# REDES NEURONALES 2

## REDES FUNCIONALES

### RED NEURONAL ANCHA Y PROFUNDA

Un ejemplo de red neuronal no secuencial es la red neuronal "ancha y profunda".

Conecta todas o parte de las entradas directamente a la capa de salida, esta arquitectura posibilita que la red neuronal aprenda dos patrones:

* profundos (usando la ruta profunda) y
* reglas sencillas (a través de la ruta corta).

Así los patrones simples en los datos no acabarán distorsionados por una secuencia profunda de transformaciones.

Diagrama

Descripción generada automáticamente

Vamos a construir una red neuronal ancha y profunda para el problema de las casas de California:

normalization\_layer = tf.keras.layers.Normalization()

hidden\_layer1 = tf.keras.layers.Dense(30, activation="relu")

hidden\_layer2 = tf.keras.layers.Dense(30, activation="relu")

concat\_layer = tf.keras.layers.Concatenate()

output\_layer = tf.keras.layers.Dense(1)

input\_ = tf.keras.layers.Input(shape=X\_train.shape[1:]) # shape=8

normalized = normalization\_layer(input\_)

hidden1 = hidden\_layer1(normalized)

hidden2 = hidden\_layer2(hidden1)

concat = concat\_layer([normalized, hidden2])

output = output\_layer(concat)

model = tf.keras.Model(inputs=[input\_], outputs=[output])

Las primeras cinco líneas crean todas las capas que necesitamos para construir el modelo, las siguientes seis líneas usan estas capas como funciones para ir desde la entrada a la salida y la última línea crea un objeto Model de Keras señalando a la entrada y la salida.

Vamos a analizar este código con más detalle:

* Primero, creamos cinco capas:
  + una capa Normalization para estandarizar las entradas
  + dos capas Dense con 30 neuronas cada una, usando la función de activación ReLU,
  + una capa Concatenate
  + una capa Dense con una sola neurona para la capa de salida, sin función de activación.
* Después, creamos un objeto Input (el nombre de variable input\_ se usa para evitar eclipsar la función integrada de Python input()). Se trata de una especificación del tipo de entrada que recibirá el modelo, incluyendo su shape y, de manera opcional, su dtype, que por defecto es flotante de 32 bits.
* A continuación, usamos la capa Normalization como una función, pasándole el objeto Input.
* Del mismo modo, después pasamos normalized a hidden\_layer1, que genera como salida hidden1, y pasamos hidden1 a hidden\_layer2, que genera como salida hidden2.
* Hasta ahora, hemos conectado las capas de manera secuencial, pero, después, usamos la capa concat\_layer para concatenar la entrada y la salida de la segunda capa oculta.
* Después, pasamos concat a la output\_layer, lo que nos da la output final.
* Por último, creamos un Model de Keras, especificando la entrada y la salida.

Una vez creado este modelo de Keras, todo es exactamente como antes: compilar el modelo, adaptar la capa Normalization (hay que aplicar adapt a la capa de normalización antes de entrenar el modelo), entrenar el modelo, evaluarlo y hacer predicciones.

optimizer = tf.keras.optimizers.Adam(learning\_rate=1e-3)

model.compile(loss="mse", optimizer=optimizer, metrics=["RootMeanSquaredError"])

normalization\_layer.adapt(X\_train)

history = model.fit(X\_train, y\_train, epochs=20, validation\_data=(X\_valid, y\_valid))

mse\_test = model.evaluate(X\_test, y\_test)

X\_new = X\_test[:3]

y\_pred = model.predict(X\_new)

### MANEJAR ENTRADAS MULTIPLES

¿Y si queremos enviar un conjunto de las características por la ruta ancha y otro (posiblemente solapado) por la profunda? En este caso, una solución sería usar múltiples entradas.

Diagrama

Descripción generada automáticamente

Por ejemplo, supongamos que queremos enviar cinco características por la ruta ancha (de la 0 a la 4) y seis por la profunda (de la 2 a la 7):

input\_wide = tf.keras.layers.Input(shape=[5]) # características de la 0 a la 4

input\_deep = tf.keras.layers.Input(shape=[6]) # características de la 2 a la 7

norm\_layer\_wide = tf.keras.layers.Normalization()

norm\_layer\_deep = tf.keras.layers.Normalization()

norm\_wide = norm\_layer\_wide(input\_wide)

norm\_deep = norm\_layer\_deep(input\_deep)

hidden1 = tf.keras.layers.Dense(30, activation="relu")(norm\_deep)

hidden2 = tf.keras.layers.Dense(30, activation="relu")(hidden1)

concat = tf.keras.layers.concatenate([norm\_wide, hidden2])

output = tf.keras.layers.Dense(1)(concat)

model = tf.keras.Model(inputs=[input\_wide, input\_deep], outputs=[output])

Hay varias cosas en las que fijarse en este ejemplo, comparado con el anterior:

* Cada capa Dense se crea y se llama en la misma línea, se trata de una práctica común, ya que hace que el código sea más conciso sin perder claridad. Sin embargo, no podemos hacer esto con la capa Normalization, puesto que necesitamos una referencia a la capa para poder llamar a su método adapt() antes de entrenar el modelo.
* Hemos utilizado tf.keras.layers.concatenate(), que crea una capa Concatenate y la llama con las entradas dadas.
* Hemos especificado inputs=[input\_wide, input\_deep] al crear al modelo, puesto que hay dos entradas.

Ahora podemos compilarlo con normalidad, pero al llamar al método fit(), en vez de pasar una sola matriz de entrenamiento X\_train, debemos pasar un par de matrices (X\_train\_wide, X\_train\_deep), una por entrada. Lo mismo ocurre con X\_valid, y con X\_test y X\_new al llamar a evaluate() o predict():

optimizer = tf.keras.optimizers.Adam(learning\_rate=1e-3)

model.compile(loss="mse", optimizer=optimizer, metrics=["RootMeanSquaredError"])

X\_train\_wide, X\_train\_deep = X\_train[:, :5], X\_train[:, 2:]

X\_valid\_wide, X\_valid\_deep = X\_valid[:, :5], X\_valid[:, 2:]

X\_test\_wide, X\_test\_deep = X\_test[:, :5], X\_test[:, 2:]

X\_new\_wide, X\_new\_deep = X\_test\_wide[:3], X\_test\_deep[:3]

norm\_layer\_wide.adapt(X\_train\_wide)

norm\_layer\_deep.adapt(X\_train\_deep)

history = model.fit((X\_train\_wide, X\_train\_deep), y\_train, epochs=20,

validation\_data=((X\_valid\_wide, X\_valid\_deep), y\_valid))

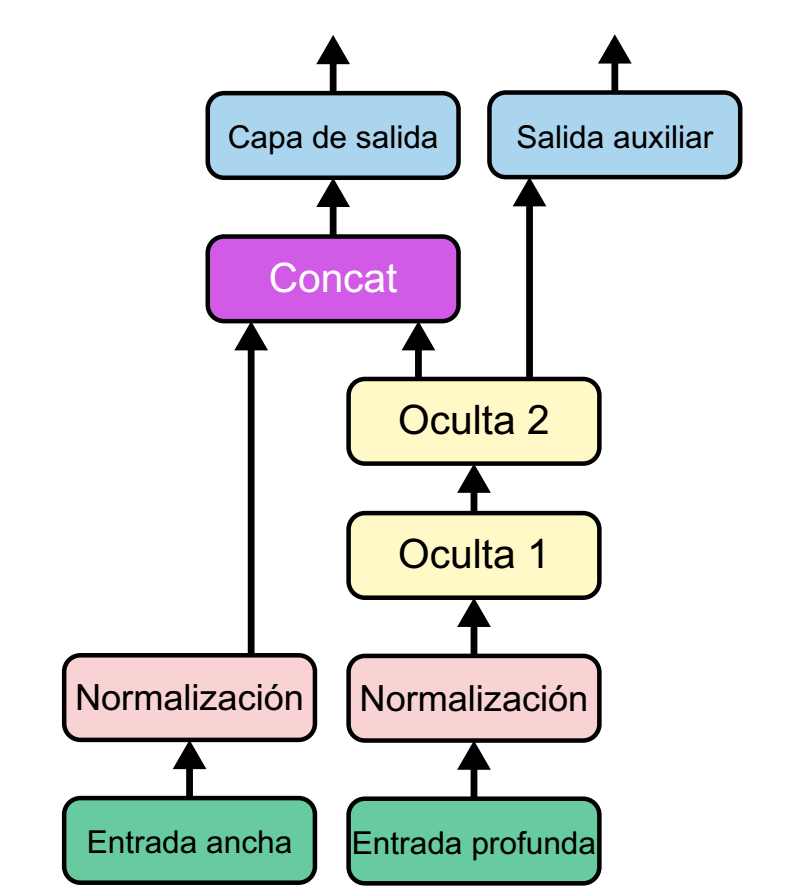
mse\_test = model.evaluate((X\_test\_wide, X\_test\_deep), y\_test)

y\_pred = model.predict((X\_new\_wide, X\_new\_deep))

### MANEJAR SALIDAS MULTIPLES

Hay también varios casos en los que interesa tener múltiples salidas.

Puede que tengamos múltiples tareas independientes basadas en los mismos datos. Desde luego, podríamos entrenar una red neuronal por tarea, pero en muchos casos conseguiremos mejores resultados entrenando una sola red neuronal con una salida por tarea. Esto se debe a que la red neuronal aprende características de los datos que son útiles entre tareas. Por ejemplo, podríamos hacer clasificación multitarea con fotos de caras, usando una salida para clasificar la expresión facial de la persona (sonrisa, sorpresa, etc.) y otra para identificar si llevan gafas o no.



Añadir una salida adicional es bastante fácil: solo tenemos que conectarla a la capa apropiada y añadirla a la lista de salidas del modelo.

input\_wide = tf.keras.layers.Input(shape=[5]) # características de la 0 a la 4

input\_deep = tf.keras.layers.Input(shape=[6]) # características de la 2 a la 7

norm\_layer\_wide = tf.keras.layers.Normalization()

norm\_layer\_deep = tf.keras.layers.Normalization()

norm\_wide = norm\_layer\_wide(input\_wide)

norm\_deep = norm\_layer\_deep(input\_deep)

hidden1 = tf.keras.layers.Dense(30, activation="relu")(norm\_deep)

hidden2 = tf.keras.layers.Dense(30, activation="relu")(hidden1)

concat = tf.keras.layers.concatenate([norm\_wide, hidden2])

output = tf.keras.layers.Dense(1)(concat)

aux\_output = tf.keras.layers.Dense(1)(hidden2)

model = tf.keras.Model(inputs=[input\_wide, input\_deep], outputs=[output, aux\_output])

Cada salida necesitará su propia función de pérdida. Así pues, al compilar el modelo, deberíamos pasar una lista de funciones de pérdida. Si pasamos una sola, Keras asumirá que debe usar la misma para todas las salidas.

Por defecto, Keras computará todas las pérdidas y simplemente las sumará para obtener la pérdida definitiva utilizada para el entrenamiento (pero es posible configurar dar un peso determinado a cada pérdida):

optimizer = tf.keras.optimizers.Adam(learning\_rate=1e-3)

model.compile(loss=("mse", "mse"), loss\_weights=(0.9, 0.1), optimizer=optimizer,

metrics=["RootMeanSquaredError","RootMeanSquaredError"])

Cuando entrenemos el modelo, tenemos que suministrar etiquetas para cada salida. En este ejemplo, la salida principal y la auxiliar deberían intentar predecir lo mismo, así que usaremos las mismas etiquetas. Entonces, en vez de pasar y\_train, tenemos que pasar (y\_train, y\_train) o un diccionario {"output": y\_train, "aux\_output": y\_train} si las salidas se han llamado "output" y "aux\_output". Lo mismo se aplica a y\_valid e y\_test:

n norm\_layer\_wide.adapt(X\_train\_wide)

norm\_layer\_deep.adapt(X\_train\_deep)

history = model.fit(

(X\_train\_wide, X\_train\_deep), (y\_train, y\_train), epochs=20,

validation\_data=((X\_valid\_wide, X\_valid\_deep), (y\_valid, y\_valid))

)

Al evaluar el modelo, Keras devuelve la suma ponderada de las pérdidas, además de todas las pérdidas y métricas individuales:

eval\_results = model.evaluate((X\_test\_wide, X\_test\_deep), (y\_test, y\_test))

weighted\_sum\_of\_losses, main\_loss, aux\_loss, main\_rmse, aux\_rmse = eval\_results

Si configuras return\_dict=True, entonces evaluate() devolverá un diccionario, en vez de una tupla grande.

De modo similar, el método predict() devolverá predicciones para cada salida:

y\_pred\_main, y\_pred\_aux = model.predict((X\_new\_wide, X\_new\_deep))

El método predict() devuelve una tupla y no tiene un argumento return\_dict para obtener un diccionario.

Ver 4\_5\_1Ejemplos redes neuronales funcionales

## PROBLEMA DE DESVANECIMIENTO DE GRADIENTE

El desvanecimiento del gradiente es un problema común en las redes neuronales, especialmente en aquellas con muchas capas (redes neuronales profundas). Ocurre durante el entrenamiento, en el proceso de retropropagación (backpropagation), donde los gradientes de la función de pérdida se utilizan para actualizar los pesos de la red.

En una red profunda, estos gradientes se calculan mediante la regla de la cadena, multiplicando las derivadas parciales a través de las capas. Cuando estas derivadas son pequeñas (menores que 1), la multiplicación repetida de estos pequeños números puede hacer que el gradiente se vuelva extremadamente pequeño. Como resultado, los pesos en las capas iniciales de la red se actualizan muy poco o prácticamente nada, lo que impide que la red aprenda eficazmente. Este fenómeno es lo que se conoce como desvanecimiento del gradiente.

Para evitar este problema existen diferentes soluciones:

1. **Uso de Funciones de Activación Apropiadas**:

La función de activación ReLU (Rectified Linear Unit) y sus variantes (como Leaky ReLU) ayudan a mitigar el desvanecimiento del gradiente, ya que no saturan en la parte positiva, lo que permite que los gradientes fluyan mejor a través de la red.

leaky\_relu = tf.keras.layers.LeakyReLU(alpha=0.2)

dense = tf.keras.layers.Dense(50, activation=leaky\_relu)

model = tf.keras.models.Sequential(

[

# [...] # más capas

tf.keras.layers.Dense(50), # sin activacion

tf.keras.layers.LeakyReLU(alpha=0.2), # activación como una capa separada

# [...] # más capas

]

)

1. **Inicialización Adecuada de Pesos:**

Inicialización de He o Xavier: utilizar técnicas de inicialización de pesos como He o Xavier puede ayudar a mantener los gradientes en un rango adecuado.

dense = tf.keras.layers.Dense(50, activation="relu", kernel\_initializer="he\_normal")

1. **Normalización por Lotes (Batch Normalization):**

La técnica consiste en añadir una operación al modelo justo antes o después de la función de activación de cada capa oculta. Esta operación simplemente centra en cero y normaliza cada entrada.

Implementar la normalización de lotes con Keras es sencillo, simplemente hay que añadir una capa BatchNormalization antes o después de la función de activación de cada capa oculta.

También podemos añadir una capa BN como la primera capa del modelo, pero, por lo general, una simple capa Normalization tiene un rendimiento igual de bueno es esta ubicación (el único inconveniente es que primero debes llamar a su método adapt()).

model = tf.keras.Sequential([

tf.keras.layers.Input(shape=(28, 28)),

tf.keras.layers.Flatten(),

tf.keras.layers.BatchNormalization(),

tf.keras.layers.Dense(300, activation="relu", kernel\_initializer="he\_normal"),

tf.keras.layers.BatchNormalization(),

tf.keras.layers.Dense(100, activation="relu",kernel\_initializer="he\_normal"),

tf.keras.layers.BatchNormalization(),

tf.keras.layers.Dense(10, activation="softmax")

])

model = tf.keras.Sequential([

tf.keras.layers.Input(shape=(28, 28)),

tf.keras.layers.Flatten(),

tf.keras.layers.Dense(300, kernel\_initializer="he\_normal", use\_bias=False),

tf.keras.layers.BatchNormalization(),

tf.keras.layers.Activation("relu"),

tf.keras.layers.Dense(100, kernel\_initializer="he\_normal", use\_bias=False),

tf.keras.layers.BatchNormalization(),

tf.keras.layers.Activation("relu"),

tf.keras.layers.Dense(10, activation="softmax")

])

Ver 4\_5\_2Ejemplos desvanecimiento del gradiente

## REUTILIZAR REDES PREENTRENADAS

Por lo general, no es buena idea entrenar un conjunto muy grande de RNP (redes neuronales profundas) desde cero sin intentar primero encontrar una red neuronal existente que realice una tarea similar a la que estamos intentando abordar. Por lo general puedes reutilizar la mayoría de sus capas, salvo las superiores. Esta técnica se conoce como **aprendizaje por transferencia**. No solo acelera el entrenamiento de forma considerable, sino que, además, requiere bastantes menos datos de entrenamiento.

Supongamos que tenemos acceso a una RNP que se ha entrenado para clasificar imágenes en 100 categorías diferentes, incluyendo animales, plantas, vehículos y objetos cotidianos y, ahora, queremos entrenar una RNP para clasificar tipos específicos de vehículos. Estas tareas son muy similares, incluso se solapan de manera parcial, así que podemos intentar reutilizar partes de la primera red.

Diagrama

Descripción generada automáticamente

Si las imágenes de entrada para la nueva tarea no tienen el mismo tamaño que las utilizadas en la tarea original, por lo general hay que añadir un paso de preprocesamiento para redimensionarlas y darles el tamaño esperado por el modelo original. A nivel más general, el aprendizaje por transferencia funcionará mejor cuando las entradas tengan características de bajo nivel similares.

Normalmente, la capa de salida del modelo original debería sustituirse, porque lo más probable es que no sea útil en absoluto para la nueva tarea y es probable que no tenga el número de salidas adecuado.

De forma similar, las capas ocultas superiores del modelo original tienen menos probabilidades de ser tan útiles como las capas inferiores, puesto que las características de alto nivel que son más útiles para las nuevas tareas pueden diferir significativamente de las que eran más útiles para la tarea original. Conviene encontrar el número adecuado de capas a reutilizar.

Cuanto más similares sean las tareas, más capas te convendrá reutilizar (empezando por las capas inferiores).

Probamos a congelar primero las capas reutilizadas (es decir, hacer que sus pesos sean no entrenables para que el descenso de gradiente no las modifique y permanezcan fijas) y, a continuación, entrenamos el modelo para ver qué rendimiento tiene. Después, probamos a descongelar una o dos de las capas ocultas superiores para permitir que la retropropagación las ajuste y comprobamos si mejora el rendimiento. Cuantos más datos de entrenamiento tengamos, más capas podremos descongelar. También resulta útil reducir la tasa de aprendizaje cuando descongelemos capas reutilizadas. Si aun así no conseguimos un buen rendimiento y tenemos pocos datos de entrenamiento, podemos probar a dejar fuera la capa o capas ocultas superiores y a volver a congelar todas las capas ocultas restantes. Podemos repetir hasta que encontremos el número adecuado de capas a reutilizar. Si tenemos muchos datos de entrenamiento, podemos probar a sustituir las capas ocultas superiores en vez de dejarlas fuera e incluso añadir más capas ocultas.

### APRENDIZAJE POR TRANSFERENCIA CON KERAS

Supongamos que el conjunto de datos Fashion MNIST contiene solo ocho clases; por ejemplo, todas las clases salvo camisetas y jerséis. Alguien ha creado y entrenado un modelo de Keras con ese conjunto y ha obtenido un rendimiento bastante bueno (>90% de exactitud). Vamos a llamar a este modelo **"A".** Ahora queremos abordar una tarea diferente: tenemos imágenes de camisetas y jerséis y queremos entrenar un clasificador binario: positivo para camisetas y negativo para jerséis. Nuestro conjunto de datos es bastante pequeño; solo tiene 200 imágenes etiquetadas. Cuando entrenamos un modelo nuevo para esta tarea (vamos a llamarlo "B") con la misma arquitectura que el modelo A obtienemos un 91,85 % de exactitud.

Mientras tomamos el café por la mañana, nos damos cuenta de que nuestra tarea es bastante similar a la tarea A, así que ¿podría ayudar el aprendizaje por transferencia? ¡Vamos a averiguarlo!

Primero, hay que cargar el modelo A y crear un modelo nuevo basado en sus capas. Decidimos reutilizar todas las capas excepto la capa de salida:

# Suponiendo que el modelo A ya estuviese entrenado y guardado en "my\_model\_A"

model\_A = tf.keras.models.load\_model("Resultados/my\_model\_A.keras")

model\_B\_on\_A = tf.keras.Sequential(model\_A.layers[:-1])

model\_B\_on\_A.add(tf.keras.layers.Dense(1, activation="sigmoid"))

Tenemos que tener en cuenta que model\_A y model\_B\_on\_A comparten ahora algunas capas. Cuando entrenemos model\_B\_on\_A, también afectará a model\_A. Si queremos evitarlo, necesitamos clonar model\_A antes de reutilizar sus capas. Para ello, clonamos la arquitectura del modelo A con clone\_model() y, después, copiamos sus pesos (clone\_model clona la arquitectura, pero no los pesos):

model\_A\_clone = tf.keras.models.clone\_model(model\_A)

model\_A\_clone.set\_weights(model\_A.get\_weights())

model\_B\_on\_A = tf.keras.Sequential(model\_A\_clone.layers[:-1])

model\_B\_on\_A.add(tf.keras.layers.Dense(1, activation="sigmoid"))

Ahora podríamos entrenar model\_B\_on\_A para la tarea B, pero, como la nueva capa de salida se ha inicializado de manera aleatoria, cometerá errores grandes (al menos durante las primeras repeticiones), así que habrá gradientes de error grandes que pueden arruinar los pesos reutilizados. Para evitarlo, una posibilidad es congelar las capas reutilizadas durante las primeras repeticiones, dando a la capa nueva algo de tiempo para que aprenda pesos razonables. Para ello, configuramos el atributo trainable de cada capa como False y compilamos el modelo:

for layer in model\_B\_on\_A.layers[:-1]:

layer.trainable = False

optimizer = tf.keras.optimizers.SGD(learning\_rate=0.001)

model\_B\_on\_A.compile(loss="binary\_crossentropy", optimizer=optimizer, metrics=["accuracy"])

Debemos compilar siempre el modelo después de congelar o descongelar capas.

Ahora podemos entrenar el modelo durante unas pocas repeticiones, descongelar las capas reutilizadas (lo cual requiere volver a compilar el modelo) y seguir entrenando para ajustar las capas reutilizadas para la tarea B. Después de descongelar las capas reutilizadas, suele ser recomendable reducir la tasa de aprendizaje, para evitar, de nuevo, que se dañen los pesos reutilizados:

history = model\_B\_on\_A.fit(X\_train\_B, y\_train\_B, epochs=4, validation\_data=(X\_valid\_B, y\_valid\_B))

for layer in model\_B\_on\_A.layers[:-1]:

layer.trainable = True

optimizer = tf.keras.optimizers.SGD(learning\_rate=0.001)

model\_B\_on\_A.compile(loss="binary\_crossentropy", optimizer=optimizer,metrics=["accuracy"])

history = model\_B\_on\_A.fit(X\_train\_B, y\_train\_B, epochs=16, validation\_data=(X\_valid\_B, y\_valid\_B))

La exactitud de prueba de este modelo es del 92,85%, superior a la del modelo sin usar parte del modelo A:

model\_B\_on\_A.evaluate(X\_test\_B, y\_test\_B)

[0.3008052408695221, 0.9284999966621399]

Realmente, la técnica no es tan buena como parece, se han probado varias configuraciones hasta que se ha encontrado una muestra con una gran mejora. Si cambiamos las clases o la semilla aleatoria, por lo general, la mejora se detiene, incluso desaparece o se invierte.

Esto se debe a que el aprendizaje por transferencia no funciona muy bien con redes densas pequeñas, presumiblemente porque las redes pequeñas aprenden pocos patrones, que tienen pocas probabilidades de ser útiles en otras tareas. El aprendizaje por transferencia funciona mejor con redes convolucionales profundas, que tienden a aprender detectores de características que son mucho más generales (sobre todo en las capas inferiores).

Ver 4\_5\_3Ejemplos redes preentrenadas.ipynb

## EVITAR EL SOBREAJUSTE MEDIANTE LA REGULARIZACIÓN

Las redes neuronales profundas suelen tener decenas de miles de parámetros, a veces incluso millones. Eso les proporciona una gran libertad y significa que pueden ajustar una variedad enorme de conjuntos de datos complejos. Pero esta gran flexibilidad también hace que la red sea propensa al sobreajuste en el conjunto de entrenamiento. A menudo se necesita la regularización para evitarlo.

Existen diferentes indicios que pueden hacer que nos demos cuenta de que está pasando:

1. **Pérdida de entrenamiento y pérdida de validación:** inicialmente, ambas curvas de pérdida (entrenamiento y validación) disminuyen a medida que el modelo aprende. El sobreajuste suele ocurrir cuando la curva de pérdida de entrenamiento continúa disminuyendo, pero la curva de pérdida de validación deja de mejorar y posiblemente comienza a aumentar.
2. **Precisión de entrenamiento y precisión de validación:** la precisión de entrenamiento tiende a aumentar a medida que el modelo se ajusta a los datos de entrenamiento. Sin embargo, la precisión de validación puede dejar de mejorar y posiblemente disminuir a medida que el modelo se sobreajusta.

Ya hemos implementado una de las mejores técnicas de regularización: la detención temprana. Ahora vamos a examinar otras técnicas de regularización populares para las redes neuronales: regularizaciones l1 y l2, dropout y la regularización max-norm.

### REGULARIZACIONES L1 Y L2

Podemos utilizar la regularización l2 para limitar los pesos de conexión de una red neuronal o la regularización l1, si quieres un modelo disperso (con muchos pesos iguales a 0).

Aplicar la regularización **l1** a los pesos implica agregar un término adicional a la función de pérdida del modelo que **penaliza los valores absolutos de los pesos**. La regularización l1 añade la suma absoluta de los valores de los pesos multiplicada por un factor de regularización (también conocido como parámetro de regularización o alpha). Podemos utilizar tf.keras.regularizers.l1() si queremos una regularización l1.

Aplicar la regularización **l2** a los pesos implica agregar un término adicional a la función de pérdida del modelo que **penaliza los pesos grandes**. La regularización L2 añade la suma cuadrada de los valores de los pesos multiplicada por un factor de regularización (también conocido como parámetro de regularización o alpha).

Veamos cómo aplicar la regularización l2 con Keras, utilizando un factor de regularización de 0,01:

layer = tf.keras.layers.Dense(100, activation="relu", kernel\_initializer="he\_normal",

kernel\_regularizer=tf.keras.regularizers.l2(0.01))

La función l2() devuelve un regularizador al que se llamará en cada paso durante el entrenamiento para calcular la pérdida de regularización.

Si quieres tanto l1 como l2 para penalizar los valores absolutos (l1) y los valores al cuadrado (l2) de los pesos, utiliza tf.keras.regularizers.l1\_l2() (especificando ambos factores de regularización).

Puesto que, por lo general, querremos aplicar el mismo regularizador a todas las capas de la red, además de utilizar la misma función de activación y la misma estrategia de inicialización en todas las capas ocultas, tendremos que repetir los mismos argumentos una y otra vez. Esto hace que quede un código feo y propenso a errores.

Para evitar eso, podemos usar bucles o utilizar la función **functools.partial()**, que permite crear una envoltura fina para cualquier elemento al que se llame, con algunos valores de argumento predeterminados:

RegularizedDense = partial(tf.keras.layers.Dense,

activation="relu",

kernel\_initializer="he\_normal",

kernel\_regularizer=tf.keras.regularizers.l2(0.01))

model\_l2 = tf.keras.Sequential([

tf.keras.layers.Flatten(input\_shape=[28, 28]),

RegularizedDense(100),

RegularizedDense(100),

RegularizedDense(10, activation="softmax")

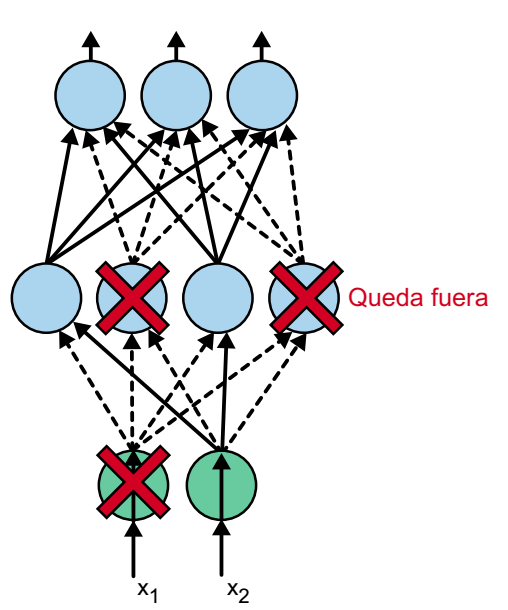
])

### DROPOUT

Dropout es una de las técnicas de regularización más populares para las redes neuronales profundas. Muchas redes neuronales de última generación usan dropout, ya que les proporciona un aumento de 1%-2% en la exactitud. Puede que no parezca gran cosa, pero, cuando un modelo tiene ya una exactitud del 95%, conseguir un aumento de un 2% en la exactitud significa reducir la tasa de error en casi un 40 % (pasa del 5% de error a aproximadamente el 3%).

Se trata de un algoritmo bastante simple: en cada paso, cada neurona (incluyendo las neuronas de entrada, pero siempre excluyendo las neuronas de salida) tiene una probabilidad p de "dejarse fuera" de manera temporal, lo que significa que se ignorará por completo durante este paso de entrenamiento, pero puede que esté activa durante el paso siguiente. El hiperparámetro p se denomina "tasa de dropout" y, por lo general, se configura entre el 10% y el 50%: más cerca del 20%-30% en redes neuronales recurrentes y más cerca del 40%-50% en redes neuronales convolucionales.

Tras el entrenamiento, las neuronas ya no se dejan fuera.



Al principio, resulta sorprendente que esta técnica destructiva funcione. ¿Tendría una empresa mejor rendimiento si se dijese a los empleados que lanzasen una moneda al aire cada mañana para decidir si van a trabajar o no? Bueno, ¿quién sabe?, ¡a lo mejor sí! La empresa se vería obligada a adaptar su organización; no podría depender de una sola persona para manejar la cafetera o realizar cualquier otra tarea crucial, así que esta habilidad tendría que distribuirse por varias personas. Los empleados tendrían que aprender a cooperar con muchos de sus compañeros, no solo con unos pocos. La empresa se volvería mucho más resiliente. Si una persona se marchase, no supondría una gran diferencia. No está claro si esta idea funcionaría de verdad para empresas, pero, desde luego, sí funciona para las redes neuronales. Las neuronas entrenadas con dropout no pueden coadaptarse a sus neuronas vecinas; tienen que ser lo más útiles posible por su cuenta. Tampoco pueden depender en exceso de solo unas pocas neuronas de entrada; deben prestar atención a cada una de sus neuronas de entrada. Acaban siendo menos sensibles a cambios pequeños en las entradas. Al final, obtenemos una red más sólida que generaliza mejor.

En la práctica, por lo general, se aplica dropout solo a las neuronas de una a tres de las capas superiores (excluyendo la capa de salida).

Hay un detalle técnico pequeño, pero importante. Supongamos que p = 75%: de media, solo el 25 % de todas las neuronas están activas en cada paso durante el entrenamiento. Eso significa que, después del entrenamiento, una neurona estaría conectada a cuatro veces más neuronas de entrada de lo que estaría durante el entrenamiento. Para compensar este hecho, necesitamos multiplicar los pesos de conexión de entrada de cada neurona por cuatro durante el entrenamiento. Si no lo hacemos, la red neuronal no tendrá un buen rendimiento, ya que verá datos diferentes durante y después del entrenamiento. A nivel más general, **necesitamos dividir los pesos de conexión entre la probabilidad de mantener cada valor (1 - p) durante el entrenamiento**.

Para implementar dropout usando Keras, podemos utilizar la capa tf.keras.layers.Dropout. **Durante el entrenamiento, deja fuera algunas entradas de manera aleatoria (estableciéndolas como 0) y divide las entradas restantes entre la probabilidad de mantener cada valor. Después del entrenamiento, no hace nada en absoluto; solo pasa las entradas a la siguiente capa**.

El siguiente código aplica la regularización dropout antes de cada capa densa, utilizando una tasa de dropout de 0,2:

model\_dropout = tf.keras.Sequential([

tf.keras.layers.Input(shape=(28, 28)),

tf.keras.layers.Flatten(),

tf.keras.layers.Dropout(rate=0.2),

tf.keras.layers.Dense(100, activation="relu", kernel\_initializer="he\_normal"),

tf.keras.layers.Dropout(rate=0.2),

tf.keras.layers.Dense(100, activation="relu", kernel\_initializer="he\_normal"),

tf.keras.layers.Dropout(rate=0.2),

tf.keras.layers.Dense(10, activation="softmax")

])

Puesto que la regularización dropout solo está activa durante el entrenamiento, comparar la pérdida de entrenamiento y la pérdida de validación puede resultar engañoso. En concreto, un modelo puede estar sobreajustando el conjunto de entrenamiento y, aun así, tener pérdidas de entrenamiento y de validación similares. Por tanto, hay que asegurarse de evaluar la pérdida de entrenamiento sin dropout (por ejemplo. después del entrenamiento).

Si observas que el modelo está sobreajustando, puedes incrementar la tasa de dropout. Por el contrario, si el modelo subajusta el conjunto de entrenamiento, deberías probar a reducir la tasa de dropout. También puede ser de ayuda aumentar la tasa de dropout para capas grandes y reducirla para capas pequeñas.

Dropout tiende a ralentizar la convergencia de forma significativa, pero, a menudo, tiene como resultado un modelo mejor cuando se ajusta de manera adecuada. Por tanto, suele merecer la pena invertir esfuerzo y tiempo extra, sobre todo para modelos grandes.

### REGULARIZACIÓN MAX-NORM

Otra técnica de regularización popular para controlar el crecimiento excesivo de los pesos durante el entrenamiento en las redes neuronales es la regularización max-norm. La idea principal detrás de la max-norm es limitar la norma l2 de los pesos de una capa a un valor máximo predefinido. Es decir, **para cada neurona, restringe los pesos w de las conexiones entrantes como , donde r es el hiperparámetro max-norm y es la norma l2.**

Reducir r incrementa la cantidad de regularización y ayuda a reducir el sobreajuste. La regularización max-norm también ayuda a paliar los problemas de gradientes inestables (si no estás usando la normalización de lotes).

Para implementar la regularización max-norm en Keras, configura el argumento kernel\_constraint de cada capa oculta como una restricción max\_norm() con el valor máximo adecuado:

dense = tf.keras.layers.Dense(

100, activation="relu", kernel\_initializer="he\_normal",

kernel\_constraint=tf.keras.constraints.max\_norm(1.))

Puedes ajustar el valor 1.0 según sea necesario para controlar la magnitud máxima permitida de los pesos. La capa resultante limitará automáticamente la norma l2​ de sus pesos durante el entrenamiento.

Ver: 4\_5\_4Ejemplos regularizacion.ipynb