Guía de estudio - Redes Neuronales (parte II)

 ¡Hola! Te damos la bienvenida a esta nueva guía de estudio.

¿En qué consiste esta guía?

Luego de sumergirnos en las redes neuronales feed forward monocapa por medio del perceptrón y su capacidad para procesar información de manera lineal, es hora de adentrarnos en las redes multicapa, cuya profundidad y complejidad nos permiten aproximar funciones más complejas.

Estas redes, dotadas de múltiples capas ocultas, presentan un desafío en su entrenamiento, donde el algoritmo **Backpropagation** es una herramienta esencial que permite calcular las derivadas parciales en forma eficiente usando la conocida **regla de la cadena**, y logrando que la red consiga aprender aproximando cada vez los pesos asociados a cada arista de cada neurona.

Sin embargo, el incremento en la complejidad trae consigo el riesgo de sobreajuste. Aquí es donde acudimos a las estrategias de regularización como la norma L1 y L2 que penalizan los parámetros de la red, controlando su magnitud y previniendo que algunos de estos tenga excesivo protagonismo en el modelo final. Además de L1 y L2, veremos una técnica de regularización usada actualmente llamada **Dropout**, que aleatoriamente desactiva neuronas durante el entrenamiento con lo que fomenta la robustez y generalización de la red.

La búsqueda por grilla se convierte en un aliado poderoso para encontrar la combinación óptima de hiper parámetros, con un costo no menor de tiempo empleado. En esta guía, explicaremos la implementación práctica de estas estrategias, desde el Backpropagation hasta la regularización y la búsqueda por grilla, utilizando las librerías de Keras y Scikit-learn. Esperamos proporcionar una hoja de ruta completa para la comprensión y aplicación de redes neuronales Feed Forward.

¡Vamos con todo!

Tabla de contenidos

¡Comencemos!

Redes neuronales multicapa

Las redes neuronales feedforward multicapa son un pilar en el campo del aprendizaje profundo. Conformadas por múltiples capas, estas redes pueden capturar relaciones complejas en los datos. Pueden ser usadas para resolver problemas complejos en visión artificial, procesamiento de lenguaje natural y otros.

Pasos en el entrenamiento en redes feedforward

Para explicar el proceso de entrenamiento de una red feedforward multicapa usaremos la arquitectura de la Figura 1.

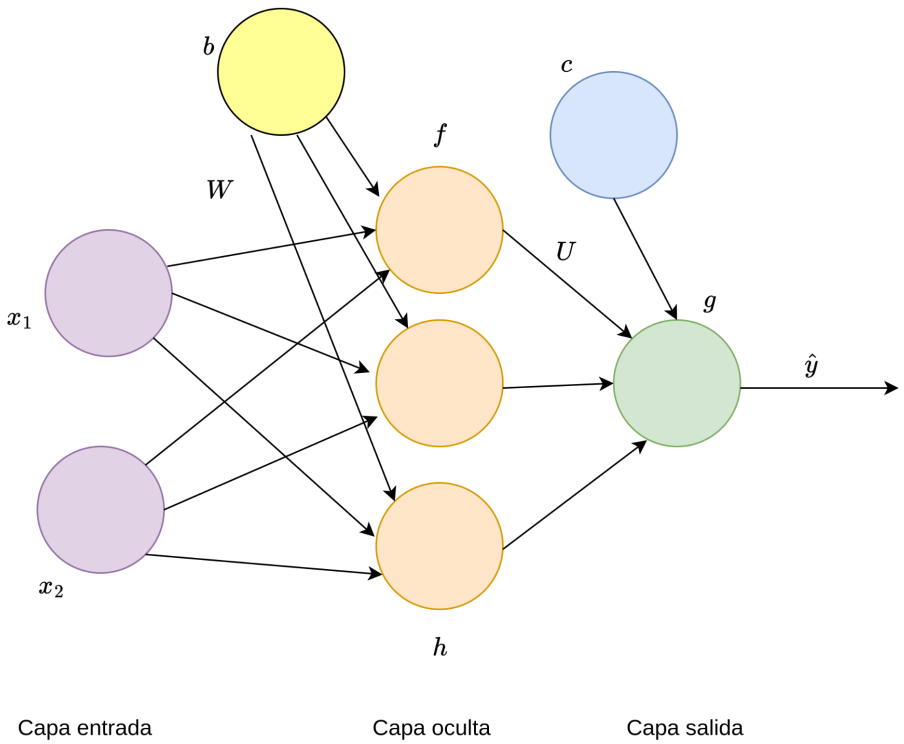

Figura 1. Red Neuronal FeedForward con una capa oculta
Fuente: Desafío Latam

Las redes feedforward son entrenadas en dos grandes pasos:

1. Forward: este paso corresponde al cálculo de la suma ponderada de los pesos por las entradas, aplicando a este resultado una función no lineal que llamamos **función de activación**. Esto se realiza en cada neurona de la red, comenzando desde las entradas de los datos hasta llegar a la última capa de salida.

$$\mathbf{h} = f(\mathbf{XW} + \mathbf{b})$$
$$\hat{y} = g(\mathbf{hU} + \mathbf{c})$$

Una vez hemos calculado las salidas de cada capa oculta y capa de salida, vamos al siguiente paso.

2. Backward: en este paso se recorre la red en dirección contraria, para ir actualizando los parámetros con el fin de minimizar la función de pérdida. Para esto se usa el algoritmo de descenso del gradiente.

   Recordemos que debemos contrastar en nuestra red los valores actuales de los parámetros con el valor esperado para los ejemplos que hemos pasado por la red, para lo que utilizamos la función de pérdida que se haya definido. En nuestro ejemplo los parámetros a estimar son **W, b, U** y **c**, y buscamos minimizar la función de pérdida.

$$\theta = (\mathbf{W}, \mathbf{b}, \mathbf{U}, \mathbf{c})$$
$$\mathcal{L}(\theta) = \arg\min_{\theta} \mathcal{L}(\theta)$$

Debemos medir cuánto influye cambiar "*un poco*" el valor de un parámetro en la función de pérdida, de forma de movernos cada vez en la dirección contraria al máximo crecimiento dado por el gradiente de esta función.

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta \mathcal{L}(x^{(i)}, y^{(i)}, \theta_t)$$

Para calcular estos gradientes se recurre a la regla de la cadena:

$$f = u(w) \qquad \frac{\partial f}{\partial x} = \frac{\partial u}{\partial w}\frac{\partial w}{\partial x}$$
$$w = g(x)$$

Una vez calculadas todas las derivadas parciales podemos actualizar los nuevos valores para los pesos.

Todo este proceso, ya sea la etapa Forward como la Backward, se realiza pasando cada vez una cantidad definida de observaciones (batch).

El término **época** se refiere a cuando hemos pasado por la red todos los ejemplos y actualizado más de una vez (excepto con el algoritmo clásico de descenso de gradiente) los parámetros.

Backpropagation

En la etapa de backward, para calcular las múltiples derivadas parciales se emplea un algoritmo eficiente que nos ayudará en esto y se trata de **Backpropagation,** el que de acuerdo a las dependencias de las neuronas de la red mantiene derivadas parciales calculadas, para ser empleadas en nodos o neuronas que necesiten ese cálculo siguiendo una estrategia conocida como programación dinámica.

En nuestro ejemplo Figura 1, el cálculo de las derivadas parciales para la capa de salida es:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{U}} = \frac{\partial \mathcal{L}}{\partial \mathbf{Z}} \frac{\partial \mathbf{Z}}{\partial \mathbf{U}} \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{c}} = \frac{\partial \mathcal{L}}{\partial \mathbf{Z}} \frac{\partial \mathbf{Z}}{\partial \mathbf{c}}$$

Z representa la suma ponderada de los pesos representados por U por las salidas de la capa oculta, y c corresponde al sesgo de la capa de salida.

$$\frac{\partial \mathcal{L}}{\partial \mathbf{U}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \mathbf{Z}} \frac{\partial \mathbf{Z}}{\partial \mathbf{U}} \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{c}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \mathbf{Z}} \frac{\partial \mathbf{Z}}{\partial \mathbf{c}}$$

Las derivadas parciales para la capa oculta es:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \mathbf{Z}} \frac{\partial \mathbf{Z}}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial u} \frac{\partial u}{\partial \mathbf{W}} \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{b}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \mathbf{Z}} \frac{\partial \mathbf{Z}}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial u} \frac{\partial u}{\partial \mathbf{b}}$$

En la práctica no tendremos que realizar este cálculo en forma manual o tener que implementarlo ya que contamos con librerías optimizadas para el uso intensivo de arquitecturas multiprocesador. Entre las más usadas están Tensorflow y Pytorch.

Funciones de pérdida

Las funciones de pérdida más utilizadas son:

1. Entropía cruzada: esta función mide la discrepancia entre dos distribuciones de probabilidad. En el contexto de aprendizaje automático para clasificación, se usa para evaluar la diferencia entre la distribución de probabilidad real de los datos y la distribución estimada por el modelo. Esta función penaliza las predicciones incorrectas de manera más significativa, cuando las predicciones se encuentran

más alejadas de los valores reales, lo que la hace importante en la optimización de modelos de clasificación. Corresponde a la fórmula

$$\mathrm{CE}(p,q) = \sum_x p(x) log\left(\frac{1}{q(x)}\right)$$

donde p(x) representa la distribución empírica de los datos, y q(x) la distribución de probabilidades entregada por la red.

2. Error cuadrático medio: es una medida comúnmente usada para evaluar la predicción de un modelo en problemas de regresión en redes neuronales feedforward. Esta función calcula el promedio de las diferencias al cuadrado entre las predicciones del modelo y los valores reales. Se calcula en la búsqueda de minimizar estas distancias.

Inicialización de pesos para redes neuronales multicapa

La inicialización de los valores de los pesos de nuestra red neuronal es de vital importancia, ya que si elegimos pesos que están lejos de un óptimo local o en una región desfavorable del espacio de búsqueda, el entrenamiento tendrá una lenta convergencia. Ahora ¿qué valores serán los mejores? Responder esta pregunta considerando que el espacio de búsqueda para esto es infinito no es en absoluto trivial.

Las principales estrategias para inicializar los pesos son:

- **Inicialización aleatoria**: escoger los valores de los pesos de acuerdo a una distribución uniforme o normal.

- **Inicialización de Xavier (Glorot)**: ajustar la escala de inicialización de acuerdo al número de entradas y salida de la capa. Esta inicialización puede ser menos efectiva al usar funciones de activación ReLU. Corresponde a escoger pesos que siguen una distribución normal con media cero y desviación estándar $\frac{\sqrt{2}}{fan\ in + fan\ out}$, donde **fan in** es la cantidad de neuronas de la capa anterior y **fan out** la cantidad de salidas en el tensor de pesos.

- **Inicialización He Normal**: similar a la inicialización de Xavier, pero ajustada para funciones de activación ReLU. Se usa la distribución normal centrada en cero con desviación estándar $\frac{\sqrt{2}}{fan\ in}$.

Para escoger la mejor inicialización de pesos debemos tener en cuenta la arquitectura de la red, las funciones de activación empleadas y el problema a resolver. En términos generales, una inicialización de Xavier es una buena opción para funciones de activación sigmoidales o

de tangente hiperbólica. Para funciones de activación ReLU una buena elección es usar la inicialización He. Se sugiere experimentar con diferentes métodos de inicialización y evaluar su efectividad en términos de convergencia y rendimiento para su elección.

Estandarización de los ejemplos de entrada en redes neuronales multicapa

Con el objetivo de evitar que nuestra red otorgue más importancia a ciertas características, producto de que se encuentren en escalas diferentes, es que comenzaremos por estandarizar los datos de entradas previamente al entrenamiento de la red neuronal.

Elección de funciones de activación para redes neuronales multicapa

La elección de funciones de activación en redes neuronales multicapa es un aspecto fundamental, ya que determina la capacidad del modelo para aprender y representar patrones complejos en los datos. Existen diferentes funciones de activación; entre las más usadas se encuentran:

- **Función sigmoidal**, con valores que van entre 0 y 1: Esta función mapea los valores en un rango limitado. Es una función adecuada para problemas de clasificación binaria en la capa de salida ya que se puede interpretar como probabilidad. Sin embargo, presenta el problema de desvanecimiento del gradiente en capas profundas.

- **Función tangente hiperbólica**, con valores entre -1 y 1: Es similar a la sigmoidal pero con un rango simétrico. Se usa habitualmente en capas ocultas, permitiendo la propagación de valores negativos.

- **Función ReLU**, con un rango de salida de 0 a infinito: Esta función asigna los valores negativos a cero, acelerando el entrenamiento debido a su naturaleza no lineal. Es popular en capas ocultas por su eficiencia computacional y prevención del desvanecimiento del gradiente.

La elección de la función de activación puede depender del problema específico, el conjunto de datos, la estabilidad del entrenamiento y/o la arquitectura de la red. Se recomienda realizar experimentos y evaluaciones comparativas utilizando validación cruzada o conjuntos de validación para determinar qué función de activación se desempeña mejor para una tarea particular. La experimentación y la comprensión de las propiedades de cada función de activación son fundamentales para seleccionar la más adecuada para un problema específico.

Elección de algoritmo de optimización para redes neuronales multicapa

Tenemos diferentes algoritmos de optimización basados en el descenso del gradiente, entre los que se encuentran el Descenso de Gradiente Estocástico (SGD), Adam (Adaptive Moment Estimation), RMSprop (Root Mean Square Propagation), AdaGrad (Adaptive Gradient Algorithm). Para escoger cuál usar consideraremos el tipo de problema, la naturaleza de los datos y la arquitectura de la red.

Adam es una elección popular ya que combina las ventajas de **RMSprop** y **Momentum**. Ahora, si estamos en presencia de un problema con datos dispersos, **AdaGrad** podría ser una opción relevante. Sin embargo, se aconseja experimentar con diferentes algoritmos sintonizando sus hiper parámetros, y con esto poder orientar la decisión respecto de cuál es el más apropiado para el problema específico.

En resumen, en la elección del algoritmo de optimización se debe tener en cuenta la velocidad de convergencia, la adaptabilidad a diferentes escalas de parámetros y qué tan buenos son estos para evitar mínimos locales.

Regularización en redes neuronales feed forward

La regularización en redes neuronales feedforward es fundamental para evitar el sobreajuste y mejorar la capacidad de generalización del modelo. Las más utilizadas son **Lasso - Norma L1, Ridge - Norma L2** y **Dropout.** Cada una de estas aplica un enfoque distinto para controlar la complejidad del modelo y con esto reducir problemas de alta varianza que pueden surgir en redes neuronales multicapas.

Regularización Lasso, Ridge y Elastic-net

Lasso y Ridge introducen términos de penalización en la función de pérdida durante el entrenamiento de la red. Estos términos adicionales se suman a la función de pérdida original y penalizan los parámetros de la red de diferentes formas.

Lasso agrega la suma de los valores absolutos de los pesos multiplicados por un factor “lambda” que debemos sintonizar. Esto impulsa la dispersión de los pesos, y permite la selección automática de características, ya que algunos pesos podrán tener valores iguales a cero y con esto se descartan características de menor importancia.

Por otro lado, la regularización Ridge agrega la suma de los cuadrados de los pesos multiplicados por un factor “lambda”. Una diferencia relevante de Ridge con respecto a Lasso es que no conduce directamente a la selección de características, sin embargo,

penaliza los pesos "grandes" y controla así la complejidad del modelo, evitando que los pesos se vuelvan excesivamente altos.

En una red neuronal feedforward multicapa, la regularización Lasso y Ridge se realiza agregando el término de regularización a la función de pérdida utilizada durante el entrenamiento. Al ajustar los pesos de la red mediante los algoritmos de optimización basados en el descenso del gradiente, se incluyen estas penalizaciones adicionales para actualizar los pesos de forma que se minimice la pérdida total.

En Keras podemos realizar una regularización a nivel de los pesos de cada capa **kernel_regularizer** y/o a nivel de la salida de una capa, es decir, luego de haber aplicado la función de activación **activity_regularizer**.

Regularización Dropout

Dropout es una técnica de regularización que controla el sobreajuste, mediante un proceso que apaga aleatoriamente un porcentaje de neuronas durante el entrenamiento. Esto significa que en cada iteración del entrenamiento, un subconjunto de neuronas se excluye temporalmente de la red, lo que se logra haciendo cero los pesos asociados a las aristas que llegan a la neurona como también las aristas que salen de la neurona en particular que se esté apagando. Esta exclusión aleatoria permite que la red sea más robusta y menos dependiente de características específicas durante el entrenamiento.

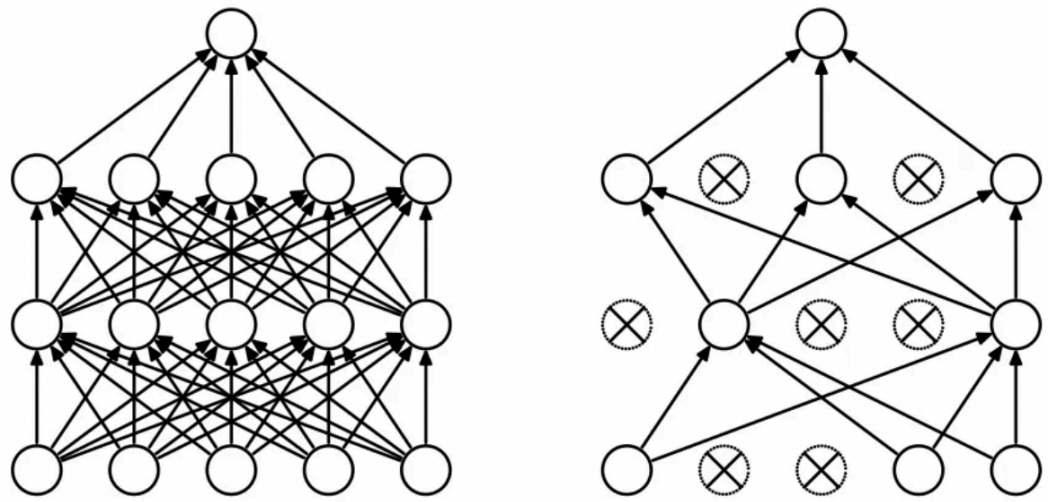

Figura 2. A la izquierda red neuronal sin regularización Dropout y la derecha ejemplo de la red, una vez aplicada una regularización Dropout en una iteración.
Fuente: Desafío Latam.

Regularización usando Keras

Al utilizar la regularización Lasso y Ridge con la librería Keras podemos ir agregando capas de regularización a las capas ocultas o de salida utilizando los parámetros **kernel_regularizer** y/o **bias_regularizer,** al definir la arquitectura de la red. Para la regularización Dropout, podemos agregar capas Dropout entre las capas ocultas, definiendo un porcentaje de neuronas a ser excluidas en forma aleatoria.

En síntesis, la regularización Lasso y Ridge controlan los pesos de la red, mientras que Dropout mejora la generalización al introducir variabilidad en la red. La combinación de estas técnicas ofrece una poderosa forma de evitar el sobreajuste y con esto obtener mejores rendimientos en redes neuronales feedforward.

Destacar que el paper que da origen a la regularización Dropout (Srivastava et al. 2014) sugiere imponer una restricción a los pesos de cada capa oculta, garantizando que la norma máxima de los pesos no supere un valor de tres. Esto se puede realizar en Keras por medio del argumento **kernel_constraint.**

Búsqueda de hiper parámetros óptimos

La búsqueda de hiper parámetros en redes neuronales feedforward corresponde a una estrategia que busca encontrar la configuración óptima para los hiper parámetros que maximizan el rendimiento del modelo. Hemos visto que la librería scikit-learn nos provee de búsqueda de grilla con validación cruzada GridSearchCV, sin embargo, podemos notar que en los modelos de redes neuronales que hemos desarrollado usando Keras se emplea procesamiento paralelo (que en nuestro caso es con Tensorflow) Esto significa que GridSearchCV de scikit-learn no se comunica directamente con Keras. Para resolver esta comunicación empleamos un Wrapper con la librería scikeras que nos permitirá tratar un modelo de red neuronal Keras como si fuese un modelo dentro de scikit-learn.

Para instalar wrapper scikeras debemos ejecutar el comando

```
pip install scikeras
```

Proceso de búsqueda de hiper parámetros

1. Comenzamos el proceso con la arquitectura de un modelo de red neuronal feedforward usando Keras, que debe estar dentro de una función que retorne el modelo.
2. Se crea un modelo usando la clase envoltorio que corresponda, de acuerdo al problema que resolverá la red regresión o clasificación “KeraRegressor” o “KerasClassifier”.

3. Se define la grilla de hiper parámetros a explorar con sus respectivos valores. Entre los hiper parámetros que deseamos sintonizar están: tasa de aprendizaje, el algoritmo de optimización, la cantidad de neuronas por capa, la cantidad de capas, la función de activación a usar en cada capa y los factores de regularización, entre otros.
4. Ejecutamos la búsqueda usando el objeto habitual GridSearchCV, indicando la grilla de búsqueda y la cantidad de fold para la validación cruzada.

Entre las ventajas de sintonizar los hiper parámetros tenemos:

1. **Automatización y Exploración Exhaustiva**: permite movernos por una amplia gama de hiper parámetros de manera sistemática.
2. **Optimización del rendimiento:** con esta búsqueda podremos lograr optimizar la mejor combinación de hiper parámetros mejorando el rendimiento del modelo.

Respecto a las limitaciones, podemos decir que el **costo computacional es muy elevado,** ya que la búsqueda exhaustiva puede ser computacionalmente costosa, especialmente con conjuntos de datos "grandes" y modelos complejos, lo que se traduce en tiempo excesivo en el entrenamiento.

En resumen, si bien la integración de Keras con scikit-learn para la búsqueda de hiper parámetros ofrece un estilo de trabajo habitual a la estructura como se manipulan los objetos scikit-learn, permitiendo mejorar en forma sistemática el rendimiento de un modelo de red neuronal, su utilización nos hace enfrentarnos al excesivo tiempo computacional requerido para encontrar los mejores hiper parámetros. Es importante equilibrar la exploración exhaustiva con el costo computacional para obtener el mejor rendimiento dentro de recursos razonables.

Preguntas de proceso

Reflexiona:

- ¿En qué consiste la regularización Dropout?
- ¿De qué forma puede influir la inicialización de los pesos de una red neuronal multicapa?
- ¿Qué se entiende por época en el contexto de redes neuronales multicapa?
- ¿Cómo se relaciona el backpropagation con el descenso del gradiente?

Referencias bibliográficas

Deep Learning, Ian Goodfellow and Yoshua Bengio and Aaron Courville 2016.

Dive into Deep Learning, Aston Zhang (Author), Zachary C. Lipton (Author), Mu Li (Author), Alexander J. Smola (Author)

Chollet, François. Deep Learning with Python : Manning, 2017.

Patterson, J; Gibson, A. 2017. Deep Learning: a Practitioner's Approach. Ch1: A Review of Machine Learning. Stochastic Gradient Descent. O'Reilly.