20 hace un muy buen trabajo: la red aprende en menos de un minuto y encima la diferencia con el "online" es sólo de un 2%. Una buena conclusión de esto es que lo mejor es buscar una solución que sea factible computacionalmente teniendo en cuenta el problema y cuán importante sea la precisión de las predicciones.

Nos hemos dado cuenta a su vez que el preprocesado de datos puede ser muy tedioso y forma parte elemental a la hora de diseñar un sistema inteligente. En la primera parte de la práctica hemos sido limitados por esto mismo y nos ha hecho ver la importancia de esta fase.

Aunque no hemos sido capaces de analizar los errores en la implementación del Modelo 1, hemos comprendido el funcionamiento del algoritmo y todas las implicaciones matemáticas que abarca haciendo que funcione casi cualquier arquitectura multicapa supervisada: backpropagation.

Para finalizar, nos gustaría en un futuro saber que conclusiones son las que se deben sacar tras desarrollar un modelo de aprendizaje inteligente y el diseño de la arquitectura, por ejemplo que variables son las más importantes a la hora de tener un accidente, que variables son las menos relevantes, etc.

Enlace github

Pasamos el enlace github en caso de que tenga problemas con el código pasado por zip.

- https://github.com/martinvplopez/BPNN-SI1

Bibliografía

- https://www.sciencedirect.com/science/article/pii/S0001457518300873#tbl0005
- https://www.youtube.com/
- http://neuralnetworksanddeeplearning.com/chap1.html#exercises_647181 written by Michael Nielsen
- https://es.wikipedia.org/wiki/Propagaci%C3%B3n_hacia_atr%C3%A1s
- https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/