Ciencia de datos

Introduction to Deep-Learning Concepts and TensorFlow

Pedro Arturo Flores Silva
email: flosipan@ciencias.unam.mx[1]

[1]IIMAS, Facultad de Ciencias, UNAM

1 Neuronas Artificiales

Las neuronas artificiales están inspiradas en las neuronas biológicas, con algunas modificaciones por conveniencia. En estas neuronas artificiales, similar a las dendritas naturales, las conexiones de entrada llevan señales de entrada, ya sea atenuadas o amplificadas desde otras neuronas artificiales vecinas. Las señales pasan hacia la neurona, donde son ponderadas y se toma una decisión en función a la señal total recibida. Por ejemplo, para una neurona de umbral binario, la respuesta tendrá valor 1 únicamente cuando la señal de entrada supera un umbral predefinido. De otra manera, la respuesta será cero. Existen otros tipos de neuronas que son empleados en las redes neuronales artificiales, sin embargo, solo difieren con respecto a la función de activación que depende de la señal de entrada total. La estructura de una neurona artificial se muestra en la figura 1.

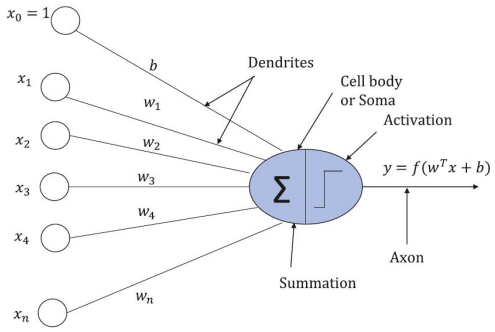

Figure 1: Estructura de una neurona artificial. Tomada de [1]

2 El Perceptrón

Los perceptrones son clasificadores binarios lineales que construyen un hiperplano que separa las dos clases. Este hiperplano está representado por un vector unitario de pesos $w' \in \mathbb{R}^{n\times 1}$ que es perpendicular al hiperplano y un termino de sesgo b que determina la distancia del origen al hiperplano. El vector w' se elige de tal forma que apunte en dirección a una de las clases que se denomina positiva (ver imagen 2).

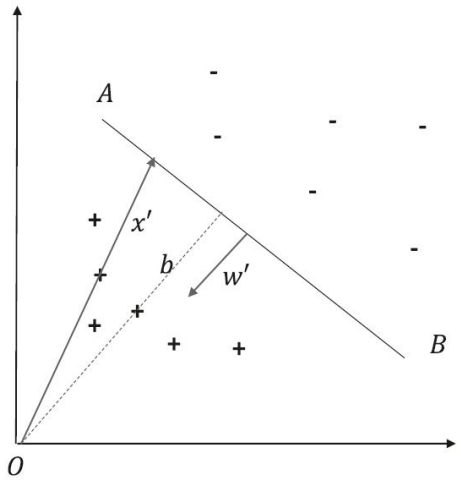

Figure 2: Hiperplano separador de ambas clases. Tomada de [1]

De la figura 2 es sencillo ver que para cualquier vector $x' \in \mathbb{R}^{n\times 1}$ cuyas entradas sean positivas, el producto interno con el negativo del vector w' da la distancia b. Formalmente, para puntos situados en el hiperplano:

$$-w'^T x' = b \iff w'^T x' + b = 0 \tag{1}$$

De forma análoga, para puntos situados debajo del hiperplano (clase positiva) se tiene que:

$$-w'^T x' < b \iff w'^T x' + b > 0 \tag{2}$$

Y para los puntos situados por encima del hiperplano (clase negativa) se tiene:

$$-w'^T x' > b \iff w'^T x' + b < 0 \tag{3}$$

En el área de aprendizaje de maquinas, la tarea consiste en aprender los parámetros del hiperplano, es decir, aprender o estimar w' y b. Generalmente, para simplificar el termino b se considera como parte de w' con valor constante de 1. De esta forma tenemos señales o datos de entrada de la forma $x = (1, x_1, x_2, ..., x_n)^T$ y parámetros a estimar $w = (b, w_1, w_2, ..., w_n)^T$. En consecuencia las reglas de clasificación son:

- $w^T x = 0$ corresponde al hiperplano y todos los puntos $x \in \mathbb{R}^{(n\times 1)\times 1}$ que yacen en el mismo.
- $w^T x > 0$ corresponden a todos los puntos de la clase positiva.
- $w^T x \leq 0$ corresponden a todos los puntos de la clase negativa. La condición de igualdad varía dependiendo del algoritmo de clasificación. Para los perceptrones, los puntos en el hiperplano son considerados pertenecientes a la clase negativa

Sea $x^{(i)} \in \mathbb{R}^{(n\times 1)\times 1}$ $\forall i = \{1, 2, ..., m\}$ que representa el m vector de características y $y^{(i)} \in \{0, 1\}$ $\forall i = \{1, 2, ..., m\}$ la correspondiente etiqueta de clase. El algoritmo de aprendizaje del perceptrón es como sigue:
Paso 1: Comienza con un conjunto aleatorio de pesos $w \in \mathbb{R}^{(n\times 1)\times 1}$
Paso 2: Evalúa la clase predicha para los datos. Para un dato de entrada $x^{(i)}$ si $w^T x^{(i)} > 0$, entonces, la clase predicha es $y_p^{(i)} = 1$, de otra forma $y_p^{(i)} = 0$. Para el perceptrón, los puntos en el hiperplano clasifícalos como $y_p^{(i)} = 0$.
Paso 3: Actualiza el vector de pesos w de la siguiente manera:

- Si $y_p^{(i)} = 0$ y la clase actual es $y^{(i)} = 1$, actualiza el vector de pesos como $w = w + x^{(i)}$.
- Si $y_p^{(i)} = 1$ y la clase actual es $y^{(i)} = 0$, actualiza el vector de pesos como $w = w - x^{(i)}$.
- Si $y_p^{(i)} = y^{(i)}$, no se actualiza w.

Paso 4: Ve al paso 2 y procesa el siguiente dato
Paso 5: Termina el algoritmo cuanto todos los datos han sido correctamente clasificados.
El perceptrón solo podrá clasificar las dos clases correctamente si existe un vector de pesos w factible, tal que pueda separar linealmente separada en dos clases, es decir, la regla de aprendizaje del perceptrón solo puede separar las clases si éstas son linealmente separable en el espacio de entrada por lo que se dice que el perceptrón únicamente puede aprender una frontera de decisión lineal para la clasificación. Por ende no puede resolver problemas donde una frontera de decisión no lineal es necesaria. En algunos casos particulares es posible definir más de un hiperplano para separar clases no lineales. Sin embargo, esto no siempre es posible.
Para lidiar con ello se emplea un perceptrón multicapa, que puede proveer una separación no lineal entre clases al introducir la no linealidad en capas ocultas. Para este perceptrón, la regla de aprendizaje anteriormente descrita no es valida en general.

3 Función de activación de las capas ocultas para la no linealidad

Si las funciones de activación para las capas ocultas del perceptrón son lineales, entonces, la salida de la neurona final será lineal y de esta forma no sería posible aprender fronteras de decisión no lineales. De esta forma existen varias funciones de activación para las unidades neuronales y su uso varia con respecto al problema a resolver y la topología de la red neuronal. Las funciones de activación más usadas se listan a continuación.

3.1 Función de activación lineal

En una neurona lineal, la salida es linealmente dependiente de sus entradas. Si la neurona recibe tres entradas x_1, x_2 y x_3, entonces la salida y de la neurona lineal esta dado por $y = w_1x_1 + w_2x_2 + w_3x_3 + b$, donde w_1, w_2 y w_3 son los pesos sinápticos de las entradas x_1, x_2 y x_3 respectivamente y b es el sesgo en la neurona unidad.

3.2 Función de activación de umbral binario

En la neurona de umbral binario, si la entrada total excede un umbral especifico, entonces la neurona es activada, es decir, su salida es 1 y cero en otro caso. Si k es el umbral, entonces la neurona se activa cuando $w^Tx + (b - k) > 0$

3.3 Función de activación sigmoide

La relación entrada salida para una neurona sigmoide se expresa como: $y = 1/(1+exp(-z))$, donde $z = w^Tx + b$ es la entrada. Cuando z es un numero mucho mayor a 1, entonces $exp(-z) =\approx 1$, por otro lado, si es un numero mucho menor a -1 tenemos que $exp(-z) =\approx 0$, Si $z = 0$, entonces $y = 1/2$.
La función sigmoide se utiliza generalmente para producir probabilidad con respecto a una clase

determinada para una clasificación binaria. Las funciones de activación del sigmoide en las capas ocultas introducen la no linealidad para que el modelo pueda aprender características más complejas.

3.4 Función de activación softMax

La función de activación softMax es una generalización de la función sigmoide y es más adecuada para clasificación multiclase. Si hay k clases de salida y el vector de pesos para la i-ésima clase es $w^{(i)}$, entonces la probabilidad predicha para la i-ésima clase dado un vector de entrada $x \in \mathbb{R}^{(n\times 1)}$ está dada por:

$$P(y_i = 1/x) = \frac{exp(w^{(i)T}x + b^{(i)})}{\sum_{j=1}^{k} exp(w^{(j)T}x + b^{(j)})} \tag{4}$$

La función de perdida para la capa softMax se conoce como entropía cruzada categórica y esta dada por:

$$C = \sum_{i=1}^{k} -y_i log P(y_i = 1/x) \tag{5}$$

3.5 Función de activación Rectified Linear Unit (ReLU)

En la función de activación ReLU, la salida es igual a la entrada de la neurona si la entrada total es mayor que cero. Si este valor es menor o igual que cero, entonces, la salida de la neurona es cero. La salida se expresa como $y = max\left(0, w^T x + b\right)$.
La función ReLU es una de los elementos estrella que revolucionó el aprendizaje profundo y es fácil de calcular. Esta función combina lo mejor de ambos mundos: El valor de su gradiente es constante mientras la entrada es positiva y tiene valor cero en otro caso. Que el gradiente sea constante asegura que el algoritmo del descenso del gradiente no para de aprender debido a la desaparición del gradiente. Al mismo tiempo, la salida cero permite la no linealidad.
Existen varias versiones de la función de activación ReLU. Para el modelo que no tiene gradiente cero aun cuando la entrada es negativa, la función parametrizada del ReLU llamada PReLU puede ser de gran utilidad. La relación entrada salida para PReLU es: $y = max(0, z) + \beta min(0, z)$. Donde $z = w^T x + b$ y β es el parámetro aprendido por el entrenamiento.
Cuando $\beta = -1$, entonces, $y = |z|$ y la función de activación se conoce como valor absoluto ReLU. Cuando β es un número pequeño positivo, entonces se conoce como ReLu débil (leaky).

3.6 Función de activación tangente hiperbólica

La relación de entrada-salida para la funcion de activación tangente hiperbolica viene dada por: $y = \frac{e^z - e^{-z}}{e^z + e^{-z}}$. Donde nuevamente $z = w^T x + b$. Cuando z es un número positivo grande, entonces $y \approx 1$. Cuando z es un número muy negativo, $y \approx -1$. Y cuando $z = 0$, $y = 0$. Esta función de activación se satura al rededor del 0. Mientras se entrena una red, si la entrada en la capa es cercano a cero el gradiente desaparece y el entrenamiento se detiene.

4 Regla de Aprendizaje para una red de perceptrones multicapa

La regla de aprendizaje del perceptrón, definida anteriormente, consiste en mantener actualizados los pesos del modelo hasta que todos los datos de entrenamiento hayan sido correctamente clasificados. Si no existe un vector de pesos factible que clasifique todos los puntos correctamente, el

algoritmo no converge. En estos casos, el algoritmo puede ser detenido por un número predefinido de iteraciones o al definir un umbral de muestras clasificadas correctamente.
Para los perceptrones multicapa y la mayoría de las redes de aprendizaje profundo, la mejor manera de entrenar un modelo es calcular una función de costo basada en el error de clasificación errónea y posteriormente minimizar esta función de costo con respecto a los parámetros del modelo. Dado que los algoritmos de aprendizaje basados en el costo minimizan la función de costo, para clasificadores binarios se emplea la función de costo como el negativo de la log-verosimilitud.
Una red de perceptrones de varias capas tendría capas ocultas, y para aprender las fronteras de decisión no lineales, las funciones de activación deberían ser ellas mismas no lineales, como sigmoide, ReLu, tanh, etc. La neurona de salida para la clasificación binaria debería tener una función de activación sigmoide para atender a la función de costo log-perdida y para producir valores de probabilidad para las clases.

5 Optimizadores

Los optimizadores sirven para optimizar las funciones de costo. La mayoría de estos optimizadores en el aprendizaje de maquina están basados en el algoritmo del descenso del gradiente del tal manera están hechos para manejar los problemas de puntos mínimos locales.

5.1 Optimizador Adagrad

El radio de aprendizaje es normalizado para cada dimensión para la cual la función de costo es dependiente. El radio de aprendizaje en cada iteración es es el aprendizaje global dividido por la norma euclidiana de los gradientes anteriores hasta la iteración actual para cada dimensión.
Si se tiene una funcion de costo $C(\theta)$ donde $\theta = (\theta_1, \theta_2, ..., \theta_n) \in \mathbb{R}^{n\times 1}$, entonces la regla de actualización es como sigue:

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \frac{\eta}{\sqrt{\sum_{\tau=1}^{t} \theta_i^{(\tau)2} + \epsilon}} \frac{\partial C^{(t)}}{\partial \theta_i}$$

Donde η es el radio de aprendizaje, $\theta_i^{(t)}$ y $\theta_i^{(t+1)}$ son los valores del i-ésimo parámetro a las iteraciones (t) y $(t+1)$. Este optimizador es bueno para aplicaciones del procesamiento del lenguaje natural y procesamiento de imágenes donde los datos son dispersos.

5.2 RMSprop

es la versión mini-batch del algoritmo resilient backpropagation optimization (Rprop). Este último resuelve el problema de que el gradiente no apunta hacia la dirección mínima en casos donde los contornos de la función de costo son elípticas, en estos casos se emplea una regla adaptativa de actualización para cada peso en lugar de usar aprendizaje global. Rprop no usa la magnitud de los gradientes del peso si no que únicamente los signos para determinar como actualizar cada peso.

5.3 Adadelta

Es una variante del Adagrad que es menos agresivo en cuanto a la reducción del radio de aprendizaje. Para cada conexión de peso, Adagrad escala la constante de la tasa de aprendizaje en una iteración dividiéndola por el cuadrado medio de la raíz de todos los gradientes pasados para ese peso hasta esa iteración. Así, la tasa efectiva de aprendizaje para cada peso es una función monótona decreciente

del número de iteraciones, y después de un número considerable de iteraciones la tasa de aprendizaje se vuelve infinitesimalmente pequeña. Adagrad supera este problema tomando la media de los gradientes cuadrados exponencialmente decadentes para cada peso o dimensión. Por lo tanto, la tasa de aprendizaje efectiva en Adadelta sigue siendo más bien una estimación local de sus gradientes actuales y no se reduce tan rápido como el método de Adagrad. Esto asegura que el aprendizaje continúa incluso después de un número considerable de iteraciones o épocas.

5.4 Adam

Conocido tambien como Adaptive Moment Estimator es otra técnica de optimización que, al igual que el RMSprop o el Adagrad, tiene una tasa de aprendizaje adaptable para cada parámetro o peso. Adagrad no sólo mantiene una media de gradientes cuadrados, sino que también mantiene una media de gradientes pasados.

5.5 Momentum y algoritmo de Nesterov

Los métodos basados en el impulso introducen un componente llamado velocidad v que amortigua la actualización del parámetro cuando el gradiente calculado cambia de signo, mientras que acelera la actualización del parámetro cuando el gradiente está en la misma dirección de la velocidad. Esto introduce una convergencia más rápida así como menos oscilaciones alrededor de los mínimos globales, o alrededor de un mínimo local que proporciona una buena generalización.
En los métodos normales de descenso por gradiente que no tienen en cuenta el impulso, la actualización de los parámetros se detendría en un mínimo local o en el punto de asiento. Sin embargo, en la optimización basada en el impulso, la velocidad previa sacaría al algoritmo de los mínimos locales, considerando que los mínimos locales tienen una pequeña cuenca de atracción, ya que $v_i^{(t+1)}$ sería distinta de cero debido a la velocidad distinta de cero de los gradientes anteriores. Además, si los gradientes anteriores apuntaran consistentemente hacia un mínimo o un mínimo local con una buena generalización y una cuenca de atracción razonablemente grande, la velocidad o el momento de descenso del gradiente sería en esa dirección. Así que, incluso si hubiera un mal mínimo local con una pequeña cuenca de atracción, el componente del momento no sólo impulsaría el algoritmo de los malos mínimos locales, pero también continuaría el descenso del gradiente hacia los mínimos globales.
Una variante específica de los optimizadores basados en el momento es la técnica de gradiente acelerado de Nesterov. Este método utiliza la velocidad existente $v^{(t)}$ para hacer una actualización del vector del parámetro, donde el nuevo vector del parámetro es la suma del vector del parámetro en la iteración anterior y la nueva velocidad.

6 Keras

6.1 Funciones de activación

Las activaciones en keras pueden ser usadas a través de la función Activation o al pasar el argumento activation. Para el primer caso tenemos:

```
from keras.layers import Activation, Dense
model.add(Dense(64))
model.add(Activation('funcion de activacion'))
```

Listing 1: Activaciones en Keras

Que es equivalente a:

```
from keras.layers import Activation, Dense
model.add((Dense(64),activation='funcion de activacion'))
```

Listing 2: Activaciones en keras

Las funciones de activación disponibles en keras son los siguientes:

- elu: Versión modificada del ReLU. Si $x > 0$ regresa x y $\alpha exp(x) - 1$ en otro caso. α es un escalar llamado pendiente de sección negativa. Una explicación muy completa de esta función de activación puede encontrarse en [2].
- softmax: función SoftMax
- selu: es una modificación del elu dada por $escala * elu(x, \alpha)$, donde α y $escala$ se escogen de tal forma que la media y la varianza de las entradas son preservadas entre dos capas consecutivas. La explicación de esta función de activación puede encontrarse en [3].
- softplus: la salida de esta función esta dada por $y = log(exp(x) + 1)$.
- softsign: la salida de esta función esta dada por $y = \frac{x}{|x|+1}$.
- relu: función ReLU
- tanh: tangente hiperbólica
- sigmoid : función sigmoide
- hard_sigmoid: definida como $y = 0$ si $x < -2.5$, $y = 1$ si $x > 2.5$ y $y = 0.2 * x + 0.5$ en otro caso.
- exponential: función exponencial
- linear: regresa la entrada x, es decir no la modifica.

6.2 Funciones de costo o perdida

Las funciones de costo es uno de los parámetros necesarios para compilar un modelo y se inicializa de la siguiente forma

```
from keras.layers import losses
model.compile(loss='funcion de activacion',optimizer='optimizador')
```

Listing 3: Funciones de costo en Keras

Las funciones de costo disponibles en keras son:

- mean_squared_error: error cuadrático medio dado por $\frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$
- mean_absolute_error: error medio absoluto, dado por $\frac{1}{n}\sum_{i=1}^{n}|Y_i - \hat{Y}_i|$
- mean_absolute_percentage_error: error medio porcentual absoluto, definido como $\frac{1}{n}\sum_{i=1}^{n}|\frac{Y_i - \hat{Y}_i}{Y_i}|$
- mean_squared_logarithmic_error: error cuadrático medio logarítmico, dado por $\frac{1}{n}\sum_{i=0}^{n}(\log(Y_i + 1) - \log(\hat{Y}_i + 1))^2$
- squared_hinge: 'bisagra cuadrada' definida como $\sum_{i=0}^{n}\left(max^2(0, 1 - Y_i * \hat{Y}_i)\right)$

- hinge: definida como $\sum_{i=0}^{n} \left(max(0, 1 - Y_i * \hat{Y}_i) \right)$
- logcosh: funcion logaritmo coseno hiperbólico. Dado por: $\sum_{i=0}^{n} log(cosh(\hat{Y}_i - Y_i))$
- huber_loss: Perdida de Huber definida como $(1/2)(Y-\hat{Y})^2$ si $|Y-\hat{Y}| \leq \delta$ y $\delta * |Y-\hat{Y}| - (1/2) * \delta^2$ en otro caso. Donde δ se define por el usuario.
- categorical_crossentropy: entropía cruzada categórica definida como $-\sum_{j=0}^{m} \sum_{i=0}^{n} (Y_{ij} log(\hat{Y}_{ij}))$
- poisson: perdida de poisson definida por $\frac{1}{n} \sum_{i=0}^{n} (\hat{Y}_i - Y_i log(\hat{Y}_i))$

Para todos los casos $\hat{Y}$ es el valor predicho y Y es el valor real. Y cada función de perdida depende del problema que se esta tratando.

6.3 Optimizadores

Los optimizadores son otro parámetro requerido para compilar un modelo en keras. Sigue el mismo script que las funciones de perdida. Los optimizadores disponibles en keras son:

- SGD: descenso del gradiente estocástico
- RMSprop
- Adagrad
- Adadelta
- Adam
- Adamax: es una variante del Adam basado en la norma $|| \cdot ||_\infty$
- Nadam: método basado en el algoritmo de Nesterov

6.4 Métricas

Una métrica es una función que se utiliza para juzgar el rendimiento de un modelo. Una función métrica es similar a una función de pérdida, excepto que los resultados de la evaluación de una métrica no se utilizan al entrenar el modelo. Es posible usar cualquiera de las funciones de pérdida como una función métrica.
Según la documentación de Keras las funciones de métricas se deben suministrar en el parámetro de métricas cuando se compila un modelo. De esta forma se inicializan así:

```
from keras.layers import metrics
model.compile(loss='funcion de activacion',optimizer='optimizador',metrics=['
    metrica 1', ..., 'metrica n'])
```

Listing 4: Meticas en Keras

Las métricas disponibles en keras son:

- accuracy
- binary_accuracy
- categorical_accuracy

- sparse_categorical_accuracy
- top_k_categorical_accuracy
- sparse_top_k_categorical_accuracy
- cosine_proximity

Claramente cada una de estas dependerá del modelo y los datos en cuestión.

References

[1] Pattanayak, S. 2017. **Pro Deep Learning with TensorFlow**. Apress

[2] Clevert, D., Unterthiner, T. & Hochreiter, S. (2016). **Fast and Accurate Deep Network Learning by exponential Linear Units (ELUs)**. arXiv:1511.07289v5

[3] Klambauer, G., Unterthiner, T., Mayr, A. & Hochreiter, S. (2017). **Self-Normalizing Neural Networks**. arXiv:1706.02515