

Deep Learning

Episodio 3: Observaciones Prácticas para el Aprendizaje

Fernando Gama

Escuela de Graduados en Ingeniería Informática y Sistemas, Facultad de Ingeniería, UBA

21 de Julio de 2022

- ▶ Qué quiere decir aprender \Rightarrow Resolver el problema de minimización del riesgo empírico
 - \Rightarrow Usar los datos porque no contamos con la distribución de los mismos
 - \Rightarrow Problema que persiste: encontrar la mejor función
- ▶ Optimizar sobre el espacio de todas las funciones posibles es prácticamente imposible
 - \Rightarrow Elegir una parametrización de la función de aprendizaje \Rightarrow Redes Neuronales
 - \Rightarrow Las redes neuronales son cascadas de capas \Rightarrow Transformación lineal + activación no lineal
 - \Rightarrow Las redes neuronales tienen hiperparámetros a cargo del diseñador: profundidad, ancho, activación
- ▶ Optimizar sobre parámetros se puede resolver \Rightarrow Pero necesitamos automatizar la optimización
 - \Rightarrow Algoritmo de gradiente (estocástico) descendente
 - \Rightarrow Algoritmo de backpropagation (calcular automáticamente la derivada \Rightarrow regla de la cadena)
 - \Rightarrow Más hiperparámetros para el diseñador: tasa de aprendizaje, tamaño de batch, cantidad de epochs

Funciones de Costo

Regularización

- ▶ La forma de aprender es minimizar el riesgo empírico para una red neuronal

$$\min_{\theta} \frac{1}{M} \sum_{m=1}^M J(f(\mathbf{x}_m; \theta))$$

- ⇒ Dado un conjunto de M muestras de entrenamiento $\{\mathbf{x}_1, \dots, \mathbf{x}_m\}$ (batches, epochs, etc.)
- ⇒ Dada una función de costo $J : \mathbb{R}^O \rightarrow \mathbb{R}$

- ▶ El algoritmo para minimizar el riesgo empírico está basado en el **cálculo del gradiente**
 - ⇒ Necesitamos elegir una **función de costo $J(\mathbf{y})$ que sea derivable** respecto de $\mathbf{y} \in \mathbb{R}^O$

$\frac{\partial J}{\partial \mathbf{y}}$ tiene que existir

- ▶ Consideremos un problema de clasificación \Rightarrow Aprendizaje supervisado
 - \Rightarrow Contamos con un conjunto de muestras de entrenamiento $\{(\mathbf{x}_m, y_m)\}$
 - \Rightarrow El vector $\mathbf{x}_m \in \mathbb{R}^N$ es el dato que queremos clasificar
 - \Rightarrow El escalar $y_m \in \{1, \dots, C\}$ es la clase correspondiente
- ▶ Queremos aprender la función $f : \mathbb{R}^N \rightarrow \mathbb{R}$ que toma el dato y nos devuelve la clase
- ▶ ¿Cuál sería la función de costo en este problema?
 - \Rightarrow Lo más sencillo sería determinar si la clase asignada por f es igual a la clase asociada y

$$J(f(\mathbf{x}; \boldsymbol{\theta})) = J(f(\mathbf{x}; \boldsymbol{\theta}), y) = 1\{f(\mathbf{x}; \boldsymbol{\theta}) \neq y\}$$

\Rightarrow La función $1\{\mathcal{A}\}$ es igual a 1 si \mathcal{A} es verdadero, y 0 en caso contrario

- ▶ La función de costo más fácil de pensar es $J(f(\mathbf{x}; \theta)) = J(f(\mathbf{x}; \theta), y) = 1\{f(\mathbf{x}; \theta) \neq y\}$
⇒ Pero esta función no es derivable ⇒ No podemos usar gradiente descendente
- ▶ Cuando no sabemos algo ⇒ Le imponemos un marco de trabajo basado en la teoría de probabilidades
- ▶ Supongamos que, en vez de aprender la clase y , lo que estamos aprendiendo es la distribución $\mathbf{Y}|\mathbf{X}$
⇒ Aprendemos la probabilidad de que la observación \mathbf{X} pertenezca a la clase \mathbf{Y}
- ▶ Tenemos $\mathbf{Y} \in \{1, \dots, C\}$ clases ⇒ La probabilidad de pertenecer a la clase C es $P(\mathbf{Y} = c) = p_c$
- ▶ Para escribir esto de manera general, asumamos que $\mathbf{Y} \in \{0, 1\}^C$ de manera que

$$P(\mathbf{Y} = \mathbf{y}) = \prod_{c=1}^C p_c^{[\mathbf{y}]_c}$$

donde $[\mathbf{y}]_c = 1$ si la muestra pertenece a la clase c , y 0 en caso contrario

- ▶ En vez de aprender a qué clase pertenece, vamos a aprender la probabilidad de pertenencia a la clase
⇒ La función $\hat{\mathbf{y}} = \mathbf{f}(\mathbf{x}; \boldsymbol{\theta})$ va a ir de \mathbb{R}^N a \mathbb{R}^C , con $\hat{\mathbf{y}} \in [0, 1]^C$ ⇒ $\hat{\mathbf{y}}$ puede tomar valores entre 0 y 1
- ▶ Entonces, tenemos $\mathbf{y} \in \{0, 1\}^C$ que es lo que observamos (aprendizaje supervisado)
- ▶ Y tenemos $\hat{\mathbf{y}} \in [0, 1]^C$ que es la probabilidad de que \mathbf{x} pertenezca a cada clase
- ▶ Queremos, entonces, aprender $\hat{\mathbf{y}}$ de manera tal que las observaciones \mathbf{y} sean altamente probables
⇒ Queremos, entonces, maximizar la verosimilitud
- ▶ Recordemos que la verosimilitud L viene dada por la probabilidad de haber observado lo que observé

$$L(\mathbf{y}; \mathbf{x}, \boldsymbol{\theta}) = \prod_{c=1}^C p_c^{[\mathbf{y}]_c} = \prod_{c=1}^C [\hat{\mathbf{y}}]_c^{[\mathbf{y}]_c} = \prod_{c=1}^C [\mathbf{f}(\mathbf{x}; \boldsymbol{\theta})]_c^{[\mathbf{y}]_c}$$

donde $[\mathbf{y}]_c = 1$ si observamos la clase c y cero en otro caso

donde interpretamos el elemento c de la salida de la red neuronal $[\mathbf{f}(\mathbf{x}; \boldsymbol{\theta})]_c = [\hat{\mathbf{y}}]_c$ como p_c

- ▶ Queremos aprender $\mathbf{f}(\mathbf{x}; \boldsymbol{\theta})$ de manera que refleje de la mejor manera las observaciones \mathbf{y}

- ▶ Queremos aprender $f(\mathbf{x}; \boldsymbol{\theta})$ de manera que refleje de la mejor manera las observaciones \mathbf{y}
- ▶ Queremos maximizar la función de verosimilitud $L(\mathbf{y}; \mathbf{x}, \boldsymbol{\theta})$

$$L(\mathbf{y}; \mathbf{x}, \boldsymbol{\theta}) = \prod_{c=1}^C [\hat{\mathbf{y}}]_c^{[\mathbf{y}]_c} = \prod_{c=1}^C [f(\mathbf{x}; \boldsymbol{\theta})]_c^{[\mathbf{y}]_c}$$

- ▶ Observemos que, respecto de $\boldsymbol{\theta}$, la verosimilitud $L(\mathbf{y}; \mathbf{x}, \boldsymbol{\theta})$ es continua y derivable en $[0, 1]^C$
⇒ Podemos tomar gradientes para optimizar (maximizar L es equivalente a minimizar $-L$)
- ▶ El problema es que la derivada de un producto es muy molesta de calcular
⇒ Aplicamos logaritmos para convertir el producto en suma
⇒ El logaritmo es una función estrictamente creciente ⇒ No afecta la ubicación del óptimo

- ▶ Optimizamos el logaritmo de la función de verosimilitud $\ln(L(\mathbf{y}; \mathbf{x}, \boldsymbol{\theta}))$ para evitar productos

$$\ln(L(\mathbf{y}; \mathbf{x}, \boldsymbol{\theta})) = \sum_{c=1}^C [\mathbf{y}]_c \ln ([f(\mathbf{x}; \boldsymbol{\theta})]_c)$$

- ▶ Esta función es derivable \Rightarrow La adoptamos para minimizar el riesgo empírico

$$\hat{R}(\boldsymbol{\theta}) = -\frac{1}{M} \sum_{m=1}^M \sum_{c=1}^C [\mathbf{y}_m]_c \ln ([f(\mathbf{x}_m; \boldsymbol{\theta})]_c)$$

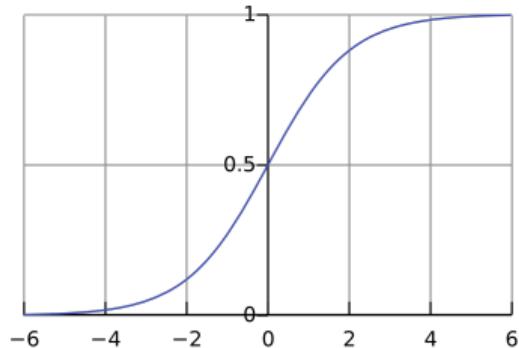
\Rightarrow Observar que, para cada muestra (\mathbf{x}_m, y_m) , la suma sobre c tiene un único sumando

\Rightarrow Si $y_m = c'$, entonces $\sum_{c=1}^C [\mathbf{y}_m]_c \ln ([f(\mathbf{x}_m; \boldsymbol{\theta})]_c) = \ln ([f(\mathbf{x}_m; \boldsymbol{\theta})]_{c'})$ porque $[\mathbf{y}_m]_{c'} = 1$ y 0 en otro caso

- ▶ La función que queremos evaluar (costo) no siempre es la misma función que optimizamos (pérdida)
- ▶ Función de **pérdida** ⇒ Se utiliza para **minimizar el riesgo empírico**
 - ⇒ Debe ser **derivable** respecto de θ para poder usar gradiente descendente
 - ⇒ Se calcula sobre las **muestras de entrenamiento**
- ▶ Función de **costo** ⇒ Se utiliza para **evaluar el desempeño** del algoritmo
 - ⇒ No tiene por qué ser derivable
 - ⇒ Se calcula sobre las **muestras de evaluación**
- ▶ Es indudable que minimizar la función de **pérdida**, debiera decrecer la función de **costo**

- ▶ Se puede explotar el desacople entre la función de costo y la de pérdida
 - ⇒ Elegir funciones de pérdida con características favorables para el aprendizaje
- ▶ Si el **gradiente de la función es grande** se traduce en mucha información para el aprendizaje
 - ⇒ Es preferible evitar funciones que saturen ⇒ No-linealidades de la última capa
- ▶ Las normas ℓ_2 y ℓ_1 no suelen trasladar bien los gradientes
- ▶ Es habitual asumir un modelo para los datos $f_{Y|X}(y|x)$ (requiere conocimiento de expertos)
 - ⇒ Se minimiza el logaritmo de la verosimilitud independientemente del costo

- ▶ Volvamos al **problema de clasificación** \Rightarrow Introdujimos la **verosimilitud**
- ▶ Convertimos la **supervisión** $y_m \in \{1, \dots, C\}$ escalar en un vector $\mathbf{y}_m \in \{0, 1\}^C$ (one-hot vector)
- ▶ Esto implica que tenemos que **adaptar la salida de la red neuronal** a ser un vector
 - \Rightarrow Es más, tiene que ser un vector de probabilidades $f(\mathbf{x}; \boldsymbol{\theta}) = \hat{\mathbf{y}} \in [0, 1]^C$
- ▶ Necesitamos **garantizar que la salida de la red neuronal sea un vector** $[0, 1]^C$
 - \Rightarrow Usamos la función de activación de la **última capa** para forzar la salida en $[0, 1]$



- ▶ Usamos la función de activación de la última capa para forzar la salida en $[0, 1]$

$$[\text{softmax}(\mathbf{x})]_i = \frac{\exp([\mathbf{x}]_i)}{\sum_{j=1}^N \exp([\mathbf{x}]_j)}$$

⇒ Esta función es útil, pero puede traer problemas numéricos si $[\mathbf{x}]_i$ es grande

- ▶ Cuando juntamos esta función con el logaritmo de la verosimilitud, obtenemos

$$\ln(L(\mathbf{y}; \mathbf{x}, \theta)) = \sum_{c=1}^C [\mathbf{y}]_c \ln \left(\frac{\exp([\mathbf{z}]_c)}{\sum_{c'=1}^C \exp([\mathbf{z}]_{c'})} \right) = \sum_{c=1}^C [\mathbf{y}]_c \left([\mathbf{z}]_c - \ln \left(\sum_{c'=1}^C \exp([\mathbf{z}]_{c'}) \right) \right)$$

donde $\mathbf{z} = \mathbf{W}_L \mathbf{x}_{L-1} + \mathbf{b}_L$ la salida de la última transformación lineal

⇒ Linearizamos, no hay más división ⇒ Numéricamente más manejable

- ▶ Si sabemos que vamos a usar el softmax en combinación con el logaritmo de la verosimilitud
 - ⇒ Juntar ambas funciones en la función de pérdida $J \Rightarrow \text{nn.CrossEntropyLoss}$ vs. nn.NLLLoss

- ▶ Podemos usar la función de activación de la última capa para forzar los valores que queremos
 - ⇒ Debemos hacerlo de manera cuidadosa, teniendo en cuenta el gradiente
 - ⇒ Y también teniendo en cuenta la interacción con la función de pérdida que elegimos
- ▶ Elecciones típicas de la capa de salida (*output units*)
 - ⇒ Unidades lineales ⇒ La función de activación es la identidad ⇒ Gradiente fácil
 - ⇒ Unidades con saturación ⇒ Fuerzan la salida a valores deseados (tanh, sigmoid, softmax)
 - Las unidades de saturación pierden el gradiente ⇒ Combinar con la función de pérdida
 - ⇒ Otras no-linealidades ⇒ Elegir cuidadosamente la función de pérdida, observar la interacción

- ▶ Cómo las funciones de activación **afectan al gradiente** ⇒ El flujo de información
- ▶ Unidades de rectificación lineal (ReLU) ⇒ El gradiente es 1 cuando la unidad está activa
 - ⇒ Vamos a tener un **flujo de información que no cambia** ⇒ Fáciles de optimizar
 - ⇒ **Cuando se inactiva, ya perdemos el flujo de información** ⇒ El *bias b* es importante
 - ⇒ Inicializar **b** con un valor positivo para asegurar que el gradiente fluye al inicio del entrenamiento
- ▶ Unidades con saturación ⇒ El gradiente se vuelve 0 fuera del origen
 - ⇒ **Cuando el gradiente es 0 no tenemos información** ⇒ **No podemos salir de la zona de saturación**
 - ⇒ Tendríamos que asegurarnos que cuando estamos en la zona de saturación el valor es correcto

- ▶ La función de costo puede no ser derivable ⇒ Diseñar una función de pérdida
 - ⇒ La función de pérdida tiene que tener buenos gradientes y ser un buen reemplazo
- ▶ Muchas veces, queremos que la función de pérdida baje, no necesariamente encontrar el mínimo
- ▶ La función de activación de la última capa tiene que darnos los valores esperados de la salida
 - ⇒ Pero a veces eso implica que los gradientes son malos ⇒ Combinarla con la función de pérdida
- ▶ En el diseño de capas intermedias, pensar cómo los gradientes determinan el flujo de información
 - ⇒ Los datos nos pueden dar información sobre la salida, pero no sobre las capas intermedias

Funciones de Costo

Regularización

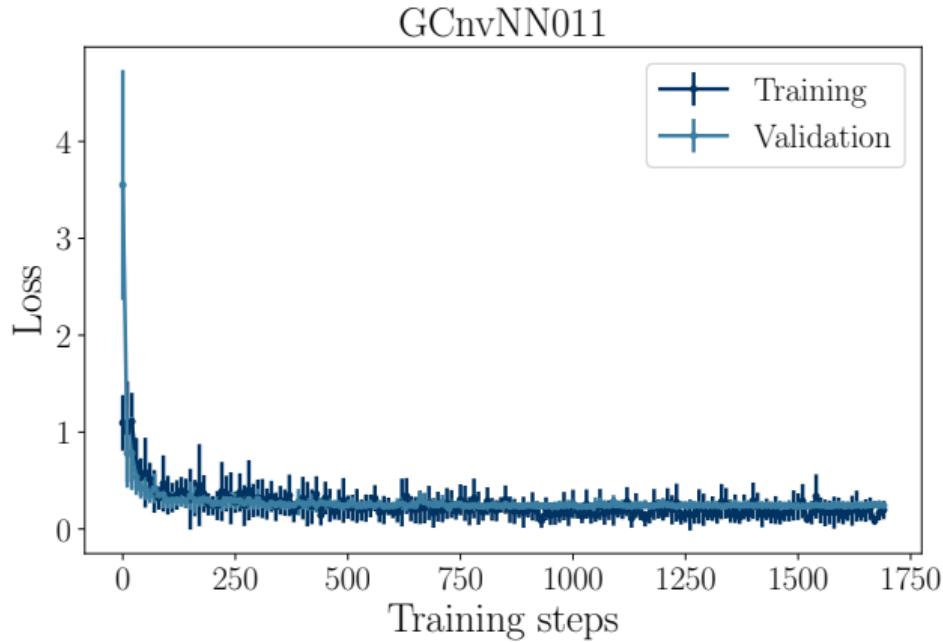
- ▶ Hasta ahora venimos diciendo que aprender es equivalente a minimizar el riesgo empírico

$$\min_{\theta} \frac{1}{M} \sum_{m=1}^M J(f(\mathbf{x}_m; \theta))$$

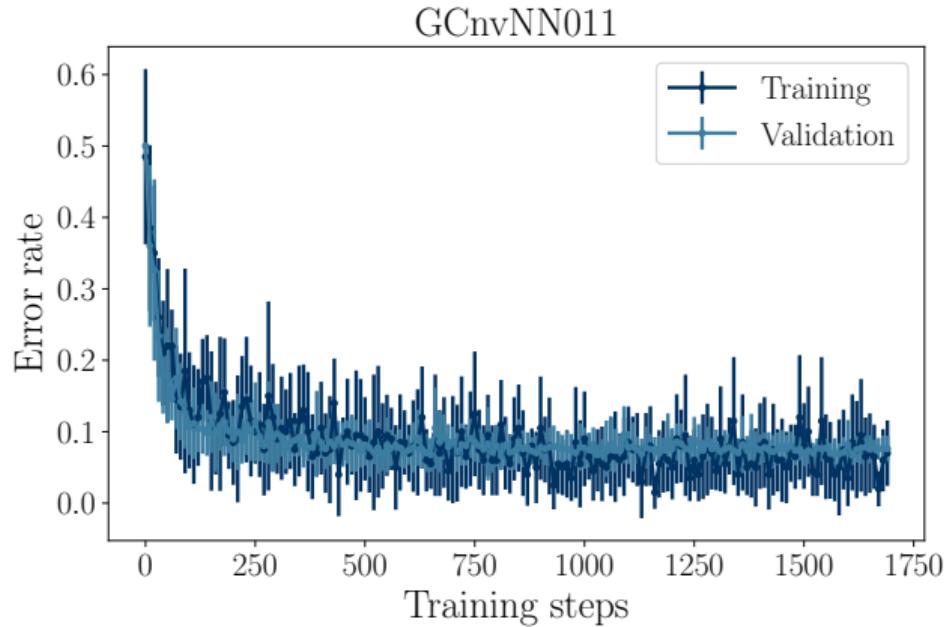
- ▶ Resolviendo este problema obtenemos una función f
 - ⇒ Esta función se aprendió mirando únicamente las muestras de entrenamiento $\{\mathbf{x}_1, \dots, \mathbf{x}_M\}$
 - ⇒ Pero lo que queremos es que la f que aprendimos funcione bien en nuevas muestras \mathbf{x}
- ▶ En teoría, esto es cierto si las nuevas muestras tienen la misma distribución que las de entrenamiento
 - ⇒ Pero en la práctica, la realidad es un poco más complicada
- ▶ Minimizar el riesgo empírico no siempre garantiza una buena generalización a nuevas muestras

- ▶ Overfitting es un problema habitual \Rightarrow Baja pérdida en el entrenamiento, pero no mejora el testeo
 \Rightarrow ¿Cómo puedo saber cuándo esto está pasando? \Rightarrow Muestras de validación
- ▶ Muestras de validación \Rightarrow Pequeño conjunto de las muestras de entrenamiento (5 %, 10 %)
 \Rightarrow No se usan para entrenar (si las usaría para entrenar, volvería al problema de overfitting)
 \Rightarrow Actúan como muestras de testeo ‘en tiempo de entrenamiento’
- ▶ Cada una determinada cantidad de actualizaciones de los parámetros $\theta_{k+1} \leftarrow \theta_k - \eta \partial \hat{R} / \partial \theta_k$
 \Rightarrow Evaluamos el riesgo en las muestras de validación y vemos cómo evoluciona

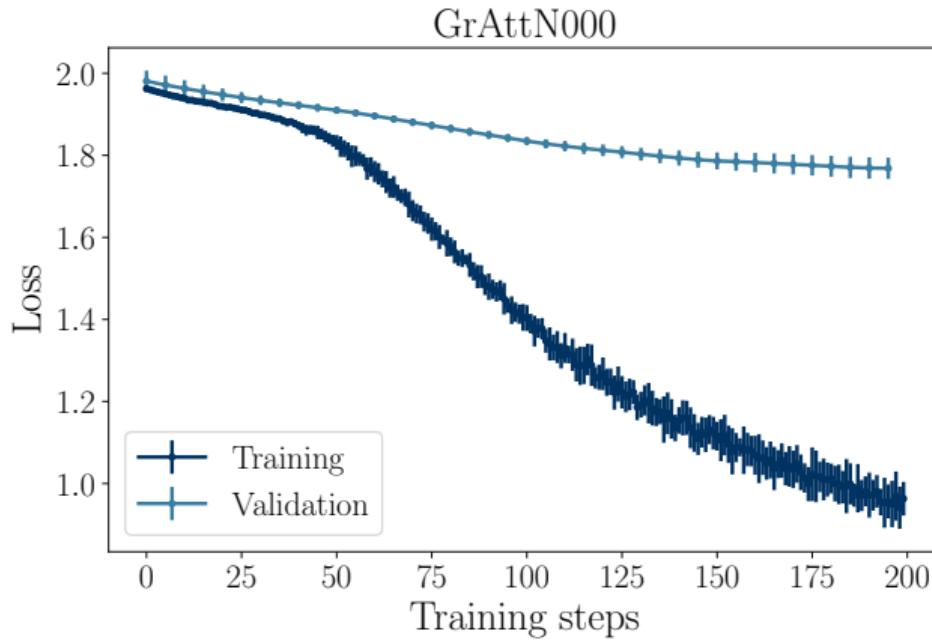
- ▶ Comparamos la evolución del riesgo en las muestras de entrenamiento y las de validación



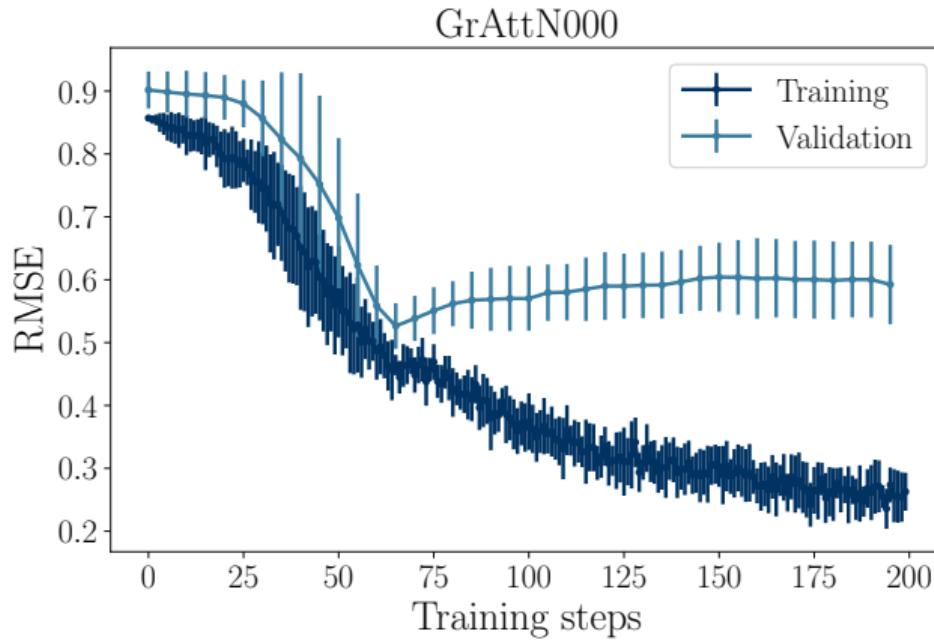
- ▶ Comparamos la evolución del riesgo en las muestras de entrenamiento y las de validación



- ▶ Comparamos la evolución del riesgo en las muestras de entrenamiento y las de validación

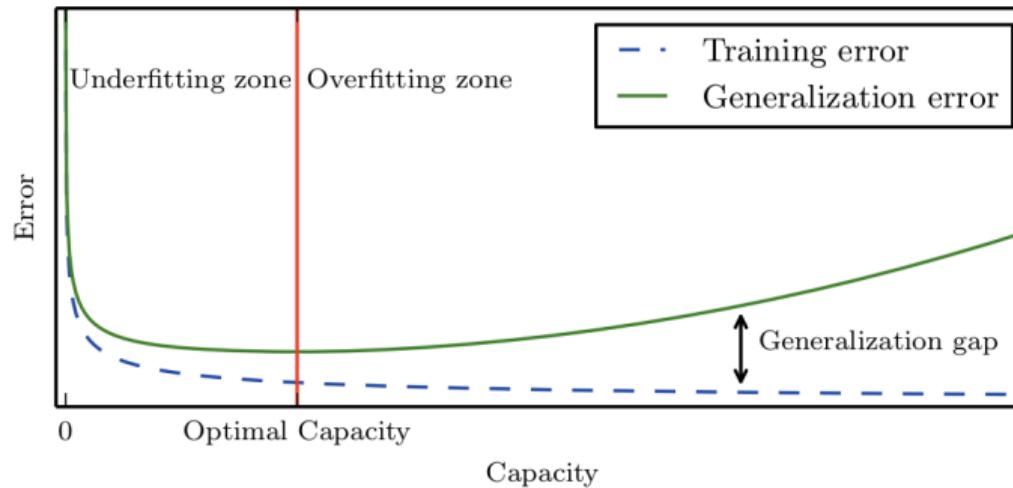


- ▶ Comparamos la evolución del riesgo en las muestras de entrenamiento y las de validación

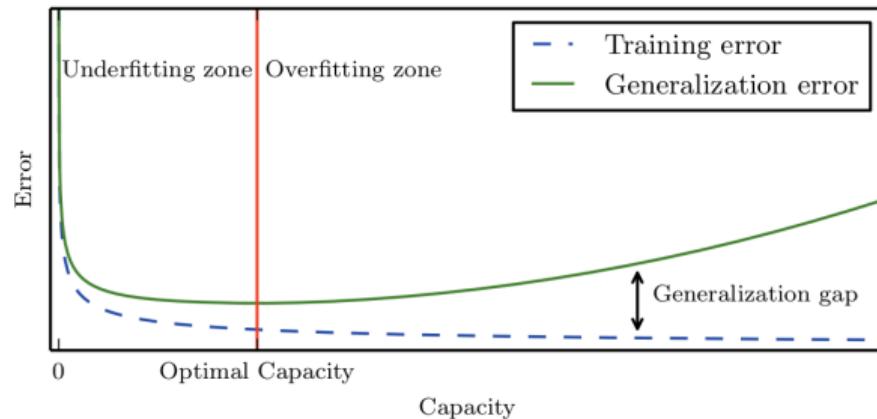


- ▶ Detectar overfitting es, posiblemente, la tarea más importante de las muestras de validación
 - ⇒ Pero no es la única
- ▶ Las redes neuronales dependen de una serie de hiperparámetros: cantidad de capas, unidades, etc.
 - ⇒ Estos hiperparámetros no se aprenden ⇒ Se diseñan
 - ⇒ Se pueden utilizar las muestras de validación para determinar los mejores hiperparámetros
- ▶ Hay varios métodos
 - ⇒ Búsqueda exhaustiva ⇒ Todas las combinaciones posibles dentro de unos valores predeterminados
 - ⇒ Búsqueda aleatoria ⇒ Combinaciones aleatorias de los valores predeterminados
 - ⇒ Búsqueda por gradiente ⇒ Si la función de costo es derivable respecto de los hiperparámetros

- ▶ El uso de muestras de validación es absolutamente clave para detectar overfitting
⇒ ¿Cómo lo arreglamos? ⇒ Ajustando la capacidad



- El problema es que el concepto de capacidad es muy difícil de medir
 - ⇒ Sabemos que depende de la dimensión de las capas ⇒ Pero no sabemos exactamente cómo
 - ⇒ Nunca sabemos exactamente en qué parte de la curva estamos



- Regularización ⇒ Técnicas usadas para mejorar la capacidad de generalización
 - ⇒ En general a expensas de incrementar el error de entrenamiento

- ▶ Regularización ⇒ Técnicas usadas para mejorar la capacidad de generalización
- ▶ Modelos sumamente complicados, con mucha capacidad, pueden ser difíciles de optimizar
 - ⇒ Además, no sabemos exactamente dónde está la función óptima ⇒ Aproximación Universal
 - ⇒ No sabemos si la arquitectura que elegimos es una buena aproximación
- ▶ Imponer restricciones sobre la arquitectura ⇒ Valores que los parámetros pueden tomar
- ▶ Agregar términos a la función de pérdida penalizando comportamientos inadecuados
- ▶ Considerar conocimiento previo sobre el comportamiento de los datos (distribución, estructura, etc.)
- ▶ Privilegiar modelos simples por sobre modelos complicados

- ▶ Limitar la capacidad del modelo restringiendo los valores que pueden tomar los parámetros
 - ⇒ Incluir una penalidad Ω en la función de pérdida

$$\tilde{J}(f(\mathbf{x}; \boldsymbol{\theta})) = J(f(\mathbf{x}; \boldsymbol{\theta})) + \alpha \Omega(\boldsymbol{\theta})$$

- ⇒ El (hiper)parámetro α se conoce como multiplicador ⇒ Determina el peso de la penalidad
- ▶ La elección de Ω influye en el comportamiento de los parámetros
 - ⇒ Este tipo de penalidades no depende de los datos
 - ⇒ En general no suman costo computacional ⇒ Son fáciles de calcular
 - ⇒ Suelen ser penalidades actuando únicamente sobre la matriz \mathbf{W}_ℓ para todas las capas

- ▶ Una forma de imponer penalidades sobre los parámetros es **controlando su norma**
⇒ Vimos que había muchos tipos de norma: L_1 , L_2 , L_∞
- ▶ La norma L_2 es la norma Euclídea: $\|\theta\|_2 = \theta^\top \theta$

$$\tilde{J}(f(\mathbf{x}; \boldsymbol{\theta})) = J(f(\mathbf{x}; \boldsymbol{\theta})) + \alpha \boldsymbol{\theta}^\top \boldsymbol{\theta}$$

- ⇒ Lleva el nombre de *weight decay*, *ridge regression* o regularizador de Tikhonov
- ▶ Aplicando **el gradiente descendente** se observa que

$$\boldsymbol{\theta} \leftarrow (1 - \eta \alpha) \boldsymbol{\theta} - \eta \frac{\partial J}{\partial \boldsymbol{\theta}}$$

⇒ La influencia del valor actual de los parámetros se reduce en $\eta \alpha$

- ▶ Vimos lo que pasa en cada actualización \Rightarrow Pero cuál es el efecto durante el entrenamiento?
 \Rightarrow Para realizar un análisis, asumimos que la función es cuadrática alrededor del óptimo θ^*

$$J(f(\mathbf{x}; \boldsymbol{\theta})) \approx J_o(\boldsymbol{\theta}^*) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}^*)^\top \mathbf{H}(\boldsymbol{\theta} - \boldsymbol{\theta}^*)$$

- ▶ Agregamos la penalidad, tomamos la derivada, e igualamos a cero

$$\frac{\partial \tilde{J}}{\partial \boldsymbol{\theta}} = \alpha \boldsymbol{\theta} + \mathbf{H}(\boldsymbol{\theta} - \boldsymbol{\theta}^*) = 0$$

Resolviendo para $\boldsymbol{\theta}$ obtenemos la siguiente expresión

$$\boldsymbol{\theta} = (\mathbf{H} + \alpha \mathbf{I})^{-1} \mathbf{H} \boldsymbol{\theta}^*$$

- El valor de θ que minimiza la función \tilde{J} viene dado por la siguiente ecuación

$$\theta = (\mathbf{H} + \alpha \mathbf{I})^{-1} \mathbf{H} \theta^*$$

- Lo primero que observamos, es que si $\alpha \rightarrow 0$ (no tenemos penalidad) $\Rightarrow \theta = \theta^*$
- Para seguir con el análisis, recordemos que \mathbf{H} es simétrica entonces puedo diagonalizar $\mathbf{H} = \mathbf{V} \Lambda \mathbf{V}^\top$

$$\theta = \mathbf{V}(\Lambda + \alpha \mathbf{I})^{-1} \Lambda \mathbf{V}^\top \theta^*$$

\Rightarrow Los autovalores de la ecuación son $\frac{\lambda_i}{\lambda_i + \alpha}$

- Las direcciones donde el gradiente avanza mucho $\lambda_i \gg \alpha \Rightarrow$ El efecto de la regularización es pequeño
- Las direcciones que no contribuyen mucho, se empujan hacia cero
- La regularización no afecta las direcciones de gran avance del gradiente

- ▶ La norma L_1 se obtiene como la suma de los valores absolutos de los elementos

$$\|\boldsymbol{\theta}\|_1 = \sum |[\boldsymbol{\theta}]_i|$$

- ▶ El efecto de una regularización de L_1 es bastante distinto al de L_2
⇒ Usar una norma de L_1 introduce una tendencia del vector a ser ralo (*sparse*)
- ▶ Un vector ralo es un vector que tiene muy pocos valores no-nulos
- ▶ Puede ser útil para determinar cuáles unidades dentro de la capa son más importantes

- En lugar de resolver el problema de minimizar la pérdida con la penalidad adicional, resolvemos

$$\underset{\theta}{\text{minimizar}} \ J(f(x; \theta)) \\ \text{sujeto a } \|\theta\| \leq C$$

⇒ Impone una restricción más fuerte sobre los valores aceptables de la norma

- Se puede resolver utilizando gradiente descendente y el concepto de Lagrangiano
⇒ Automáticamente elige el valor óptimo de α ⇒ Pero es más costoso
- Se puede utilizar gradiente descendente y luego un esquema de proyecciones ⇒ Es más costoso
- Incluir una penalidad, en vez de una restricción, puede crear mínimos locales

- ▶ Cuanto más datos tenemos, mejor va a generalizar el algoritmo de aprendizaje
- ▶ El problema es que la cantidad de datos disponibles es típicamente limitada
 - ⇒ Pero en muchas situaciones, es posible crear muestras “falsas”
- ▶ En el caso de problemas de clasificación con imágenes, esto es bastante fácil



Original



Rotación



Traslación

- ▶ En el caso de reconocimiento de objetos en imágenes, estas técnicas de incremento son fundamentales
⇒ Rotación, traslación, son operaciones fácilmente aplicables que mejoran mucho el aprendizaje
- ▶ Pero hay muchos casos en los que conviene tener cuidado con no cambiar el significado



Original



Rotación

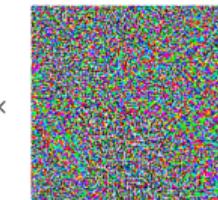
- ▶ La imagen rotada... ¿Es un 9 como la original, o es un 6?

- ▶ En muchos casos, sin embargo, no va a ser tan fácil (rotaciones fuera del plano)
⇒ En general vamos a querer muestrear $f_x(\mathbf{x})$ ⇒ Pero esto es justamente lo que desconocemos
- ▶ Una forma fácil de incrementar las muestras es inyectarles ruido (pequeño)
- ▶ Las redes neuronales deberían funcionar bien para muestras con poco ruido ⇒ Pero no es el caso



“panda”

57.7% confidence



noise

=



“gibbon”

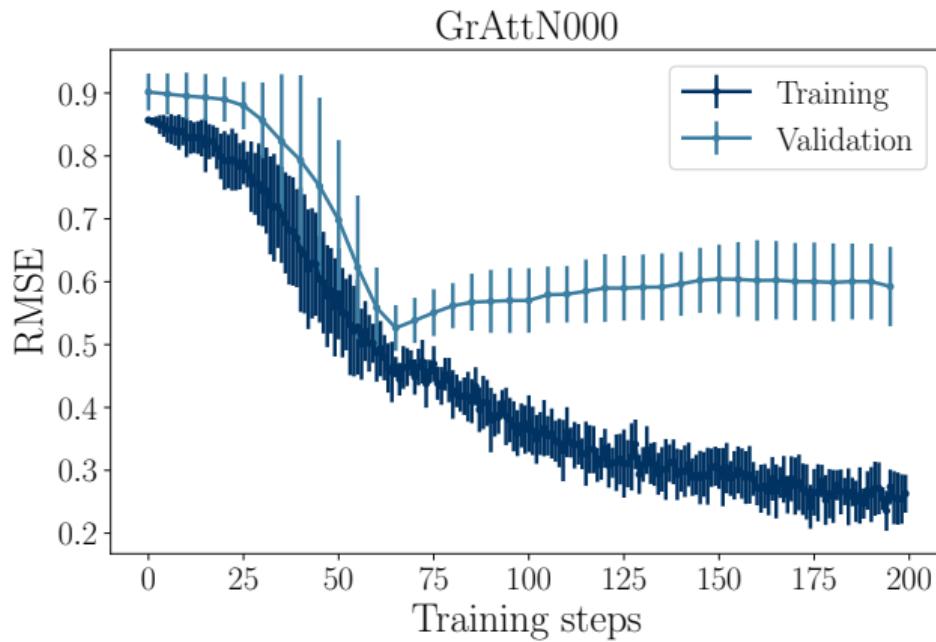
99.3% confidence



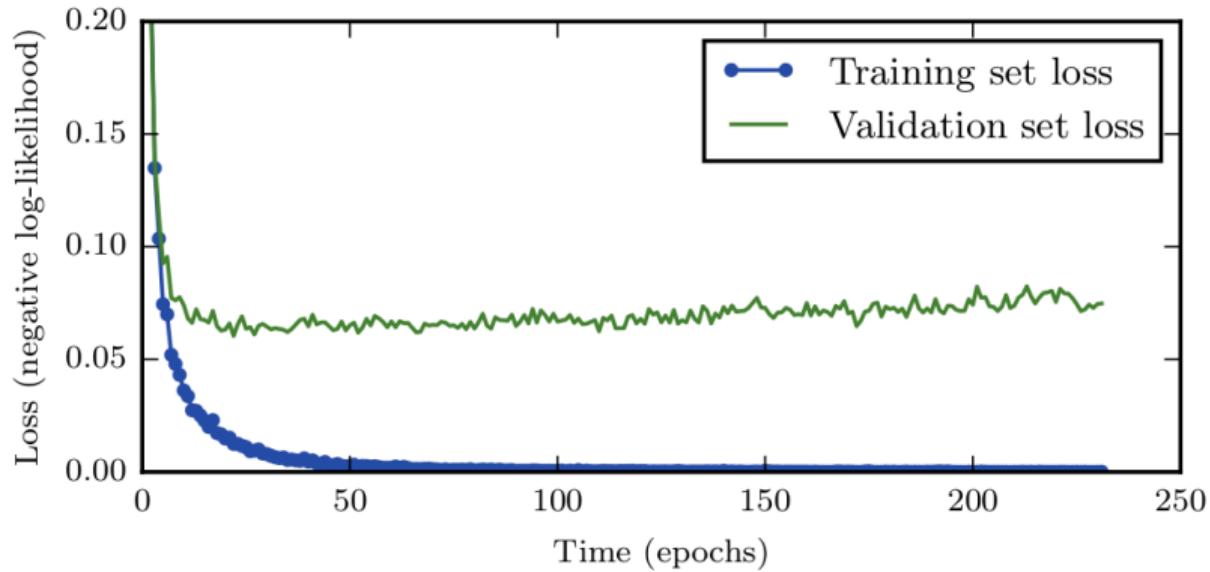
Esto es un *gibbon*

- ▶ Incrementar el dataset inyectando ruido ayuda a la robustez de la red neuronal

- ▶ Recordemos que estas situaciones pasan en la vida real



- ▶ Recordemos que estas situaciones pasan en la vida real \Rightarrow Hasta el libro la muestra



- ▶ Si el entrenamiento iba bien hasta un determinado momento donde empezó el overfit
- ▶ Y a partir de allí el costo sobre las muestras de validación empezó a aumentar
 - ⇒ ¿Por qué seguimos entrenando? ¿No sería perjudicial eso? Sí, lo es.

- ▶ Parada anticipada (*Early Stopping*) ⇒ La forma más popular de regularizar el entrenamiento
 - ⇒ Es eso: cuando veo que el costo en las muestras de validación sube, paro
- ▶ Es altamente efectivo y sumamente simple de implementar
- ▶ Se lo puede pensar como una selección automática de la cantidad de epochs
- ▶ Implica calcular el costo sobre las muestras de validación ⇒ Más lento (paralelizar)
- ▶ Requiere mantener en memoria una copia de los parámetros ⇒ Complicado para redes grandes

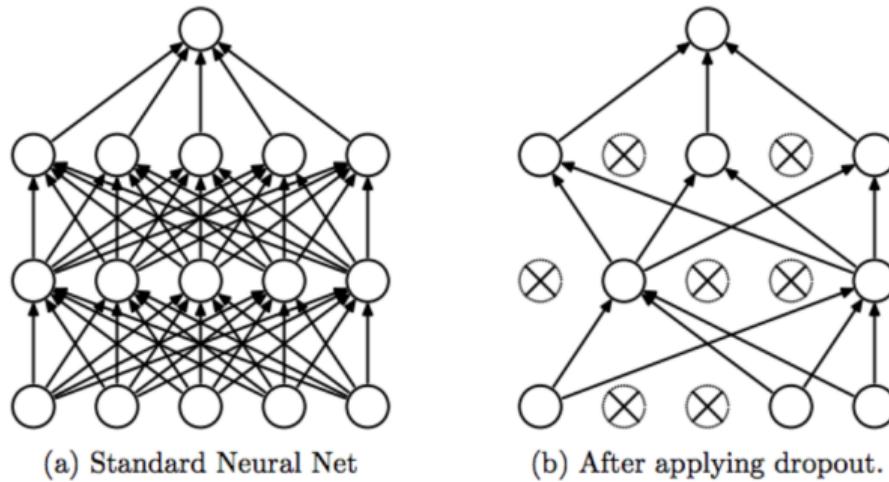
- ▶ Parada anticipada \Rightarrow Se puede interpretar como una manera transparente de regularización
 - \Rightarrow No afecta la función de pérdida ni restringe los valores posibles de los parámetros
 - \Rightarrow Es fácil de usar sin alterar la dinámica de aprendizaje
- ▶ Se puede usar en soledad, o se puede combinar con otras formas de regularización
- ▶ Suele resultar en una reducción del costo de entrenamiento \Rightarrow Se entrena menos tiempo
- ▶ No demanda la especificación de ningún tipo de multiplicador o el cálculo de gradientes extra

- ▶ A veces no sabemos exactamente los valores de \mathbf{W}_ℓ y \mathbf{b}_ℓ
⇒ Pero **sabemos (por conocimiento del dominio)** que los elementos tienen que estar relacionados
- ▶ Un ejemplo es cuando usamos penalidades sobre la norma
- ▶ Pero se puede poner como **restricción, obligando a ciertos elementos de \mathbf{W}_ℓ a ser iguales**
⇒ *Parameter sharing* porque se ‘comparten’ los parámetros (los distintos elementos de \mathbf{W}_ℓ)
- ▶ Una ventaja importante: **se aprenden menos parámetros** ⇒ Facilita la optimización

- ▶ Podemos pedir que las representaciones (i.e. la salida de la red neuronal) sean ralas
⇒ Imponemos una penalidad de norma L_1 sobre el vector de salida

- ▶ El **overfitting** se asocia con la noción de **memorización**
 - ⇒ La red neuronal memoriza las muestras de entrenamiento
 - ⇒ En particular, **cada neurona memoriza un concepto/un aspecto diferente**
- ▶ Si queremos evitar overfitting ⇒ Tenemos que evitar la memorización de las neuronas
 - ⇒ **Dropout** ⇒ Apagar aleatoriamente ciertas neuronas

- ▶ Dropout \Rightarrow Apagar aleatoriamente ciertas neuronas
 - \Rightarrow Distintas neuronas están activas a distintos momentos
 - \Rightarrow Varias neuronas tienen que aprender el mismo concepto
 - \Rightarrow Una neurona cualquiera tiene que aprender muchos conceptos distintos



- ▶ Dropout \Rightarrow Apagar aleatoriamente ciertas neuronas
- ▶ Las neuronas se apagan aleatoriamente de forma independiente con probabilidad p
- ▶ Distintas neuronas se apagan en cada paso del entrenamiento
- ▶ Las neuronas sólo se apagan durante entrenamiento, no durante la evaluación
 - \Rightarrow Varias neuronas procesan el mismo concepto \Rightarrow Menos propenso a errores
- ▶ De fácil implementación, no requiere ningún tipo de memoria extra
- ▶ Es una forma de restringir los parámetros, de facilitar la optimización

- ▶ Normalización del batch ⇒ Calcular media y varianza de cada batch ⇒ Normalizar
 - ⇒ Particularmente útil para **modelos profundos** (con muchas capas)
 - ⇒ Evita que los valores de las capas intermedias se disparen
 - ⇒ Se puede aplicar antes de cada una de las capas ⇒ Normalizar las entradas
- ▶ Normalización de la capa ⇒ Aprender los valores de normalización
 - ⇒ Se aprende sobre todas las muestras **en todas las capas**
 - ⇒ Son los mismos valores durante el entrenamiento y la evaluación
 - ⇒ Particularmente útil en **redes neuronales recurrentes** (Episodio 6)

- ▶ Minimizar el riesgo empírico usando gradiente descendente
 - ⇒ Implica que la función es **derivable** ⇒ **Costo vs. Pérdida**
- ▶ No necesariamente necesitamos encontrar el mínimo, sino bajar el costo
- ▶ Muestras de **validación** ⇒ Sirven para **observar el overfitting**
- ▶ **Regularización** ⇒ Técnicas para **mejorar la generalización** del modelo
 - ⇒ Penalidades ⇒ Agregan un término a la función de pérdida
 - ⇒ Parada anticipada ⇒ Fácil implementación, requiere muestras de validación
 - ⇒ Dropout ⇒ Apagar neuronas de manera aleatoria ⇒ Evitar memorización

- ▶ A veces, los **datos tienen estructuras** que se pueden explotar para mejorar el algoritmo
 - ⇒ Señales en el tiempo, sistemas para procesarlos
- ▶ Operación de **convolución** ⇒ Operación lineal que **explota la estructura**
- ▶ **Respuesta en frecuencia** ⇒ Representación alternativa de señales con estructura
 - ⇒ Permite encontrar una **manera rápida y fácil de calcular la convolución**
- ▶ Convolución en **imágenes** ⇒ Particularidades y observaciones