Universidad de la República
Facultad de Ciencias Económicas y de Administración

Trabajo final de grado para obtener el título de Licenciado en Estadística

REDES GENERATIVAS ADVERSARIAS

Julio Cesano y Mathias Fuidio

Docente orientador: Mathias Bourel

Montevideo Uruguay, Marzo 2024

Facultad de Ciencias Económicas y de Administración.

El tribunal docente integrado por los abajo firmantes aprueba el trabajo final de grado:

REDES GENERATIVAS ADVERSARIAS

Julio Cesano y Mathias Fuidio

Docente orientador: Mathias Bourel

Licenciatura en Estadística

Tribunal	**Firma**
Mathias Bourel	
Marco Scavino	
María Inés Fariello	

Nota final: 12

Resumen

Las redes generativas adversarias, o GAN (por sus siglas en inglés), son un enfoque para el modelado generativo que utiliza métodos de aprendizaje profundo, como las redes neuronales. El modelado generativo es un algoritmo particular dentro del aprendizaje automático que implica descubrir y aprender patrones en los datos de entrada y generar nuevos ejemplos que de manera plausible podrían haberse extraído del conjunto de datos original.

Las GAN son una forma de entrenar un modelo generativo al abordar el problema como un ejercicio de aprendizaje no supervisado con dos submodelos: un modelo generador que se entrena para generar nuevos ejemplos y otro modelo discriminador que clasifica los ejemplos como reales (provenientes del dominio original) o falsos (provenientes del generador). A grandes rasgos, los dos modelos se entrenan conjuntamente en un juego de suma cero, contradictorio, hasta que el modelo discriminador es "engañado" aproximadamente la mitad de las veces, lo que significa que el modelo generador está generando ejemplos plausibles de pertenecer al dominio real.

El presente trabajo, estudia en detalle el funcionamiento de las redes neuronales generativas. Se presenta un detalle teórico y se aplica al contexto de visión artificial, generando: en primer lugar, un modelo que permite clasificar el conjunto de datos MNIST a partir de una arquitectura de red neuronal convolucional. En segundo lugar, un modelo de tipo GAN convolucional capaz de generar dígitos manuscritos. Se logró comprender en profundidad el funcionamiento de las redes neuronales generativas adversarias en el contexto de la visión artificial, su arquitectura, métodos de aprendizaje y análisis de performance. Se implementó exitosamente una GAN al caso de estudio antes mencionado.

Palabras clave: GAN, modelos generativos, redes neuronales, discriminador, generador, función de perdida.

Índice

1. Introducción

En inteligencia artificial y aprendizaje automático se han desarrollado algoritmos eficientes en la resolución de diversos problemas en disciplinas como el reconocimiento de patrones; que nos permite asignar correctamente la categoría de un nuevo elemento habiendo aprendido previamente de una base de datos. Soluciones a problemas de regresión; donde se estima un valor numérico a partir de un conjunto de datos relacionados. Incluso algoritmos con la capacidad de derrotar a un gran maestro de ajedrez, estimar los movimientos del precio de las acciones y clasificar si es probable que una transacción con tarjeta de crédito sea fraudulenta; como se expresa en [1, cap 1]. Sin embargo, cuando se desea generar datos realistas de manera artificial, las computadoras han presentado serias dificultades; hasta la aparición de las redes generativas adversarias o GAN por sus siglas en inglés (Generative Adversarial Networks). Estas fueron presentadas por Ian Goodfellow, Yoshua Bengio y su equipo en 2014 [2]. Estos algoritmos han permitido la generación de datos con una calidad que en el pasado se consideraba imposible. En este marco, se destaca la generación de rostros artificiales de personas tan reales que son indistinguibles para el ojo humano [1, cap 1]. Esta última aplicación motiva el estudio en profundidad de esta herramienta y la realización del presente trabajo.
Las GAN son una clase de algoritmos de inteligencia artificial, implementadas por un sistema compuesto por dos modelos que compiten entre sí. Por un lado, el generador entrenado para producir datos artificiales y, por otro, el discriminador entrenado para distinguir datos creados artificialmente de aquellos reales [1, cap 1]. El objetivo global de un modelo GAN, es la generación de datos artificiales a partir de un conjunto de datos de entrenamiento no etiquetados. De ahí proviene el término **generativo**. Por lo tanto, se puede decir que las GAN son un modelo de aprendizaje automático **no supervisado**. El término **adversario** hace referencia a la dinámica competitiva existente entre los dos modelos que componen una GAN. Finalmente, el término **redes** (o networks) refiere a la clase de modelos de aprendizaje automático generalmente utilizados para representar al generador y al discriminador. Estos modelos son redes neuronales, cuya estructura y funcionamiento básico, se verá a lo largo del trabajo [3, cap 1].

El generador (G) modela la distribución de los datos, mientras que el discriminador (D) estima la probabilidad de que la muestra provenga de un conjunto de datos reales. El proceso de entrenamiento se basa en la teoría de juegos, donde nos enfrentamos a un problema de suma cero. Por un lado, G busca maximizar la probabilidad de que D clasifique de manera errónea y, por otro, D busca minimizar dicho error [2].

Es muy amplio el espectro de elementos que pueden ser simulados. A lo largo del presente trabajo profundizaremos en distintas aplicaciones de las cuales se puede destacar: la creación de rostros humanos, animales, obras de arte, dígitos manuscritos. A su vez, se destaca el gran éxito de estos modelos respecto a otros modelos generadores ya existentes [4].

El capítulo 3 *Redes generativas adversarias* presenta cada componente y cómo estos se relacionan entre sí. Por último, el capítulo 4 *Puesta en práctica de redes neuronales y GAN* presenta dos aplicaciones en el contexto de visión artificial: la primera refiere a un clasificador de dígitos manuscritos y la segunda implementa un modelo con arquitectura GAN que permite la generación de imágenes de dígitos similares a los utilizados para su entrenamiento.

2. Redes neuronales - Conceptos básicos

Esta sección desarrolla el marco teórico de las redes neuronales, dado que es la estructura más utilizada para representar al Discriminador y Generador, siendo estos los dos componentes principales de un modelo GAN. El capítulo comienza con una breve reseña histórica de las redes neuronales. Plantea la estructura del Perceptrón y se muestra el impacto y evolución de las redes neuronales en el área de la inteligencia artificial, particularmente en la visión por computadora. La segunda sección, explica las redes neuronales de tipo prealimentada ("feedforward" en inglés) y las distintas funciones de activación utilizadas en una red. La tercera sección explica el procedimiento para entrenar una red, los conceptos de regularización, optimización y el método de backpropagation. Finalizamos el capítulo presentando redes neuronales de convolución.

2.1. Antecedentes

La primera red neuronal fue creada en el año 1957 en la universidad de Cornell por Frank Rosenblatt, logrando crear las primeras máquinas capaces de aprender según [5] y [6] y que las redes neuronales actuales han mantenido el mismo enfoque a lo largo del tiempo.

Perceptrón

La estructura de esta primera red neuronal [6] denominada Perceptrón, se compone de la sumatoria de un vector de entradas y un vector de pesos seguido de una función, que llamaremos de activación. El uso de este modelo se centra en la clasificación binaria de elementos [6]. Por simplicidad se define una clase con el valor 1 y otra con el valor 0. Por otro lado, se define la función de activación $\Theta(\mathbf{z})$ donde $\mathbf{z}$ es una combinación lineal del vector de entrada $\mathbf{x} \in R^n$ y el vector de pesos, $\mathbf{w} \in R^n$. Luego se tiene que $\mathbf{z} \in R$ será la entrada de la función de activación Θ. Esta estructura conforma una red neuronal simple compuesta por solamente una neurona artificial.

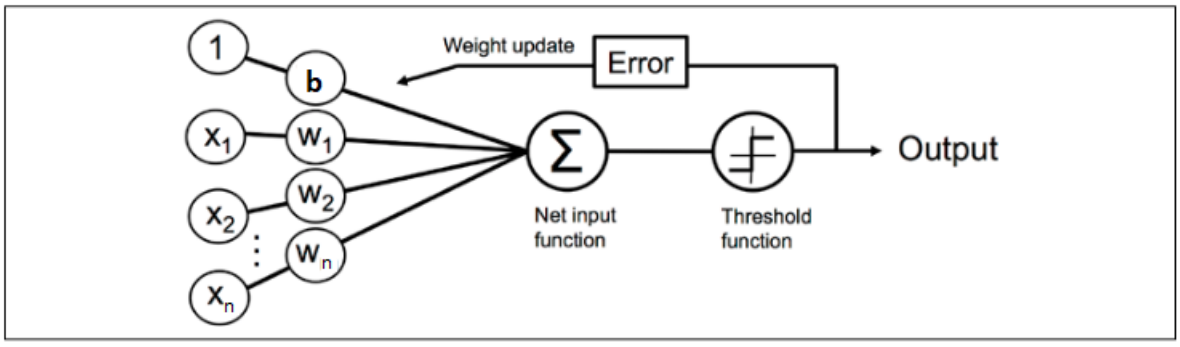


Figura 1: Diagrama perceptrón [6]

Sea:

$$\mathbf{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \quad \mathbf{z} = x_1 w_1 + \cdots + x_n w_n$$

Tomando al i-ésimo elemento de un conjunto de datos, que se denotará como $\mathbf{x}^{(i)}$, el resultado de la función de decisión $\Theta(\mathbf{z})$ aplicada a $\mathbf{x}^{(i)}$ será 0 o 1 dependiendo de un umbral definido, $b \in R$, como se muestra a continuación [6]:

$$\Theta(\mathbf{z}) = \begin{cases} 1 & \mathbf{z} \geq b \\ 0 & \text{caso contrario} \end{cases}$$

En la literatura de aprendizaje automático, al umbral o cota b se lo denomina sesgo (unit bias en inglés).

Por simplicidad, se define $b = 0$, por lo tanto, se obtiene la siguiente función de decisión:

$$\Theta(\mathbf{z}) = \begin{cases} 1 & \mathbf{z} \geq 0 \\ 0 & \text{caso contrario} \end{cases}$$

En la siguiente sección se presenta la regla de aprendizaje del perceptrón, en donde se verá a través de un ejemplo sencillo, el procedimiento y cálculos numéricos que se ejecutan para el entrenamiento del modelo.

Regla de aprendizaje del perceptrón

El algoritmo de entrenamiento comienza inicializando los pesos con un valor aleatorio. Estos serán ajustados a medida que se procesa cada elemento del conjunto de datos. Para cada ejemplo de entrenamiento, se calcula su valor de salida (o predicción) el cual notaremos como $\hat{y}^{(i)}$ correspondiente al paso de entrenamiento i. Si hay una diferencia entre el valor de predicción y el valor real, los pesos serán ajustados.[6]. El ajuste es proporcional al producto de dos factores; por un lado, el error de predicción calculado como la diferencia entre la salida deseada y la salida calculada) y; por otro, el valor de la entrada correspondiente. Cuando la estimación es correcta los pesos se mantienen sin cambios.

Es importante señalar que la convergencia del algoritmo de un Perceptrón se garantiza siempre que el conjunto de datos **contenga dos clases linealmente separables** y la tasa de aprendizaje ("learning rate" en inglés) sea lo suficientemente pequeña [6]. Obsérvese que si el conjunto de datos no es linealmente separable, el algoritmo quedará en un ciclo infinito de actualización de pesos, dado que el algoritmo no encontrará un hiperplano capaz de discriminar todos los elementos de forma correcta [6]. En la figura 2 se presentan ejemplos de conjuntos linealmente separables y no separables [6].

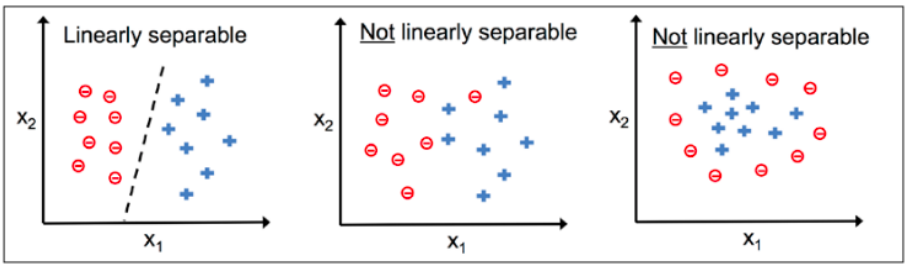


Figura 2: Ejemplos de conjuntos linealmente separables y no separables [6]

Los pasos del algoritmo de entrenamiento se pueden resumir en el siguiente orden:

1. Inicializar los pesos w_j con 0 o bien un valor pequeño.
2. Repetir hasta que no se produzcan cambios
   a) Para cada elemento del conjunto de entrenamiento, $\mathbf{x}^{(i)}$:
      1) Calcular el valor $\hat{y}^{(i)}$
      2) Calcular $\Delta w_j = \eta.(y^{(i)} - \hat{y}^{(i)})x_j^{(i)}$.
      3) Actualizar el valor de los pesos $w_j = w_j + \Delta w_j$.

El valor de actualización para w_j denominado Δw_j, es calculado por la regla del aprendizaje como:

$$\Delta w_j = \eta.(y^{(i)} - \hat{y}^{(i)})x_j^{(i)}.$$

El parámetro η se denomina tasa de aprendizaje (o "learning rate" en inglés) y típicamente se le asignan valores entre 0 y 1 de manera arbitraria, generalmente tomando un valor pequeño. El valor $y^{(i)}$ corresponde a la clase verdadera de $\mathbf{x}^{(i)}$ mientras que $\hat{y}^{(i)}$ corresponde a la predicción calculada por el modelo. Es importante notar que los pesos w_j se actualizan en forma simultánea [6].

Ejemplo de aplicación de la regla del Perceptrón a la función AND

Para brindar mayor claridad, se presenta a continuación la ejecución del algoritmo de aprendizaje del Perceptrón aplicado a la función AND.

Se toma un conjunto de datos perteneciente a R^2 donde para cada elemento $\mathbf{x}^{(i)} = (x_1^{(i)}, x_2^{(i)})$, se necesitan las siguientes actualizaciones simultáneas:

$$\Delta b = \eta(y^{(i)} - \hat{y}^{(i)})$$
$$\Delta w_1 = \eta(y^{(i)} - \hat{y}^{(i)})x_1^{(i)}$$
$$\Delta w_2 = \eta(y^{(i)} - \hat{y}^{(i)})x_2^{(i)}$$

Recordar que b corresponde al sesgo del modelo.
En los casos que el modelo predice correctamente, los valores de los pesos permanecen sin cambios.

(1) $y^{(i)} = 0$, $\hat{y}^{(i)} = 0$, $\Delta w_j = \eta.(0-0)x_j^{(i)} = 0$.

(2) $y^{(i)} = 1$, $\hat{y}^{(i)} = 1$, $\Delta w_j = \eta.(1-1)x_j^{(i)} = 0$.

Sin embargo, en los casos de error, los pesos se ajustan en dirección hacia el valor correcto del elemento.

(3) $y^{(i)} = 1$, $\hat{y}^{(i)} = 0$, $\Delta w_j = \eta.(1-0)x_j^{(i)} = \eta.(1)x_j^{(i)}$.

(4) $y^{i} = 0$, $\hat{y}^{i} = 1$, $\Delta w_j = \eta.(0-1))x_j^{(i)} = \eta.(-1)x_j^{(i)}$.

Vemos el funcionamiento del Perceptrón para clasificar el conjunto de puntos correspondiente a la función lógica AND. A continuación se presenta la tabla de entradas y salidas de la función:

x_1	x_2	AND
0	0	0
0	1	0
1	0	0
1	1	1

Gráficamente se desea encontrar una línea recta que separe los puntos que se muestran en la figura 3. En la figura se puede apreciar que claramente es posible encontrar una recta que separe adecuadamente el conjunto de puntos negros de los amarillos.

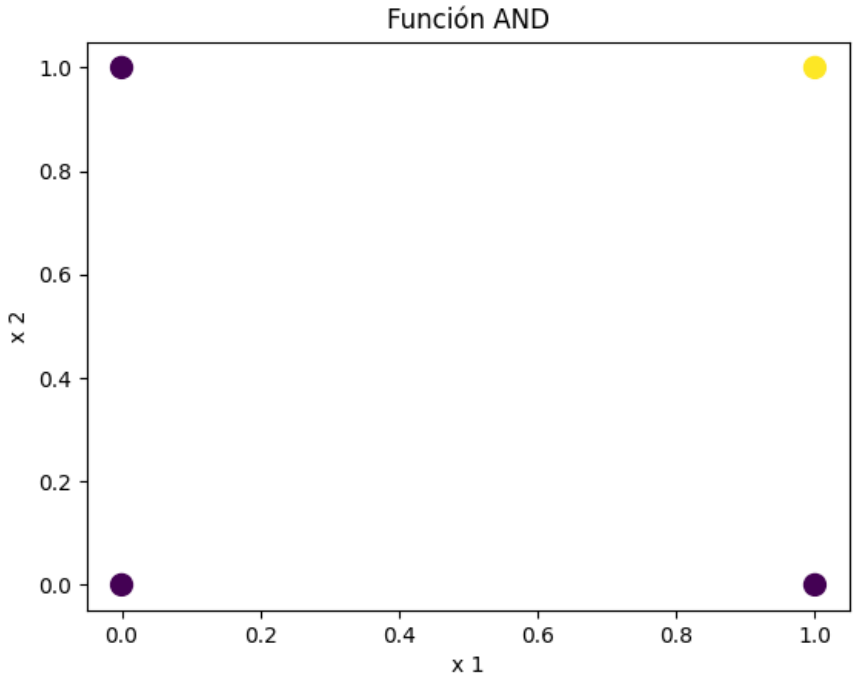


Figura 3: Gráfica que muestra los puntos de la función AND

Como valores iniciales de los parámetros, se toman valores arbitrarios pero pequeños de acuerdo a lo recomendado. Sin embargo, para el ejemplo de la función AND podemos inicializar simplemente con 1 todos los parámetros, obteniendo como valores iniciales:

$$b = 1(sesgo)$$
$$w_1 = 1$$
$$w_2 = 1$$

Por otro lado, se define un valor de tasa de aprendizaje de 0.3. A partir de la inicialización anterior, se presenta una tabla donde se muestra iterativamente cómo se actualizan los valores para cada iteración o época hasta que no se producen más cambios. Una época se cumple cuando todos los elementos del conjunto de datos han sido procesado.

Época	x_1	x_2	b	w_1	w_2	$\mathbf{z}$	$\hat{y}$	y	Δb	Δw_1	Δw_2
1	0	0	1	1	1	1	1	0	-0.3	0	0
1	0	1	0.7	1	1	1.7	1	0	-0.3	0	-0.3
1	1	0	0.4	1	0.7	1.4	1	0	-0.3	-0.3	0
1	1	1	0.1	0.7	0.7	1.5	1	1	0	0	0
2	0	0	0.1	0.7	0.7	0.1	1	0	-0.3	0	0
2	0	1	-0.2	0.7	0.7	0.5	1	0	-0.3	0	-0.3
2	1	0	-0.5	0.7	0.4	0.2	1	0	-0.3	-0.3	0
2	1	1	-0.8	0.4	0.4	0	1	1	0	0	0
3	0	0	-0.8	0.4	0.4	-0.8	0	0	0	0	0
3	0	1	-0.8	0.4	0.4	-0.4	0	0	0	0	0
3	1	0	-0.8	0.4	0.4	-0.4	0	0	0	0	0
3	1	1	-0.8	0.4	0.4	0	1	1	0	0	0

En la tabla anterior se observa que luego de 3 épocas no se producen más cambios, por lo que el algoritmo finaliza obteniendo los valores que determinan la recta final que clasifica el conjunto de datos. Para comprender la tabla anterior, a continuación se explican las primeras líneas y su relación con el algoritmo de aprendizaje del Perceptrón.

A partir de los valores iniciales de nuestros parámetros $b = 1$,$w_1 = 1$ y $w_2 = 1$ y el primer dato de entrada $x_1 = 0$ y $x_2 = 0$ en la primera fila, se observan las actualizaciones de los parámetros en función de la diferencia entre lo calculado $\hat{y} = 1$ y el resultado esperado $y = 0$. Se calculan los deltas de cada parámetro y se obtiene los resultados: $\Delta b = -0{,}3$, $\Delta w_1 = 0$ y $\Delta w_2 = 0$. Luego se actualizan los parámetros en la fila 2 obteniendo los valores: $b = 0{,}7$, $w_1 = 1$ y $w_2 = 1$.

En la segunda fila, se toma el segundo dato de entrada, se calcula nuevamente la salida $\hat{y}$ y se compara con el resultado esperado y para esta segunda salida. De ahí se obtienen los nuevos valores de los deltas y se actualizan los parámetros cuyos resultados se ingresan en la fila 3. Se continua con este procedimiento hasta procesar los 4 puntos de entrada del conjunto de datos. Una vez que se ha procesado todo conjunto se dice que se ha completado la época 1.

Se continúa con este procedimiento hasta que no se produzca ningún cambio para ninguno de los puntos del conjunto. Esto se aprecia en la época 3. Por lo que se ha llegado al final del algoritmo y los resultados de los parámetros determinan la recta que clasifica correctamente el conjunto de datos. Los parámetros finales son: $b = -0{,}8$, $w_1 = 0{,}4$ y $w_2 = 0{,}4$ lo que determina la recta $0{,}4x_1 + 0{,}4x_2 - 0{,}8 = 0$. Si despejamos x_2 tenemos que $x_2 = -x_1 + 2$.

En la figura 4 se muestra el conjunto de puntos de la función AND y la recta clasificadora.

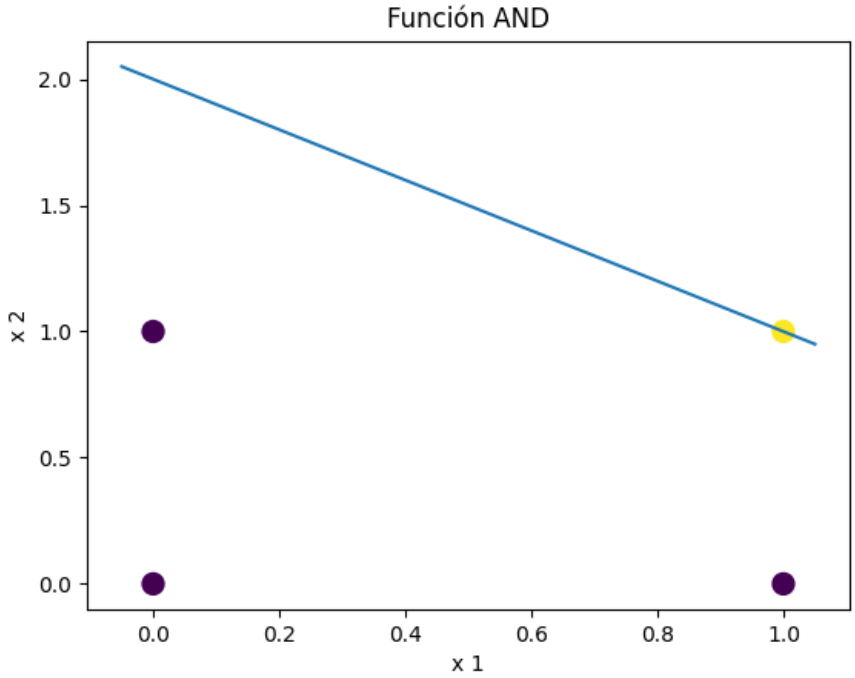


Figura 4: Gráfica que muestra los puntos de la función AND y su recta de clasificación

Demostración por absurdo que perceptrón no puede resolver la función XOR

Hubo gran expectativa respecto a la variedad de problemas de clasificación que se esperaba resolver a través de este modelo, hasta que en el año 1969 Marvin Minsky y Seymor Papert, publicaron un estudio profundo sobre el Perceptrón y mostraron limitaciones importantes respecto al modelo. Específicamente demostraron que el Perceptrón tiene la capacidad de clasificar únicamente patrones linealmente separables, como se ha mencionado anteriormente. Sin embargo, no puede resolver algo tan básico como la función XOR, como se ilustra en la figura 5. Antes de continuar con la demostración, en la figura se puede apreciar que no existe forma de separar linealmente el conjunto de puntos negros de los amarillos.

Por absurdo, supongamos que el Perceptrón puede resolver la función XOR. Entonces, se pueden construir las siguientes cuatro restricciones:

$$\begin{aligned} w_1x_1 + w_2x_2 &\geq 0 && \text{para } x_1 = 0, x_2 = 0 \\ w_1x_1 + w_2x_2 &< 0 && \text{para } x_1 = 0, x_2 = 1 \\ w_1x_1 + w_2x_2 &< 0 && \text{para } x_1 = 1, x_2 = 0 \\ w_1x_1 + w_2x_2 &\geq 0 && \text{para } x_1 = 1, x_2 = 1 \end{aligned}$$

Sustituyendo en las ecuaciones anteriores podemos ver que:

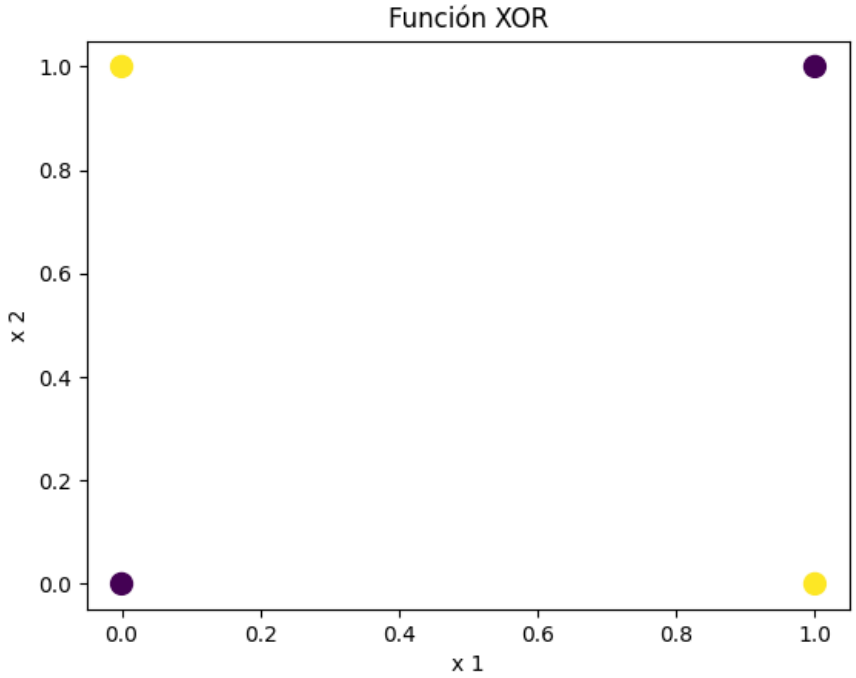


Figura 5: Gráfica que muestra los puntos de la función XOR

$$
\begin{aligned}
w_1 0 + w_2 0 &\geq 0 && \text{para } x_1 = 0, x_2 = 0 \\
w_1 0 + w_2 1 &< 0 && \text{para } x_1 = 0, x_2 = 1 \\
w_1 1 + w_2 0 &< 0 && \text{para } x_1 = 1, x_2 = 0 \\
w_1 1 + w_2 1 &\geq 0 && \text{para } x_1 = 1, x_2 = 1
\end{aligned}
$$

A partir de la segunda ecuación se obtiene que $w_1 < 0$ y de la tercera que $w_2 < 0$. Esto contradice la cuarta ecuación que establece que siendo ambos negativos, su suma es positiva $(w_1 x_1 + w_2 x_2 > 0\ )$. Por lo tanto, llegamos a la conclusión de que el perceptrón no puede resolver la función XOR.

Sin embargo, es posible clasificar correctamente la función XOR con un Perceptrón multicapa o red prealimentada, que se verá más adelante.

En los años 80 hubo investigadores liderados por Yann Le Cun, que trabajaron en el área de redes neuronales hasta lograr mediante una red neuronal, reconocer dígitos escritos a mano de una imagen [4]. Esta red fue la primera **red convolucional** entrenada mediante backpropagation. Los conceptos de convolución y backpropagation se verán más adelante a lo largo del capítulo. Este grupo de investigadores continuó trabajando hasta 1998, año en el que presentó la quinta versión de una red denominada LeNet-5; destacando el uso de la misma en el reconocimiento de dígitos escritos en cheques bancarios. La estructura general de la red LeNet-5 se muestra en la figura 6. La arquitectura consta de un total de 7 capas que constan de 2 conjuntos de capas convolucionales y 2 conjuntos de capas de agrupación promedio (también conocidas como capas de "Average Pooling" en inglés), seguidas de una

capa convolucional plana. Luego, se utiliza dos capas densas completamente conectadas y finalmente un clasificador softmax. Todos estos conceptos se estudiarán en detalle a lo largo del presente trabajo. Se puede encontrar un detalle sobre la estructura de LeNet5 en el artículo [7].

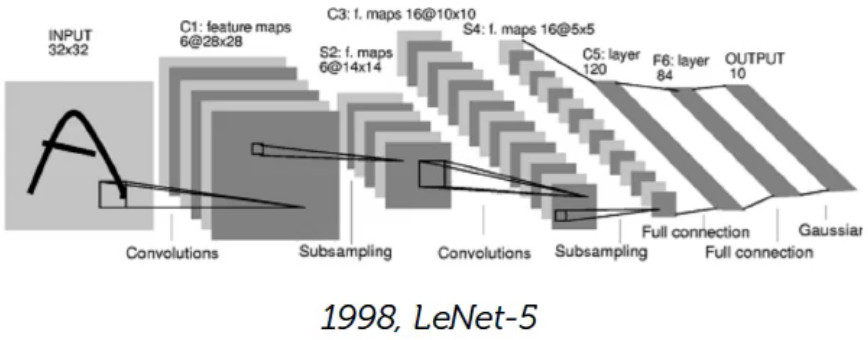


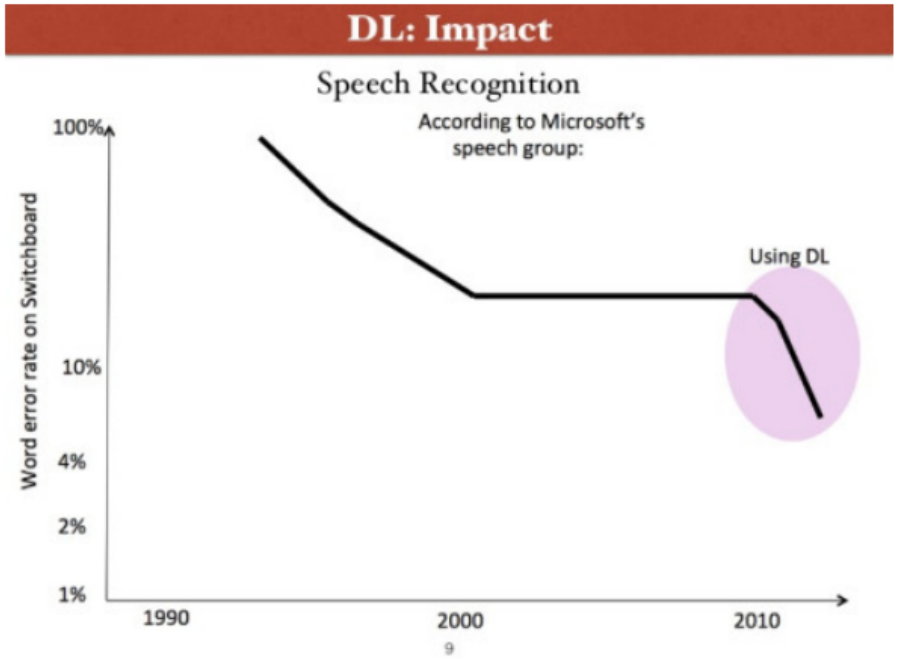
Figura 6: Red Neuronal Convolucional LeNet-5 para reconocimiento de dígitos [4]

En el año 2000 aparece el término de profundidad de las redes. Hasta ese momento únicamente se hablaba de redes neuronales. Geoffrey Hinton, investigador de la Universidad de Toronto, junto a Yann Le Cun y Bengio entre otros, trabajaron en conjunto para estudiar redes de múltiples capas. Si bien las redes neuronales ya habían demostrado ser de utilidad en diversos problemas de visión artificial, no habían logrado ser un algoritmo dominante, pues dentro de la inteligencia artificial aún persistían algoritmos más clásicos como el algoritmo de Viola & Jones[5].

Según [5] previo al año 2012, las principales razones de la falta de popularidad de las redes neuronales eran:

- Demasiados parámetros ($\mathbf{w}$) para aprender: en una red profunda pueden existir millones de parámetros a entrenar. Cuando se podría definir un conjunto significativamente menor de características o parámetros asociados a conocimiento experto sobre el problema. A modo de ejemplo, dentro de la visión artificial, se podrían definir características asociadas a la detección de bordes únicamente para el reconocimiento de objetos determinados.
- Optimización no convexa: estos problemas a resolver son no convexos. Lo que implica manejar el obstáculo de los mínimos locales en un problema de muy alta dimensión.
- Modelo de caja negra: aún logrando el aprendizaje de los numerosos parámetros asociados a este algoritmo, no era posible tener una interpretación razonable sobre lo que está haciendo. La falta de interpretación, con frecuencia genera inseguridad en los usuarios por lo que es un factor que ayuda a la falta de interés en su utilización.
- Dificultad para reproducir/obtener resultados: algo importante era la gran dificultad que existía respecto al entrenamiento de las redes. No había librerías adecuadas para entrenar redes neuronales en esos años.
- Estimaciones puntuales: otro elemento de crítica respecto a estos modelos era que a partir de una entrada se obtenía una salida puntual. Es decir que no se podía obtener un resultado probabilístico.

Hubo dos hechos puntuales que generaron un cambio drástico respecto al uso de las redes neuronales [5]. El primero fue protagonizado por Geoffrey Hinton, quien en 2009 desarrolló un algoritmo de reconocimiento de voz en teléfonos, mediante redes neuronales que según fue publicado, tuvo el mejor desempeño respecto a todos los competidores de ese momento. Como se muestra en la figura 7 a partir de 2010 se observó un quiebre en la performance de los algoritmos de reconocimiento de voz en teléfonos a partir de la publicación de Hinton.



(imagen de R. Pieters)

Figura 7: Evolución de la performance de algoritmos en el reconocimiento de voz [5]

Finalmente, en 2012 las redes neuronales se consolidaron como el algoritmo con mejor desempeño en el área de la visión artificial luego de una competencia llamada *Large Visual Recognition Competition* la cual implicaba acertar la categoría de una imagen. Se brindaba un conjunto de entrenamiento de más de un millón de imágenes (1.2 millones) conformando un total de 1000 categorías posibles. Se permitía seleccionar 5 categorías posibles y si dentro de estás se encontraba la categoría correcta de la imagen, se consideraba un acierto. El primer puesto utilizó redes neuronales profundas. Se puede apreciar en el Cuadro 1 un resultado que supera al resto de los algoritmos que no usaban redes neuronales, por más de 10 puntos porcentuales [5].

En las siguientes dos instancias de la competencia, la mayoría de los competidores que llegaron a los primeros puestos utilizaban redes neuronales profundas (ver Cuadro 2). Se puede notar que en tan solo tres años se pasó de un error porcentual de 26.1 sin utilizar redes neuronales, a un total de 6.6 utilizando redes neuronales profundas.

Otra forma de apreciar la evolución y el impacto de las redes neuronales profundas, es observar la mejora en la performance de los algoritmos al aumentar la profundidad de las redes. Se puede observar en la figura 8 que en los años 2013, 2014 y 2015 cómo mejora el error

Participantes 2012	% Error
Supervision (Toronto)	15.3
ISI (Tokyo)	26.1
VGG (Oxford)	26.9
xRCE INRIA	27
UvA (Amsterdam)	29.6
INRIA LEAR	33.4

Cuadro 1: Resultado de los primeros puestos de la competencia Large Scale Visual Recognition Competition en 2012

Participantes 2013	**% Error**	**Participantes 2014**	**% Error**
Clarifai (NYU)	11.7	GoogleNet	6.6
NUS (singapore)	12.9	VGG	7.3
Zeiler-Fergus (NYU)	13.5	MSRA	8.0
A Howard	13.5	A. Howard	8.1
OverFeat	14.1	DeeperVision	9.5
UvA (Amsterdam)	14.2	NUS-BST	9.7
Adobe	15.2	TTIC-ECP	10.2
VGG	15.2	**x**YZ	11.2
VGG (Oxford)	23.0	UvA	12.1

Cuadro 2: Resultado de los primeros puestos de la competencia Large Scale Visual Recognition Competition en 2013 y 2014

y su relación con el aumento en la profundidad de las redes. Se observa que pasar de un error porcentual de 6.7 a 3.57, requirió pasar de una red de 22 capas de profundidad a una de 152. Esto muestra que mejorar el error en el entorno de valores pequeños requiere redes profundas.

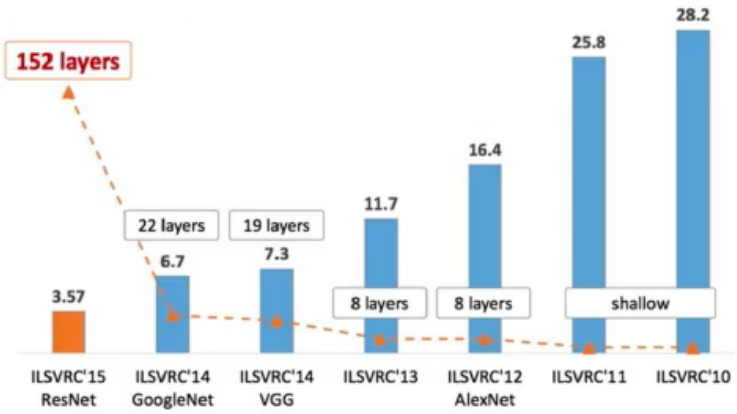


Figura 8: Resultado de los primeros puestos de la competencia Large Scale Visual Recognition Competition entre 2010 y 2015 [5]

La pregunta que el lector podría hacerse es qué cambió para que se pudieran utilizar estos

modelos de forma eficiente. Las principales razones son:

- Grandes bases de datos: se cuenta con enormes volúmenes de datos. Large Scale Visual Recognition Competition brindaba más de un millón de imágenes de entrenamiento.
- Aumento en la capacidad de procesamiento: las GPU en particular permiten hacer operaciones asociadas a las redes neuronales de manera eficiente.
- Mejores arquitecturas de red: surgieron otras formas de definir redes que empíricamente demostraron tener mejor desempeño.

2.2. Redes neuronales prealimentadas

Como se indica en [8] las redes neuronales prealimentadas, "feedforward" en inglés, también conocidas como perceptrones multicapa son una base fundamental de los modelos de aprendizaje profundo. El objetivo es aproximar cualquier función genérica f^*. Por ejemplo, supongamos f^* es una función que mapea un vector de entrada $\mathbf{x}$ a una salida y siendo esta una variable categórica, es decir $y = f^*(\mathbf{x})$. Se desea aproximar dicha función mediante una red neuronal de la forma $\hat{y} = f(\mathbf{x}; \hat{\theta})$ de manera tal que $\hat{y} \approx y$ [8].

Estos modelos se denominan prealimentados [8] dado que la información fluye a través de la función que se evalúa desde el vector de entrada $\mathbf{x}$, a través de un conjunto de cálculos intermedios para finalmente obtener una salida. Esta salida no vuelve a utilizarse dentro de la red. El término redes se emplea debido a que normalmente se representan como una composición de funciones. El modelo está asociado con un gráfico acíclico dirigido que describe cómo se componen las funciones. Como se presenta en [8], se puede definir una red neuronal prealimentada de n capas, a partir de la siguiente expresión:

$$f(\mathbf{x}, \theta) = h^{(n)}(h^{(n-1)}(\dots h^{(2)}(h^{(1)}(\mathbf{x}, \theta^{(1)}) \ , \theta^{(2)}) \ \dots, \theta^{(n-1)}) \ , \theta^{(n)})$$

En donde $h^{(1)}$ es la primera capa de la red, $h^{(2)}$ la segunda y así sucesivamente. Diremos que la cantidad de capas de nuestra red determinarán la profundidad de la misma. La primera capa se denomina de entrada por ser la única en contacto con los datos ($\mathbf{x}$). La última capa se denomina de salida ya que se observa el resultado del modelo. Por último, denominamos ocultas aquellas capas intermedias donde no se observan resultados. [8]. El término $\theta = \{\theta^{(1)}, \theta^{(2)}, \dots, \theta^{(n)}\}$ representa al conjunto de parámetros de la red, siendo $\theta^{(i)}$ el conjunto de parámetros de la capa i.

El término "neuronales" se inspira en la neurociencia. Cada capa oculta de la red se compone de un vector; a su vez, la dimensión de cada capa determina el ancho de la misma y cada elemento del vector se interpreta como una neurona. De esta forma, en lugar de pensar en una capa formada por un vector, se puede pensar como una capa compuesta por un conjunto de neuronas artificiales. Cada una de ellas, se parece a una neurona en el sentido de que recibe información de otras semejantes y calcula su propio valor de activación. Para este cálculo, las neuronas artificiales están formadas por funciones lineales compuestas por una función no lineal, modelo inspirado en cómo funcionan las neuronas del sistema nervioso [5]. Como se ve en la figura 9, cada neurona tiene un conjunto de entradas llamadas "dendritas", donde la neurona recibe "estímulos" y en función de esta información, se activa o no. Si se activa, la neurona dispara un estímulo cuya salida se denomina "axón". Esto sucede en función de las

entradas.

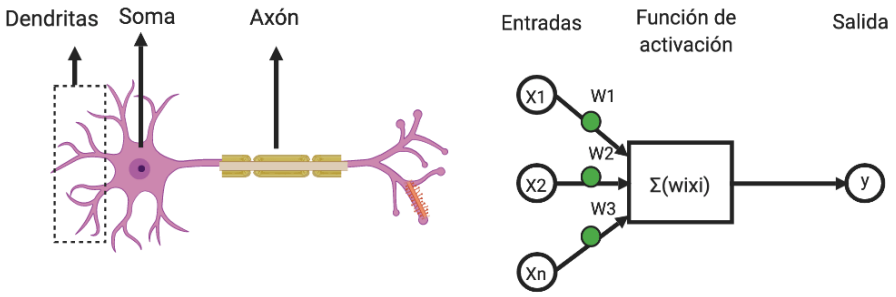


Figura 9: Neurona del sistema nervioso y neurona artificial de una red neuronal [5]

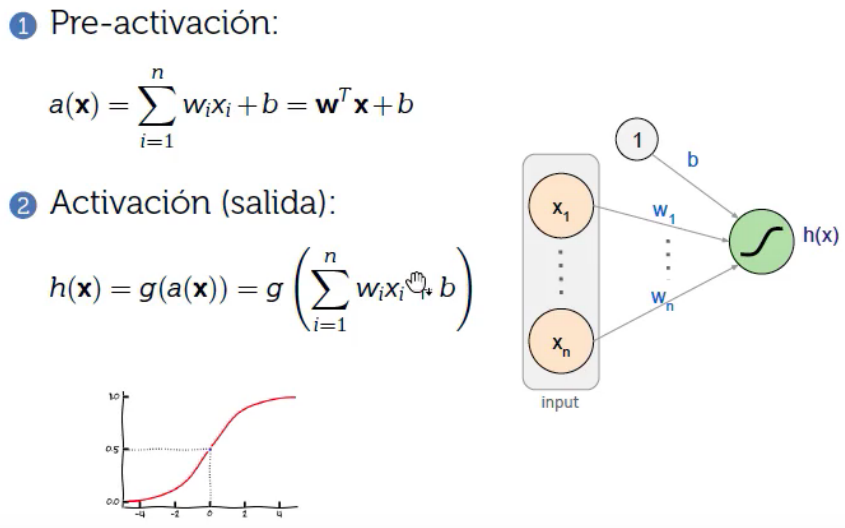


Figura 10: Neurona de una red neuronal [5]

Las neuronas artificiales tiene un comportamiento más simple. Existe una parte lineal y otra no lineal como se muestra en la figura 10. La función $a(x)$ corresponde a la parte lineal, la cual se denomina pre-activación de la neurona. Se calcula como el producto escalar $\mathbf{w}^T\mathbf{x} + b$. La segunda etapa implica la aplicación de una función no lineal g al resultado anterior. Esta segunda etapa es la denominada activación de la neurona.

Funciones de Activación

Las funciones de activación definen la forma en que se aplica el componente no lineal en la etapa de activación de una neurona. Existen distintas funciones y éstas deben seguir cierto requisitos según [5]. El criterio de selección de estas funciones se basa en la capacidad que poseen para permitir el entrenamiento de una red. Estas funciones deben ser diferenciables y no lineales, tomando como entrada un número real. Debe ser diferenciable, dado que esta propiedad es necesaria para aplicar la propagación hacia atrás ("backpropagation" en inglés), algoritmo fundamental para entrenar y actualizar los parámetros de una red neuronal. Esto

se verá con mayor profundidad más adelante en la presentación del algoritmo de "Backpropagation". La no linealidad garantiza específicamente que la composición de las capas no se reduzcan a una función lineal, permitiendo a la red modelar patrones más complejos. Se puede utilizar cualquier función personalizada como activación para un modelo de aprendizaje profundo siempre que se asegure que sea diferenciable y no lineal.

A continuación, se presentan cuatro funciones de activación de uso frecuente. La primera se denomina ReLU, la segunda es una variante llamada Leaky ReLU y las dos últimas se denominan Sigmoide y Tangente Hiperbólica. Hay un gran número de posibles funciones de activación. Cada investigador prueba cuál le resulta más eficiente según el contexto puntual en el que esté trabajando.

ReLU

Una de las funciones de activación más populares se conoce como ReLU, por sus siglas en inglés de "Rectifier Linear Unit". Esta función toma el máximo entre cero y un valor que llamaremos x. Su expresión es la siguiente:

$$g(x) = \text{máx}(0, x)$$

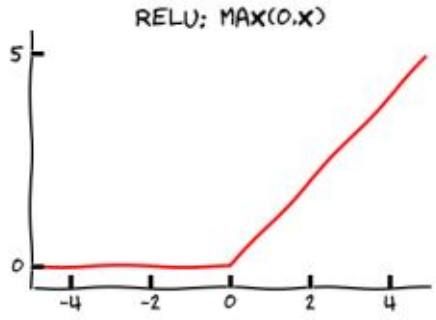


Figura 11: Función ReLu - Rectifier Linear Unit [5]

Se puede notar que ReLU estrictamente hablando no es diferenciable en x. Por convención e implementación, la derivada de ReLU en $x = 0$.

Como se observa en la figura 11 la parte plana de la función de activación de ReLU cuando x es negativo siempre tiene una derivada igual a cero. Esto puede generar problemas a la hora del entrenamiento de la red. Con una derivada igual a cero, algunos nodos se atascan en el mismo valor y sus pesos dejan de actualizarse. Podría interpretarse como que estos nodos dejan de aprender. De hecho, los componentes anteriores de la red también se verán afectados, este problema se conoce como el problema de la muerte de ReLU ("Dying ReLu Problem" en inglés) [5] porque es el final del aprendizaje.

Leaky ReLU

Leaky ReLU es una variación de ReLu que mantiene la misma forma que ReLU para el caso en el que x es positivo, pero agrega una pequeña pendiente en la línea cuando x es negativo.

Su expresión es la siguiente:

$$g(x) = \text{máx}(a.x, x)$$

En general a tiene un valor entre 0 y 1, como ser 0.05. Esto resuelve el problema de la muerte de ReLU. Al igual que en la función ReLu, se puede notar que Leaky ReLU estrictamente hablando no es diferenciable en $x = 0$. Y por convención e implementación, también la derivada de Leaky ReLU en $x = 0$ es igual a cero.

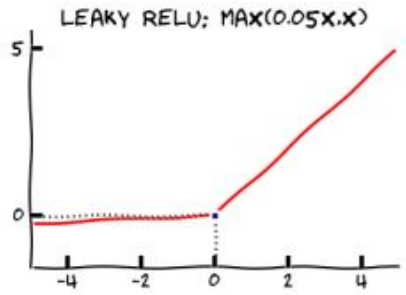


Figura 12: Función Leaky ReLu con a = 0.05 [5]

Sigmoide

Esta función tiene una forma suave y genera valores entre 0 y 1. Cuando x es mayor o igual a cero la función genera un valor entre 0.5 y 1. Cuando x es menor que cero genera un valor entre 0 y 0.5. Debido a que genera un valor entre 0 y 1, la función de activación Sigmoide se utiliza a menudo en modelos de clasificación binaria en la última capa para indicar una probabilidad.

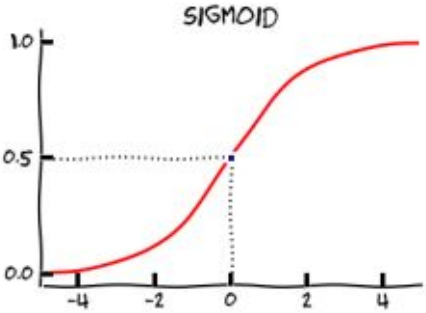


Figura 13: Función Sigmoide [5]

Sin embargo, esta función no se usa muy a menudo en capas ocultas, pues la derivada de la función tiende a cero en las colas. Se aproxima asintóticamente a uno en la parte superior

y asintóticamente a cero en la parte inferior. Esto produce el denominado problema de desvanecimiento del gradiente el cual se detalla más adelante[5].

Tangente hiperbólica

Otra función con una forma similar a la Sigmoide es la Tangente hiperbólica o tanh para abreviar. Sin embargo, en contraste con la función Sigmoide, genera valores entre -1 y 1. Cuando x es positivo, genera un valor entre 0 y 1 y cuando x es negativo, genera un valor entre -1 y 0.

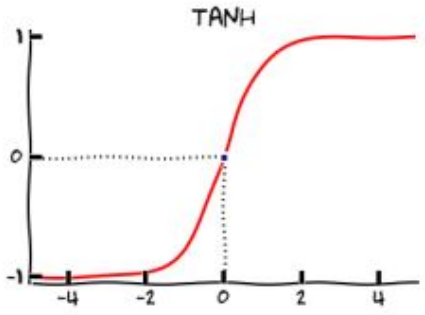


Figura 14: Función Tangente hiperbólica [5]

Una diferencia clave con la función Sigmoide es que la función Tangente hiperbólica mantiene el signo de la entrada x. Eso puede resultar útil en algunas aplicaciones, sin embargo ocurre el mismo problema de saturación y desvanecimiento del gradiente.

Red prealimentada de una capa oculta

Ya se ha presentado una definición de una red neuronal prealimentada, el concepto de neurona artificial y cómo funciona. A continuación, se analiza un ejemplo de red neuronal prealimentada de una única capa oculta [5].

La red consta de un vector de entrada $\mathbf{x} \in R^n$, una capa oculta compuesta por una cantidad d de neuronas y una capa de salida compuesta por una única neurona que recibe información de la capa oculta. El vector de entrada calcula la pre-activación a través de una transformación lineal:

$$a(\mathbf{x}) = W^{(1)T}\mathbf{x} + \mathbf{b}^{(1)}$$

Esa transformación consta de una matriz $W^{(1)T}$ de dimensión $(d\ ,\ n)$, el vector de entrada $\mathbf{x}$ de dimensión $(n\ ,\ 1)$ y el vector de sesgo $\mathbf{b}^{(1)}$, de dimensión $(d\ ,\ 1)$.

- Entrada: $\mathbf{x}$
- Pre-activación:

$$\mathbf{a} = \mathbf{W}^{(1)T}\mathbf{x} + \mathbf{b}^{(1)}$$

- Activación capa oculta:

$$\mathbf{h}^{(1)}(\mathbf{x}) = g(\mathbf{a}(\mathbf{x}))$$

- Capa de salida:

$$f(\mathbf{x}) = o\left(\mathbf{w}^{(2)T}\mathbf{h}^{(1)} + b^{(2)}\right)$$

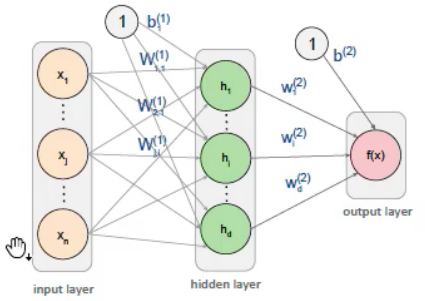


Figura 15: Ejemplo de procesamiento de imagen con una red prealimentada [5]

$$W^{(1)T} = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} & \dots & w_{1n}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} & \dots & w_{2n}^{(1)} \\ \vdots & \vdots & \vdots & \vdots \\ w_{j1}^{(1)} & w_{j2}^{(1)} & \dots & w_{jn}^{(1)} \\ \vdots & \vdots & \vdots & \vdots \\ w_{d1}^{(1)} & w_{d2}^{(1)} & \dots & w_{dn}^{(1)} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \mathbf{b}^{(1)} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_d \end{bmatrix}$$

La fila j de la matriz $W^{(1)T}$ corresponde al conjunto de entradas que recibe la neurona h_j. Por ejemplo la fila 1, corresponde a todas las entrada que recibe la neurona h_1. En los índices de cada elemento $w_{ji}^{(1)}$: j indica la neurona destino, i el elemento del vector de entrada $\mathbf{x}$ a la neurona $h_i^{(1)}$ y el supraíndice igual a 1 indica la capa de la red neuronal.

Como se indica en la figura 15, en la etapa de activación de esta red, se aplica una función no lineal a la salida de la pre-activación $h^{(1)}(\mathbf{x}) = g(a(\mathbf{x}))$, resultando en la salida de la capa 1 de dimensiones $(d\ ,\ 1)$, que se multiplica por una matriz de pesos $W^{(2)T}$ de tamaño $(1\ ,\ d)$ y se pasa a la capa de salida de la red. Finalmente, en la capa de salida se aplica una transformación lineal seguido de aplicar otra función de activación de la forma $f(\mathbf{x}) = o(W^{(2)T}h^{(1)}(\mathbf{x}) + b^{(2)})$. Según se explica en [5] en general en la capa de salida la función de activación suele ser la función identidad. Es decir, que no se aplica ninguna función no lineal.

Resolviendo la función XOR

Para brindar un ejemplo concreto de aplicación de una red prealimentada sencilla, en esta sección se resolverá la función XOR. Anteriormente se planteó en la sección 2.1 la dificultad del modelo de Perceptrón para resolver esta función. Aquí se plantea nuevamente el problema de clasificación de esta función, demostrando su solución mediante una red neuronal prealimentada de una capa oculta de dos neuronas [8].

La función XOR o también llamado OR exclusivo toma como entrada dos valores binarios x_1 y x_2. Cuando exactamente una de estas variables toma el valor 1, la función XOR vale 1, de lo contrario vale 0. Por lo tanto, XOR será la función objetivo que se denotará como $y = f^*(x)$. A nuestro modelo lo denominaremos $\hat{y} = f(x, \Theta)$ y será la función que aproxime a la función objetivo. Para ello, aplicaremos un modelo representado por una red neuronal y un algoritmo de aprendizaje que permita encontrar los valores adecuados para los parámetros incluidos en Θ. Vale destacar que en este ejemplo no es de interés que el modelo generalice para nuevos datos, sino que simplemente la red neuronal clasifique correctamente los 4 valores de entrada $X = [0,0]^T, [0,1]^T, [1,0]^T, [1,1]^T$ para nuestra función objetivo XOR. Por lo que una vez definida la estructura de la red, se debe entrenar para estos 4 puntos y en cada caso lograr ajustar correctamente el valor de salida. Se puede ver como un problema de regresión y utilizar como función de pérdida el error cuadrático medio (MSE por sus siglas en inglés "Mean Square Error"). Se utilizará esta función por simplicidad debido a que se alcanza una solución. Pero en la práctica no se suele utilizar el error cuadrático medio como función de costo en problemas con datos binarios.

La función de costo MSE evaluado en los 4 puntos de interés, tiene la siguiente expresión:

$$L(\Theta) = \frac{1}{4} \sum_{x \in X} (f^*(x) - f(x, \Theta))^2$$

Se demostró en la sección 2.1 que no es posible resolver el problema mediante un modelo lineal. La solución en este caso será introducir una red prealimentada con una capa oculta de dos neuronas. La figura 16 muestra la representación de la red a utilizar en la resolución del problema.

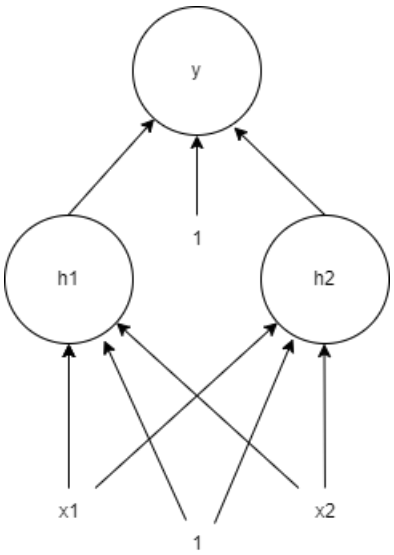


Figura 16: Representación de una red prealimentada con una capa oculta de dos neuronas

La capa oculta recibe los datos de entrada representado por $\mathbf{x} = (x_1, x_2)$ y se representará mediante una función que denominaremos $f^{(1)}(\mathbf{x}, \theta^{(1)}, c^{(1)})$. La salida de esta función será entrada de la capa de salida, quien dará el resultado final. La capa de salida recibe el valor de $f^{(1)}(\mathbf{x}, \theta^{(1)}, c^{(1)})$ y aplicará una transformación que llamaremos $f^{(2)}(f^{(1)}(\mathbf{x}, \theta^{(1)}, c^{(1)}), \theta^{(2)}, b^{(2)})$

teniendo así nuestra red neuronal expresada como una composición de funciones. Como aclaración, el conjunto total de parámetros será $\Theta = (\theta_1^{(1)}, \theta_2^{(2)}, b^{(2)}, c^{(1)})$ que es la unión de todos los parámetros de cada capa. Los parámetros $b^{(2)}$ y $c^{(1)}$ son valores de sesgo en cada una de las capas. El conjunto de parámetros de la capa 1 estará dado por: $c^{(1)}$ que es una matriz de tamaño 2x1 y $\theta^{(1)}$ una matriz 2x2 de la forma:

$$\theta^{(1)} = \begin{bmatrix} \theta_{11}^{(1)} & \theta_{12}^{(1)} \\ \theta_{21}^{(1)} & \theta_{22}^{(1)} \end{bmatrix} \text{ y } c^{(1)} = \begin{bmatrix} c_1^{(1)} \\ c_2^{(1)} \end{bmatrix}$$

La transformación lineal que se aplica en la capa 1 toma los datos de entrada $\mathbf{x} = (x_1, x_2)$ y se expresa como: $(\theta^{(1)})^T\mathbf{x} + c^{(1)}$. Sin embargo, como se vio en la demostración del Perceptrón, no es posible resolver este problema mediante una función lineal, por lo que es necesario aplicar una función no lineal para resolverlo. Por ello, se define:

$$f^{(1)}(\mathbf{x}, \theta^{(1)}, c^{(1)}) = g((\theta^{(1)})^T\mathbf{x} + c^{(1)}),$$

siendo $g()$ una función no lineal. El conjunto de parámetros de la capa 2, estará dado por: $b^{(2)}$ que es una matriz de 1x1 y $\theta^{(2)}$ que es una matriz 2x1 de la forma:

$$\theta^{(2)} = \begin{bmatrix} \theta_1^{(2)} \\ \theta_2^{(2)} \end{bmatrix} \text{ y } b^{(2)} = \begin{bmatrix} b \end{bmatrix}$$

En la capa 2, que es la capa de salida, aplicaremos únicamente una transformación lineal que será de la forma:

$$\hat{y} = f^{(2)}(\mathbf{h}, \theta^{(2)}, b^{(2)}) = (\theta^{(2)})^T\mathbf{h} + b^{(2)},$$

donde $\mathbf{h} = f^{(1)}(\mathbf{x}, \theta^{(1)}, c^{(1)}) = g((\theta^{(1)})^T\mathbf{x} + c^{(1)})$. Para finalizar la definición de la red, se utilizará la función ReLU como función de activación no lineal. Es decir que tendremos que $g(x) = max\{0, x\}$.

Ahora es posible dar una expresión completa de la red como se muestra a continuación:

$$\hat{y} = f(\mathbf{x}, \theta^{(1)}, c^{(1)}, \theta^{(2)}, b^{(2)}) = (\theta^{(2)})^T max\{0, (\theta^{(1)})^T\mathbf{x} + c^{(1)}\} + b^{(2)}$$

Finalmente, como solución al problema de la función XOR se especifican las siguientes asignaciones:

$$\theta^{(1)} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \; c^{(1)} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \; \theta^{(2)} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \; b^{(2)} = \begin{bmatrix} 0 \end{bmatrix}$$

Los datos de entrada se definen dentro de una matriz de 2x4 como sigue:

$$\mathbf{x} = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

En primer lugar calculamos:

$$
\begin{aligned}
(\theta^{(1)})^T\mathbf{x} + c^{(1)} &= \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix} \\
&= \begin{bmatrix} 0 & 1 & 1 & 2 \\ 0 & 1 & 1 & 2 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix} \\
&= \begin{bmatrix} 0 & 1 & 1 & 2 \\ -1 & 0 & 0 & 1 \end{bmatrix}
\end{aligned}
$$

A continuación, es preciso aplicar la función no lineal ReLU que tiene la expresión $g(x) = max\{0, x\}$ a la transformación lineal $(\theta^{(1)})^T\mathbf{x} + c^{(1)}$, obteniéndose el siguiente resultado:

$$
g((\theta^{(1)})^T\mathbf{x} + c^{(1)}) = g(\begin{bmatrix} 0 & 1 & 1 & 2 \\ -1 & 0 & 0 & 1 \end{bmatrix}) = \begin{bmatrix} 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

Finalmente resta hacer la operación

$$
(\theta^{(2)})^T g((\theta^{(1)})^T\mathbf{x} + c^{(1)}) + b^{(2)} = \begin{bmatrix} 1 & -2 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 & 0 \end{bmatrix}
$$

Se obtuvo el resultado correcto para cada uno de los 4 puntos de la función XOR.

Teorema de aproximación universal

"Una red prealimentada de una capa oculta, con un número finito de neuronas y función de activación sigmoide puede aproximar cualquier función continua en un espacio compacto de R^n." [9]

Si bien este teorema muestra el gran potencial de las redes neuronales, no define cómo obtener los parámetros óptimos que debería tener el modelo. Kurt Hornik demostró en 1991 [9] que no es la elección específica de la función de activación, sino la propia arquitectura de prealimentación de múltiples capas la que brinda a las redes neuronales el potencial de ser aproximaciones universales. Moshe Leshno et al. en 1993 [10] y más tarde Allan Pinkus en 1999 [11] demostraron que la propiedad de aproximación universal, [12] es equivalente a tener una función de activación no polinómica.

Red prealimentada de más de una capa oculta

A continuación se presenta un ejemplo más general de una red neuronal prealimentada con más de una capa oculta.

En la figura 17 se puede observar un ejemplo de una red neuronal prealimentada de 3 capas, o bien de 2 capas ocultas. El funcionamiento de las redes es análogo al visto en el ejemplo de una sola capa. A partir de un vector de entrada $\mathbf{x}$, se aplica una pre-activación mediante una transformación lineal utilizando los pesos y el sesgo de la primera capa. Se aplica una función de activación y su salida será el nuevo vector de entrada de la capa siguiente, que ejecutará

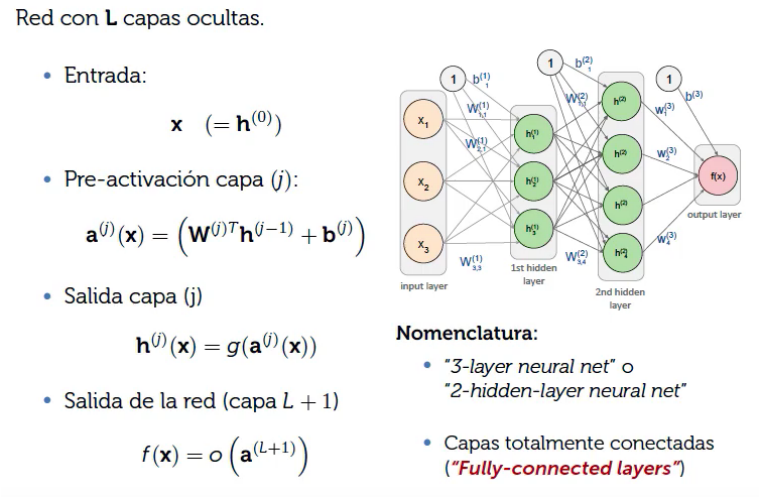


Figura 17: Red prealimentada de más de una capa oculta [5]

un proceso similar y así sucesivamente. Cuando una neurona se conecta con cada elemento de la capa anterior, se dice que la red tiene capas "totalmente conectadas" o "Fully Connected" en inglés.

Hasta ahora fueron definidos ejemplos de redes neuronales, pero aún no se han definido estrategias que determinen la forma de entrenarlas.

2.3. Entrenamiento de una red neuronal

Supongamos que tenemos definida una red neuronal de n capas de la forma.

$$f(\mathbf{x}\ , \theta) = f^{(n)}(h^{(n-1)}(\dots h^{(2)}(h^{(1)}(\mathbf{x}\ , \theta^{(1)}), \theta^{(2)}) \dots , \theta^{(n)})$$

Entrenar una red consta de encontrar los valores de $\hat{\theta}$ de forma de minimizar el siguiente problema [5]:

$$\hat{\theta} = arg\ min_{\theta} \sum_{i=1}^{n} L(f(x_i; \theta); y_i)$$

Donde $\theta = (\theta^{(1)}, \theta^{(2)}, \dots, \theta^{(n)})$ representa el conjunto de parámetros de la red y L es una **función denominada función de costo o de pérdida** que medirá las discrepancias entre lo predicho (la clasificación asignada) por el modelo y las etiquetas reales de los elementos de entrada representadas por y_i. El método de optimización utilizado para resolver el problema es el denominado **Descenso por Gradiente** [5] y el algoritmo para calcularlo es denominado **BackPropagation** el cual se verá más adelante [5].

En un contexto de clasificación, las redes neuronales pre alimentadas tienen por objetivo resolver un problema de optimización que busca minimizar discrepancias entre un resultado predicho por el modelo y la etiqueta real. Se parte del conjunto:

$$\{(\mathbf{x}_1, y_1)\}, \{(\mathbf{x}_2, y_2)\}, \dots, \{(\mathbf{x}_n, y_n)\},\ \text{donde } \mathbf{x}_i \in \mathbb{R}^d,\ y_i \in \{c_1, c_2, \dots, c_k\}$$

Nuestro modelo deberá aprender a predecir correctamente la etiqueta y_i a partir de un conjunto finito de datos de entrada representado por los vectores $\mathbf{x}_i$, denominado conjunto de datos de entrenamiento; y a partir de un procesamiento adecuado de esta información, frente a un nuevo dato de entrada digamos $\mathbf{x}_k$ poder predecir su etiqueta $\hat{y_k}$. Este proceso de aprendizaje se denomina **aprendizaje supervisado**.

En el proceso del aprendizaje supervisado se requieren cuatro componentes [5]:

- el conjunto de datos de entrada representado por los pares $(\mathbf{x}_i, y_i)$.
- definir una familia de posibles clasificadores $\mathcal{F}$. Esta familia de funciones dependerá de un conjunto de parámetros θ.
- una medida de discrepancia entre lo predicho y lo real. Esta es la función L. Algunos ejemplos de esta función pueden ser: BCE, Wasserstein, entre otras, las cuales se estudiarán más adelante.
- un método de optimización para encontrar un modo de elegir la función $f \in \mathcal{F}$ de forma de minimizar:
$$\frac{1}{n}\sum_{i=1}^{n} L(f(\mathbf{x}_i;\theta);y_i)$$

2.4. Regularización

El desafío central que enfrentan los modelos en el área de aprendizaje automático y, en particular dentro del aprendizaje supervisado, es la capacidad de generalización. Esta capacidad está directamente relacionada con la capacidad de predicción del modelo frente a nuevos datos de entrada. Estos modelos se basan en un conjunto de datos al cual se ajustan para luego predecir la salida correspondiente a nuevos datos. Un problema frecuente en las redes neuronales, es que logran un gran desempeño al ajustarse a los datos de entrenamiento, pero no frente a datos nuevos; esta situación indica sobreajuste del modelo. Por otro lado, si no se logra un buen ajuste en el conjunto de entrenamiento, el modelo tendrá poca o ninguna capacidad de generalizar lo que se denomina subajuste. Por lo que es en este sentido la necesidad de buscar un equilibrio en el ajuste del modelo.

Muchas estrategias utilizadas en el aprendizaje automático están diseñadas explícitamente para reducir el error del modelo frente nuevos datos posiblemente a expensas de un mayor error en el conjunto de entrenamiento. Estas estrategias se conocen colectivamente como **regularización** [8]. La función L se utiliza a efectos de lograr un buen ajuste a los datos de entrenamiento y debe permitir que el modelo tenga un buen desempeño frente a datos no conocidos por el modelo[5].

Penalización de norma de parámetros del modelo

Para lograr el equilibrio entre el error en los datos de prueba y el error en los datos de entrenamiento, una estrategía consiste en limitar la capacidad del modelo agregando un nuevo término a la función de costo que llamaremos término de regularización [8] que será una función que dependerá de los parámetros del modelo [5].

La nueva función de costo queda expresada a continuación [8]:

$$\tilde{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n} L(f(x_i;\theta);y_i) + \lambda R(\theta)$$

Donde $\lambda \in [0,\infty)$ es un hiperparámetro que pondera la contribución relativa del término de penalización de parámetros, representado por $\lambda R(\theta)$. Cuando $\lambda = 0$ significa que no se aplica regularización, mientras que un valor alto en λ corresponde a aplicar mayor regularización en la función objetivo.

Cuando el algoritmo de entrenamiento minimiza la función objetivo regularizada $\tilde{L}(\theta)$, se minimiza la función objetivo original $L(\theta)$ y en alguna medida el tamaño de los parámetros θ.

Algunos ejemplos de funciones de regularización que suelen ser utilizadas según [4]: Penalización L2, Penalización L1, entre otros. La elección de la función $R(\theta)$ tendrá un efecto diferente sobre los parámetros del modelo.

Antes de profundizar en el comportamiento de regularización de estas diferentes funciones, vale señalar que para las redes neuronales, normalmente se utiliza una penalización que afecta únicamente los parámetros de cada capa, dejando sin regularizar los parámetros correspondiente al sesgo. Estos parámetros suelen requerir menos datos para ajustarse con precisión. Cada parámetro de peso, especifica cómo interactúan dos variables. Ajustar estos parámetros, requiere observar ambas variables en una variedad de condiciones; sin embargo, cada sesgo controla sólo una variable. Esto significa que no se induce demasiada variación al dejar los sesgos sin regularizar. Por otro lado, la regularización de los parámetros de sesgo puede introducir subajuste en el modelo. Por lo que a continuación se denotará al conjunto de parámetros que excluye a los parámetros de sesgo, como el vector w, mientras que el vector θ representa a todo el conjunto de parámetros [8].

En el contexto de las redes neuronales, en ocasiones es deseable utilizar una penalización separada con un valor diferente de λ para cada capa de la red. Sin embargo, debido a que puede resultar costoso buscar el valor correcto de múltiples hiperparámetros, es razonable usar la misma disminución de peso en todas las capas y así reducir el tamaño del espacio de búsqueda.

Regularización L2

La regularización L2, también conocida como "Caída de pesos" (en inglés "Weight Decay", modifica o contrae el tamaño de los pesos hacia el origen del espacio al agregar un término $R(w) = \frac{1}{2}\|w\|^2$ a la función objetivo. L2 también se conoce como regularización de Ridge o regularización de Tikhonov [8].

Se puede obtener información respecto al comportamiento de esta función regularizada mediante el estudio de su gradiente. Se plantea a continuación la función objetivo regularizada L2 [8]:

$$\tilde{L}(w,\mathbf{x},y) = \frac{\lambda}{2}w^T w + L(w,\mathbf{x},y)$$

donde w es el conjunto de parámetros excluyendo aquellos de sesgo, $\mathbf{x}$ es un dato de entrada y finalmente y representa el valor real correspondiente a la entrada $\mathbf{x}$.

El gradiente correspondiente a la función $\tilde{L}(w, \mathbf{x}, y)$ es:

$$\nabla_w \tilde{L}(w, \mathbf{x}, y) = \lambda w + \nabla_w L(w, \mathbf{x}, y)$$

En un paso simple de actualización mediante el descenso por gradiente (ver anexo 6.1) se realiza la siguiente asignación:

$$w \leftarrow w - \epsilon(\lambda w + \nabla_w L(w, \mathbf{x}, y))$$

Lo anterior se puede reescribir como:

$$w \leftarrow (1 - \epsilon\lambda)w - \epsilon\nabla_w L(w, \mathbf{x}, y))$$

Se puede observar que la adición del término de penalización ha modificado la regla de aprendizaje para reducir el vector de peso en un factor constante en cada paso, justo antes de realizar la actualización de gradiente habitual. Esto describe lo que sucede en un solo paso.

Para analizar qué ocurre durante todo el resto del proceso de aprendizaje, se simplifica el análisis haciendo una aproximación cuadrática a la función objetivo en la vecindad del valor que minimiza la función objetivo sin regularizar, $w^* = argmin_w L(w)$. Si se hace una aproximación cuadrática en el punto w^*, se obtiene lo siguiente [8]:

$$\hat{L}(w) = L(w^*) + \frac{1}{2}(w - w^*)^T H(w - w^*)$$

donde H es la matriz Hessiana de L respecto de w evaluada en w^*. Se observa que no hay término de primer orden dado que w^* es el mínimo de $L(w)$. Dado que w^* es el mínimo de L, se puede concluir que H es una matriz semidefinida positiva.

El mínimo de $\hat{L}$ ocurre cuando $\nabla_w \hat{L}(w) = H(w - w^*) = 0$. Este término surge de derivar $\hat{L}(w)$ respecto de w y luego igualar a 0.

A continuación se plantea la versión regularizada de $\hat{L}$, obteniendo la siguiente expresión:

$$\hat{L}_{Reg}(w) = L(w^*) + \frac{1}{2}(w - w^*)^T H(w - w^*) + \frac{\alpha}{2} w^T w$$

Aplicando la derivada se obtiene la expresión del gradiente como sigue:

$$\alpha w + H(w - w^*)$$

Sea $\tilde{w}$ el mínimo de la función $\hat{L}_{Reg}(w)$, entonces se cumple que:

$$\alpha\tilde{w} + H(\tilde{w} - w^*) = 0$$

Ahora se despeja $\tilde{w}$ como sigue:

$$(H + \alpha I)\tilde{w} = Hw^* \quad \Rightarrow \quad \tilde{w} = (H + \alpha I)^{-1} Hw^*$$

A medida que α se acerca a 0, la solución regularizada $\tilde{w}$ se acerca a w^* [8]. A continuación se estudia el comportamiento cuando α crece. Dado que H es una matriz simétrica, se puede descomponer en sus valores y vectores propios como sigue:

$$H = Q\Lambda Q^T$$

Donde Q es una matriz ortonormal compuesta por los vectores propios de H y Λ es una matriz diagonal compuesta por los valores propios de H. Dado que la matriz Q es ortogonal, se cumple que $Q^T = Q^{-1}$, por lo tanto $Q^T Q = I$.
Sustituyendo H por $Q\Lambda Q^T$ y utilizando que $Q^T = Q^{-1}$, calculamos nuevamente $\tilde{w}$ como sigue:

$$\tilde{w} = (Q\Lambda Q^T + \alpha I)^{-1} Q\Lambda Q^T w^* = \left[(Q(\Lambda Q^T + \alpha I)Q^T)\right]^{-1} \left[Q\Lambda Q^T w^*\right] = Q(\Lambda + I\alpha)^{-1}\Lambda Q^T w^*$$

Se observa que el efecto de aplicar esta regularización es reescalar w^* en dirección a los ejes definidos por los vectores propios de H. Es decir que el componente de w^* asociado al i-ésimo vector propio de H es reescalado por un factor de $\frac{\lambda_i}{\lambda_i+\alpha}$. En las direcciones en donde los valores propios de H son relativamente grandes, por ejemplo cuando se tiene que $\lambda_i \gg \alpha$, el efecto de la regularización es relativamente pequeño. Por el contrario, cuando se tiene $\lambda_i \ll \alpha$, el componente se contrae de forma significativa. Este efecto se ilustra en la figura 18

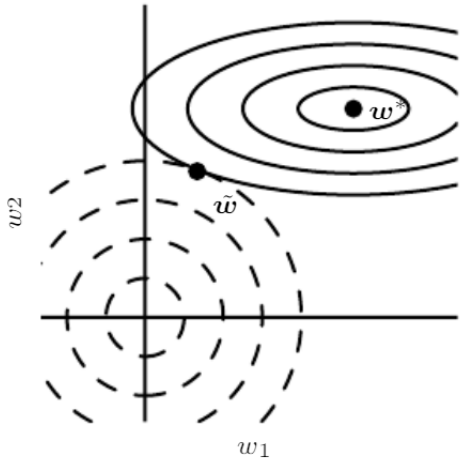


Figura 18: Ilustración del efecto de la regularización L2 sobre el valor óptimo w^* [8] cap.7. Las elipses sólidas representan contornos de igual valor del objetivo no regularizado. Los círculos punteados representan contornos de igual valor del regularizador. En el punto $\tilde{w}$, estos objetivos en competencia alcanzan un equilibrio. En la primera dimensión, el valor propio del Hessiano de L es pequeño. La función objetivo no aumenta mucho cuando se aleja horizontalmente de w^*. Debido a que la función objetivo no expresa una fuerte preferencia en esta dirección, el regularizador tiene un fuerte efecto en este eje. El regularizador acerca w_1 a cero. En la segunda dimensión, la función objetivo es muy sensible a los movimientos que se alejan de w^*. El valor propio correspondiente es grande, lo que indica una curvatura alta. Como resultado, la caída del peso afecta relativamente poco la posición de w_2.

Regularización L1

Otra forma de penalizar los parámetros del modelo es a partir de la Regularización L1 o también conocida como regularización Lasso [13]. Formalmente la regularización del modelo al parámetro w está definida por:

$$\Omega(\theta) = \|w\|_1 = \sum |x_i|$$

La regularización L1 funciona al agregar un término a la función de pérdida del modelo que penaliza los coeficientes de las características según su magnitud absoluta. Este término de penalización está dado por la suma de los valores absolutos de los coeficientes multiplicado por un parámetro λ que controla el grado de penalización [8].

La función de costos regularizada y su correspondiente gradiente son de la forma:

$$\tilde{L}(w, \mathbf{x}, y) = \lambda\|w\| + L(w, \mathbf{x}, y)$$

$$\nabla_w \tilde{L}(w, \mathbf{x}, y) = \lambda \; signo(w) + \nabla_w L(w, \mathbf{x}, y)$$

donde $signo(w)$ es simplemente el signo de w aplicado elemento a elemento.

Al examinar el gradiente se puede ver que el efecto de la regularización L1 es diferente al de la regularización L2. Específicamente, la contribución de la regularización al gradiente ya no escala linealmente con cada w_i; en cambio, es un factor de una constante por un signo (signo(w_i)) [8]. Una consecuencia de esta forma del gradiente es que no necesariamente veremos soluciones algebraicas claras para aproximaciones cuadráticas de $L(w, \mathbf{x}, y)$ como en la regularización L2.

El gradiente de la aproximación cuadrática de la función de costo es:

$$\nabla_w \hat{L}(w) = H(w - w^*)$$

donde H es la matriz Hessiana de la matriz L respecto a w evaluado en w^*. Para simplificar los cálculos se asume que H es una matriz diagonal de la forma $H = diag([H_{1,1}, \ldots, H_{n,n}])$ y $H_{i,i} > 0 \; \forall \; i$

La aproximación cuadrática de la función objetivo regularizada con L1 se descompone en una suma sobre los parámetros.

$$\hat{L}(w, \mathbf{x}, y) = L(w^*, \mathbf{x}, y) + \sum_i [\frac{1}{2} H_{i,i}(w_i - w_i^*) + \lambda |w_i|]$$

La solución al problema de minimización tiene un enfoque analítico para cada i, de la forma: $w_i = \text{signo}(w_i^*)\, \text{máx}\{|w_i^*| - \frac{\lambda}{H_{i,i}} \; , 0\}$

Cuando $w_i^* > 0$ para todos los i tenemos que:

- Si $w_i^* \leq \frac{\lambda}{H_{i,i}}$ el valor óptimo de w_i bajo el objetivo regularizado es simplemente $w_i = 0$. Esto ocurre porque la contribución de $L(w, X, y)$ al objetivo regularizado $\tilde{L}(w, X, y)$ es anulada en la dirección i, por la regularización L1 que empuja el valor de w_i hacia cero.

- Si $w_i^* > \frac{\lambda}{H_{i,i}}$ entonces la regularización no mueve el valor óptimo de w_i a cero, sino que simplemente lo desplaza en esa dirección por una distancia igual a $\frac{\lambda}{H_{i,i}}$.

Un proceso similar ocurre cuando $w_i^* < 0$, pero con la penalización L1 haciendo que w_i sea menos negativo por $\frac{\lambda}{H_{i,i}}$, o 0.

En comparación con la regularización L2, la regularización L1 produce una solución más dispersa. En este contexto, la dispersión se refiere al hecho de que algunos parámetros tienen un valor óptimo de cero. La dispersión resultante de la regularización L1 es un comportamiento cualitativamente diferente al que surge con la regularización L2 [8]. Por último, si volvemos a la ecuación utilizada en la suposición de H diagonal y definida positiva que introdujimos para nuestro análisis de la regularización L2, encontramos que $\tilde{w}_i = \frac{H_{i,i}}{H_{i,i}+\lambda} w_i^*$. Si w_i^* es distinto de cero, entonces $\tilde{w}_i$ permanece distinto de cero. Esto demuestra que la regularización L2 no causa que los parámetros se vuelvan dispersos, mientras que la regularización L1 puede hacerlo para λ lo suficientemente grande [8].

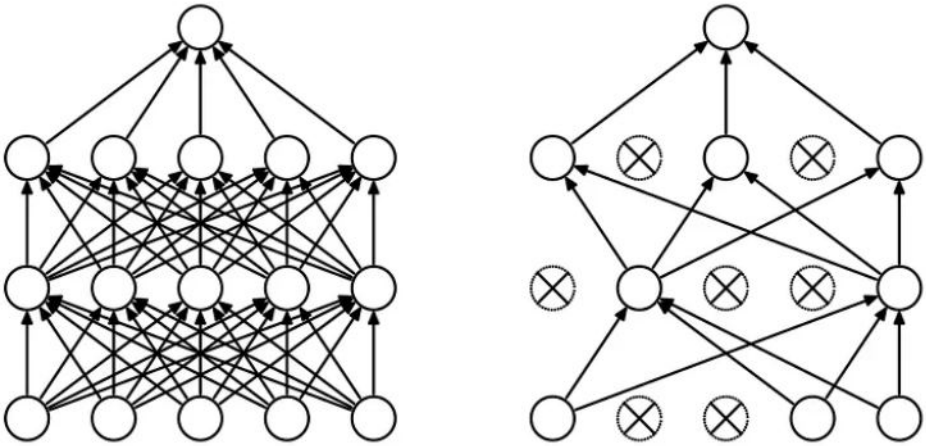
Método "Dropout"

Debido a que las redes profundas tienen un alto grado de poder expresivo, son capaces de sobreajustarse significativamente. Si bien este problema puede resolverse utilizando un conjunto de datos muy grande, los conjuntos de datos grandes no siempre están disponibles. El método de "Dropout" proporciona una solución computacionalmente económica pero poderosa para regularizar una amplia gama de modelos [8].

Esta solución refiere a la práctica de ignorar al azar distintos nodos en una capa de la red durante los pasos del proceso de entrenamiento. Este método busca simular el entrenamiento de varias redes neuronales con distintas arquitecturas en una única red [8]. Para evitar que una neurona se apoye de manera exclusiva en un conjunto pequeño de características del resultado de la capa anterior. Se evita el sobreajuste de la red al garantizar que ninguno de sus nodos sean codependientes entre sí. El método de "Dropout" consiste en llevar a 0 de manera aleatoria distintos valores del vector de entrada en cada una de las iteraciones de nuestro entrenamiento. Esto obliga a la neurona a aprender a dar respuestas válidas aún cuando varias de las características que entiende como más importantes no existen en esa iteración. Es común excluir nodos con una probabilidad entre el 20 % y el 50 % [8]. El método se implementa por capa; se puede aplicar sobre cualquier capa oculta de la red o incluso en la capa de entrada. No se puede utilizar en la capa de salida.

El proceso de entrenamiento procede utilizando los siguientes pasos, que se repiten una y otra vez para recorrer todos los puntos de entrenamiento en la red:

- Se selecciona una red neuronal de la red base. Los nodos de entrada se seleccionan cada uno con probabilidad p_i, y los nodos ocultos se seleccionan cada uno con probabilidad p_h. Además, todas las muestras son independientes entre sí. Cuando se elimina un nodo de la red, también se eliminan todas sus aristas incidentes.
- Se selecciona una única instancia de entrenamiento o un "mini-batch" de instancias de entrenamiento.



(a) Standard Neural Net (b) After applying dropout.

Figura 19: Ejemplo de aplicación del método dropout en una red neuronal [8]

- Se actualizan los pesos de las aristas retenidas en la red utilizando la retropropagación sobre la instancia de entrenamiento muestreada o el "mini-batch" de instancias de entrenamiento.

Una ventaja del método "Dropout" es que es muy económico en términos computacionales. Utilizarlo durante el entrenamiento solo requiere una computación de $O(n)$ en cada paso de actualización para generar n números binarios aleatorios y multiplicarlos por el estado a asignarle a la neurona, siendo n la cantidad de neuronas de toda la red. Dependiendo de la implementación, también puede requerir una memoria de $O(n)$ para almacenar estos números binarios hasta que la etapa de retropropagación termine. Predecir nuevas muestras con el modelo ya entrenado tiene el mismo costo computacional ya sea si se usara o no el método [8].

Srivastava et al. (2014) demostraron que el método "Dropout" es más efectivo que otros regularizadores estándar computacionalmente económicos, como la caída de peso, "filter norm constraints" y "sparse activity regularization" [8].

Normalización por lotes

Desde un punto de vista matemático las redes neuronales profundas se pueden ver como una composición de múltiples funciones o capas. La forma de cada función será determinada por los parámetros de cada capa. En cada paso de aprendizaje, el gradiente indica cómo actualizar los parámetros de una capa bajo la suposición de que las capas restantes no cambian. En la práctica, todas las capas se actualizan simultáneamente, pudiendo ocurrir resultados inesperados ya que muchas funciones compuestas se modifican simultáneamente[8].

La normalización por lotes proporciona una forma elegante de reparametrizar casi cualquier red profunda [8]. Esto reduce significativamente el problema de coordinar actualizaciones en múltiples capas. Se puede aplicar a cualquier capa de entrada u oculta en una red. En esencia, se calcula la media y la desviación estándar de las activaciones de una capa sobre el lote de datos de entrenamiento y luego se normalizan las activaciones restando la media y

dividiendo por la desviación estándar. Este proceso ayuda a mantener las activaciones en una escala razonable durante el entrenamiento, permitiendo utilizar tasas de aprendizaje más altas y ser menos cuidadoso con la inicialización de los pesos, obteniendo mejoras de velocidad de convergencia y estabilidad del modelo.

A continuación, se presenta un ejemplo sencillo que ilustra el funcionamiento de este método. Dada una red neuronal simple con dos variables de entrada x_1, que indica tamaño y x_2, que indica color. Su salida, una variable binaria que determina la presencia de un determinado objeto. Ambas variables tienen una distribución normal con diferente media y desviación estándar. Esta diferencia puede afectar la forma de aprendizaje de la red. Por ejemplo, al intentar alcanzar un mínimo local de la función de costos a partir de estas distribuciones como se muestra en la figura 21, puede resultar en un aprendizaje más difícil, lento y dependiente de la inicialización de los pesos [4].

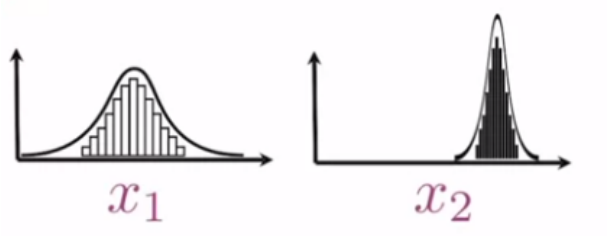


Figura 20: Ejemplo dos distribuciones normales con distinta media y desviación estándar [4].

Por otro lado, nuevos datos de entrenamiento con colores muy diferentes podría implicar un cambio en la distribución de los datos y en consecuencia en la forma de la función de costo. La ubicación de un mínimo local de la función podría moverse, aunque lo que determina la presencia o no del objeto, permanezca sin cambios. Este fenómeno se conoce como cambio covariable y ocurre con bastante frecuencia en los conjuntos de entrenamiento y prueba donde no se han tomado precauciones sobre la distribución de los datos [4].

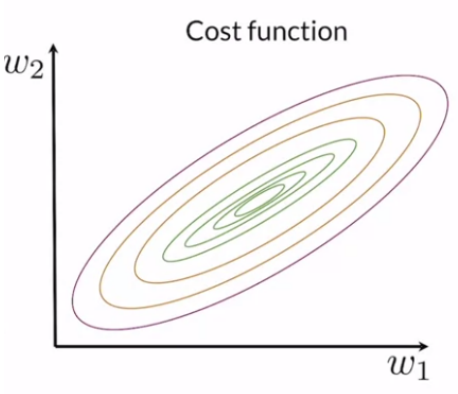


Figura 21: Ejemplo función de costos de forma elíptica [4]

Al normalizar las variables de entrada x_1' y x_2' la función de costo tendrá una forma más suave y equilibrada entre estas dos dimensiones, dando como resultado un entrenamiento potencialmente más rápido y sencillo.

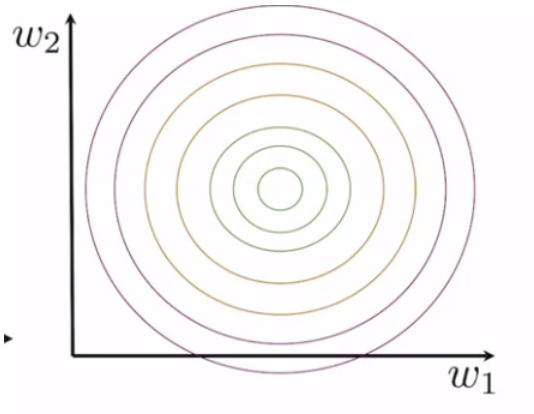


Figura 22: Batch Normalization - Función de Costo Normalizada [4]

A fin de entender cómo funcionan las operaciones de normalización por lotes y cómo se diferencian durante las etapas de entrenamiento y prueba, se presenta un ejemplo. Sea una red neuronal que cuenta con dos capas ocultas, múltiples entradas y una única salida.

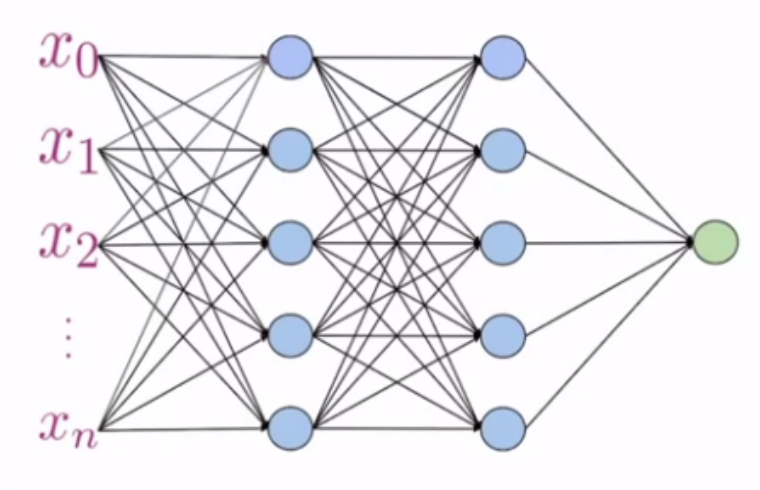


Figura 23: Ejemplo de una Red neuronal de 2 capas ocultas [4]

Se estudiará la primera capa oculta interna para explicar el funcionamiento de la normalización por lotes respecto de una neurona de la siguiente capa, como se muestra en la figura 24.

Como se vio anteriormente, se definió un valor z proveniente de todos los nodos anteriores. La normalización por lotes considera todos los ejemplos z_i del lote. Por ejemplo, dado un lote de tamaño 32 se tendrán 32 z's. Nuevamente, como recordatorio, i representa el i-ésimo nodo y ℓ representa la ℓ-ésima capa. La normalización por lotes toma el tamaño del lote y procede a normalizar el mismo de forma tal de tener una media de cero y una desviación estándar de uno.

En primer lugar se obtiene la media μ y la varianza σ^2 del lote. Acto seguido se procede a resta la media y divide por la desviación estándar a cada z.

Por un lado, se tiene que:

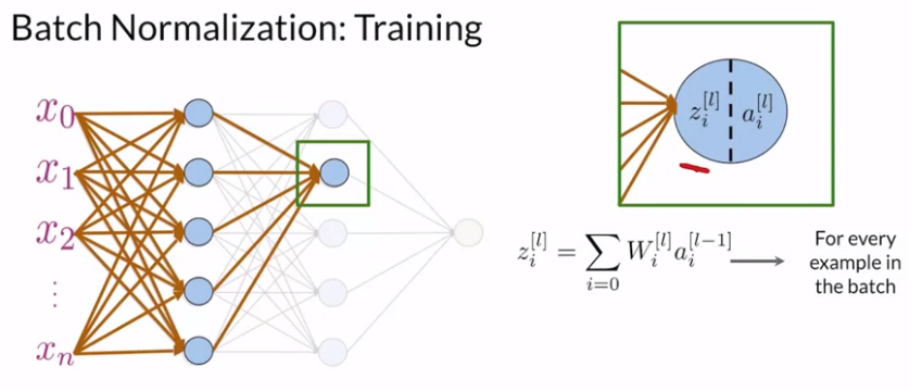


Figura 24: Normalización por lotes respecto de un nodo de la l-esima capa [4]

$$z_i^{[l]} = \sum_{j=0}^{n} W_j^{[l]} a_j^{[l-1]} + b^{[l]}$$

A partir de los valores z_i del lote, se calculan sus valores normalizados como sigue:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} z_i \ , \ \sigma = \frac{1}{n}\sum_{i=1}^{n}(z_i - \mu)^2 \ , \ \hat{z}_i^{[l]} = \frac{z_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

Observar que se agrega un ϵ dentro de la raíz cuadrada de la varianza para asegurarse que el denominador sea distinto de cero. Se denota $\hat{z}$ a los valores de z normalizados.

Como se indica en [14], vale señalar que esta transformación se hace para cada lote. Esto fue probado empíricamente por primera vez en 2015, donde se observó una mejora significativa en la performance del entrenamiento de las redes neuronales. Sin embargo, en [5] se destaca que esta medida de centrar y normalizar los datos puede ser drástica y restrictiva en cuanto al poder de expresividad de la red.

Por este motivo, para manejar esta situación se agregan dos parámetros $\gamma, \beta \ \in R$. Los cuales luego de centrar y normalizar los datos, se emplean para transformar los datos normalizados mediante la siguiente expresión:

$$\hat{y}_i^{[l]} = \gamma \hat{z}_i^{[l]} + \beta$$

Estos parámetros se ajustan durante el entrenamiento, para cada "mini-batch" mediante "Backpropagation".

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Figura 25: Batch Normalization - Procedimiento sobre un mini-batch [14]

Agregar estos parámetros y la función BN le brinda mayor variabilidad al modelo y mayor capacidad de ajuste, pero además cambia la dinámica del entrenamiento de la red haciéndola más fácil de entrenar[14].

En resumen, con la normalización por lotes se mejora la robustez frente a la inicialización: con este método, la inicialización no será tan importante dado que en distintos momentos de la red se fijan la media y la varianza. Por otro lado, se puede mostrar que esto se comporta como un regularizador, pues a cada dato se le resta la media del lote. Si a ese dato se lo pone en otro lote, se le restará otra media (que podrá ser más o menos cercana pero es otro valor). Eso es como agregar un poco de ruido a la red. Los parámetros se calculan con el "mini-batch".

¿Cómo se aplica la función BN en test?[5] Algo que nos podríamos preguntar es cómo es posible aplicar la función BN en la muestra de prueba dado que aquí contamos con un dato solo por vez. No tenemos un "mini-batch". Se utilizan los parámetros de media y varianza calculados durante el entrenamiento para centrar y normalizar el nuevo dato de entrada. Sin embargo, tenemos una media y una varianza para cada "mini-batch". Se utilizará el promedio de los parámetros calculados (μ, σ, γ y β) de todos los "mini-batches". Estos promedios se calculan como medias móviles, pues durante el entrenamiento resultan con mayor peso los valores más recientes.

Esta transformación de los datos es la entrada de la función de activación [4]. La normalización por lotes difiere de la normalización estándar dado que durante el entrenamiento se utilizan estas estadísticas a partir de cada lote y no de todo el conjunto de datos. Esto reduce el tiempo de cálculo y mejora el tiempo de entrenamiento.

2.5. Método de propagación hacia atrás

El algoritmo de propagación hacia atrás o en inglés "Backpropagation", nos permite calcular el gradiente de una red neuronal. Este será utilizado para la optimización de la función de discrepancia mediante el descenso por gradiente. Si se considera a la red como una función $f(x)$, se denota su gradiente como $\nabla f(x)$.El algoritmo utiliza derivadas parciales, la regla de la cadena y grafos computacionales.

Para explicar el concepto de grafo computacional se considera la función: $f(x, y, z) = max((x + y)z, 0)$. Esta función es una composición de la función $ReLU(g(x, y, z))$, siendo $g(x, y, z) = (x + y)z$. El grafo computacional correspondiente a esta expresión se muestra en la figura 26.

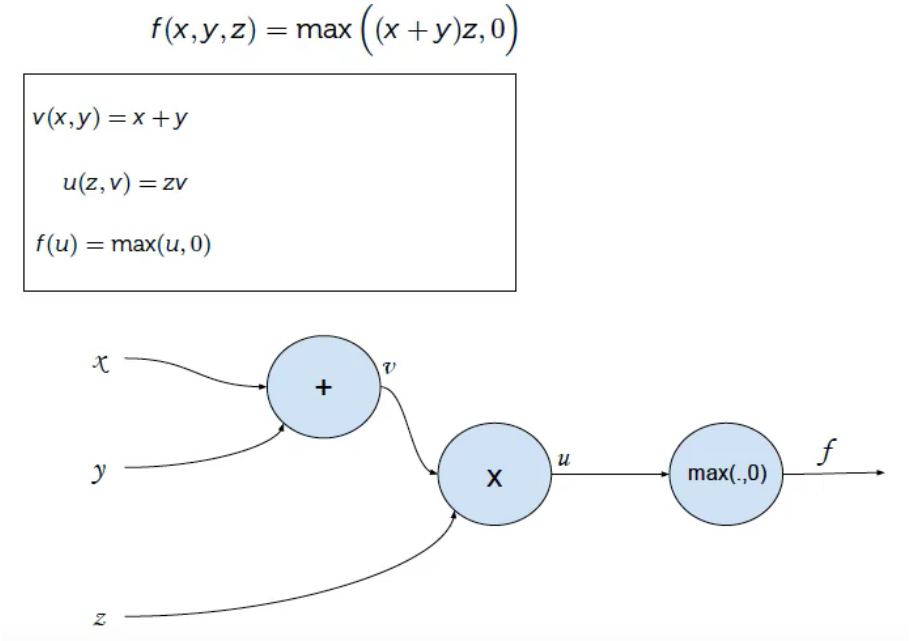


Figura 26: Grafo computacional de la función $f(x) = max((x + y)z, 0)$ [5]

En la figura 26 se observa que las entradas x e y se conectan con un nodo que ejecuta la operación Suma. La salida de esta suma se conecta con el siguiente nodo que ejecuta la multiplicación de $x + y$ con la entrada z. Finalmente esta salida se conecta con otro nodo que ejecuta la operación Máximo.

A partir de este ejemplo ($f(x) = max((x + y)z, 0)$ se explica el algoritmo de propagación hacia atrás.

Se calcula el gradiente de esta función a través de las derivadas parciales respecto a las entradas, a partir del gradiente de cada uno de los nodos. Primero se definen variables auxiliares asociadas a cada uno de los nodos como se muestra en la figura 26. Se asignan las variables auxiliares u, v y f a los nodos de multiplicación, suma y máximo respectivamente. El gradiente siempre se calcula respecto a un punto determinado de la red. En el ejemplo se elige el punto $x = -2$, $y = 3$ y $z = 2$ como se muestra en la figura 27.

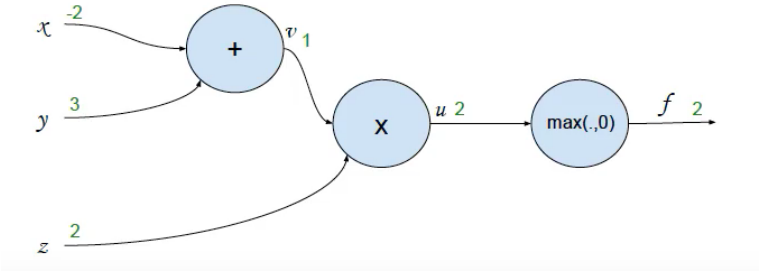


Figura 27: Grafo computacional luego de la etapa de Forward Pass[5]

El primer paso del algoritmo se denomina "forward pass" en inglés, o pasada hacia adelante. En esta etapa se hacen los cálculos a partir de la entrada, que es el punto sobre el cual deseamos calcular el gradiente. Es decir, se evalúa la red en el punto $x = -2$, $y = 3$ y $z = 2$. Luego se ejecuta un segundo paso denominado "Backward pass" en inglés, o pasada hacia atrás. En esta etapa se calculan las derivadas parciales de la función f respecto a cada una de las entradas x, y y z evaluadas en el punto elegido ($x = -2$, $y = 3$ y $z = 2$).

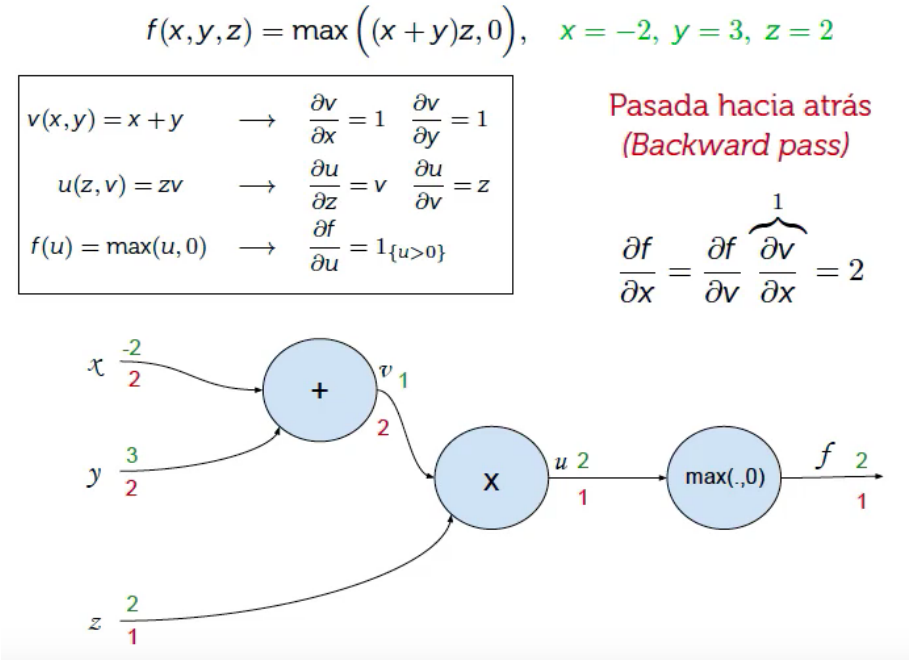


Figura 28: Grafo computacional luego de la etapa de Backward Pass. En verde se expresa el resultado parcial de la operación y en rojo la derivada parcial respecto de la función (pasada hacia atrás) [5]

El algoritmo comienza por el nodo "final" calculando $\frac{\delta f}{\delta f} = 1$. Luego continúa con el nodo anterior u y calcula $\frac{\delta f}{\delta u} = 1_{\{u>0\}}$. Esta derivada evaluada en $u = 2$ es igual a 1. Aplicando la regla de la cadena se calcula la $\frac{\delta f}{\delta z} = \frac{\delta f}{\delta u}\frac{\delta u}{\delta z} = 1v = 1 \cdot 1 = 1$. Recordar que en la etapa de propagación hacia adelante, como se muestra en la figura 27 $v = 1$.

Ahora aplicando nuevamente la regla de la cadena ,se calcula $\frac{\delta f}{\delta v} = \frac{\delta f}{\delta u}\frac{\delta u}{\delta v} = 1z = 1 \cdot 2 = 2$.

De un modo análogo se obtiene $\frac{\delta f}{\delta y} = \frac{\delta f}{\delta v}\frac{\delta v}{\delta y} = 2 \cdot 1 = 2$. Finalmente, $\frac{\delta f}{\delta x} = \frac{\delta f}{\delta v}\frac{\delta v}{\delta x} = 2 \cdot 1 = 2$.

La figura 28 muestra el resumen de los cálculos realizados. Donde se puede observar que el cálculo de esta segunda etapa comienza desde el "final" del grafo considerando a las entradas como el "principio" del grafo para calcular las derivadas parciales. En forma previa, es preciso evaluar la red en todos los nodos dado que estos resultados también son utilizados al calcular las derivadas parciales.

2.6. Redes neuronales de convolución

Las redes de convolución, son en la actualidad el tipo de redes más utilizado en el área de visión artificial [5]. A diferencia de las redes neuronales prealimentadas, las redes de convolución conectan cada nodo con un subconjunto de elementos de la capa anterior.

La visión humana es invariante, en el sentido que puede, por ejemplo, reconocer la presencia de un gato en distintas imágenes como se muestra en la figura 29. Esta capacidad es una característica muy deseable en las redes neuronales utilizadas en visión artificial. Para lograr esto, se podría utilizar un gran conjunto de datos de entrenamiento que presenten una gran variedad de escenas que contengan un gato. De este modo la red neuronal podrá "aprender" a ser invariante a partir de los datos. Otro camino posible es que la propia estructura de la red otorgue la capacidad al modelo de ser invariante frente a nuevas imágenes.

Figura 29: Invarianzas frente a distintas imágenes que contienen un gato [4]

Algunos de los problemas más comunes en el campo de visión artificial son: segmentación semántica, descripción de imágenes, detección de acciones o transferencia de estilos [5]. Estos son considerados de alta dificultad debido esencialmente a que se requieren invarianzas frente a distintas transformaciones de una imagen. En estos tipos de escenarios las redes neuronales de convolución tienen un buen desempeño.

La información visual se procesa por el cerebro humano de manera jerárquica. Por ejemplo, en una imagen con árboles primero se reconocen bordes, luego la presencia de hojas, ramas, etc. Para finalmente reconocer la presencia de un conjunto de árboles en la imagen. En un sentido más general, se detectan elementos muy simples construyendo conceptos cada vez más abstractos para finalmente construir el concepto general de la imagen. Esta capacidad se logra mediante profundidad en las redes de convolución. Las capas de bajo nivel procesarán los conceptos simples, mientras que las capas de más alto nivel crearán los conceptos más abstractos [5].

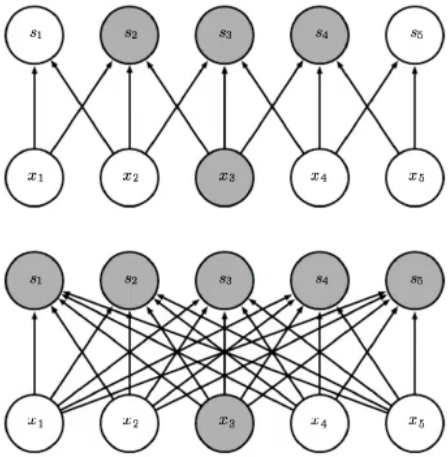


Figura 30: Ejemplo de red de convolución contra su correspondiente red totalmente conectada [5].

En las redes de convolución, cada nodo se conecta con un subconjunto de elementos de la capa anterior. Esto permite disminuir significativamente la cantidad de parámetros y la profundidad de la red manteniendo la capacidad de ser invariante. Para dar una idea de la diferencia de parámetros entre una red totalmente conectada y una red de convolución se muestra un ejemplo en la figura 30 donde se observan dos redes neuronales: arriba una red de convolución y debajo una red totalmente conectada. Al comparar las redes se puede observar que mientras la red totalmente conectada tiene 25 parámetros (sin contar los parámetros de sesgo), la red de convolución tiene solamente 13. En el procesamiento de una imagen, con esta estructura de red, las capas más cercanas a la entrada (que es un vector de datos que representa la imagen), procesarán los elementos más sencillos y las capas de más alto nivel trabajarán con información ya procesada de la capa anterior como se muestra en la figura 31.

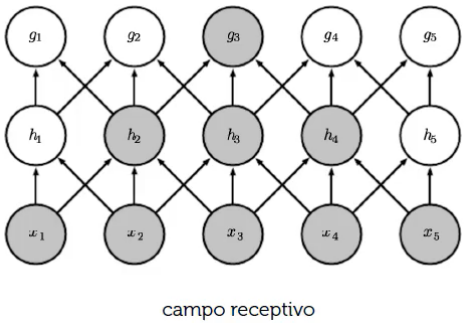


Figura 31: Campo receptivo en una red de convolución [5]

La convolución permite detectar características claves en diferentes áreas de una imagen usando lo que se denominan filtros [4]. Un filtro es una matriz de pesos o parámetros, que recorrerá la imagen en busca de una característica específica. Durante el entrenamiento, el algoritmo de optimización ajusta los parámetros iterativamente para minimizar la función de pérdida.

Los valores de los pesos se inicializan al principio del entrenamiento de la red, aunque existen distintas técnicas de inicialización, la estrategia más común consiste en muestrear los valores a partir de una distribución normal o uniforme. Los valores iniciales juegan un rol relevante en la determinación del tipo de características que aprenderán. Por ejemplo, si los valores iniciales de los filtros están configurados de manera que favorezcan la detección de líneas verticales, entonces esos filtros tenderán a especializarse en la detección de bordes verticales en los datos de entrada. Basados en esta idea, se pueden obtener filtros de mayor complejidad, ya sea un filtro para detectar ojos, otro detectará la presencia de una nariz y otro podría detectar orejas, tal como se muestra en la figura 32.

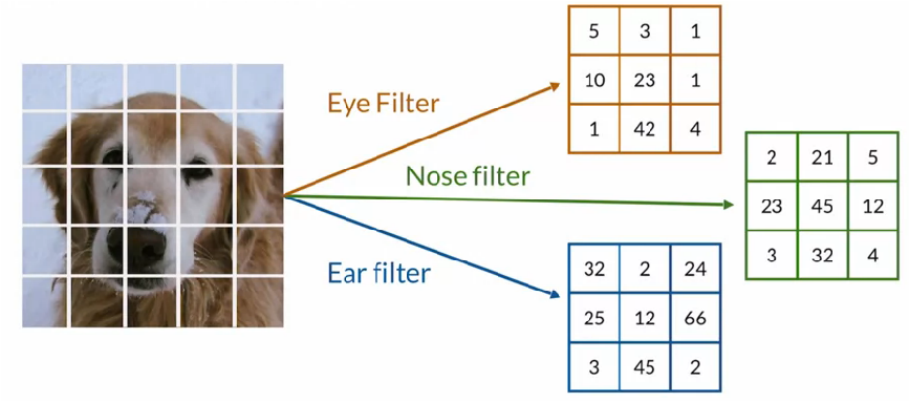


Figura 32: Detección de características en una imagen a partir de la aplicación de filtros en una red de convolución [4].

La convolución es una operación que se aplica a través de un proceso iterativo entre un filtro y una imagen, ambas representadas como matrices numéricas y dicha operación se representará con un asterisco. A modo de ejemplo, aplicaremos una convolución de un filtro de 3x3 sobre la figura 33 que es una imagen representada por una matriz de 5x5 en escala de grises, donde cada píxel tiene un valor entre 0 y 255. El valor 0 representa píxeles negros y 255 representa píxeles blancos, siendo los valores del medio matices de gris.

50	50	0	0	0
50	50	0	0	0
50	50	0	0	0
50	50	0	0	0
50	50	0	0	0

Figura 33: Ejemplo de una imagen en escala de grises [4]

El filtro tiene como objetivo la detección de líneas verticales y estará formado por: unos en la primera columna, ceros en la segunda y menos unos en la tercera, tal como se muestra en la figura 34. El proceso iterativo comienza con el primer paso colocando el filtro en la esquina superior izquierda de la imagen y se multiplican todos los elementos del filtro uno

a uno por la matriz correspondiente; luego se suman los productos y su resultado se guarda en el elemento superior izquierdo de la matriz resultante tal como se muestra en la figura 34.

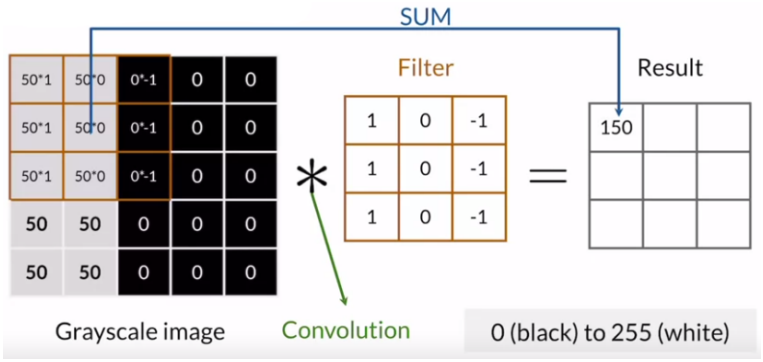


Figura 34: Ejemplo de aplicación de filtro 3x3 a una imagen en escala de grises [4]

En la segunda iteración, se desplaza el filtro una posición hacia la derecha; se obtiene nuevamente el producto elemento a elemento entre el filtro y la nueva región de la imagen, se suman los resultados y se almacena el valor en la fila 1 y columna 2 de la matriz resultante. El procedimiento anterior se repite hasta que el filtro llegue al extremo derecho de la imagen. Una vez se cumple con esta condición, el filtro se desplaza nuevamente al extremo izquierdo de la imagen y una posición hacia abajo. El procedimiento se repite hasta que el filtro se haya desplazado por la totalidad de la imagen tal como se muestra en la figura 35.

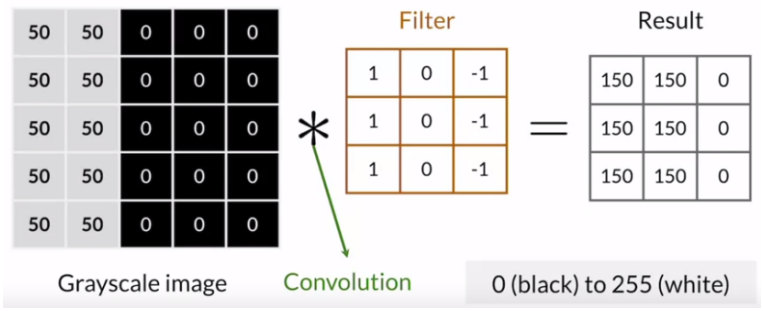


Figura 35: Convolución - Filtro aplicado a imagen en escala de grises [4]

Desplazamientos de convolución

Al aplicar un filtro convolucional de tamaño 3x3 sobre una imagen en donde este se mueve un paso hacia la derecha y luego un paso hacia abajo como se vio anteriormente, diremos que el filtro utiliza un desplazamiento de 1 paso (o "stride" en inglés). De esta forma, se puede definir el desplazamiento como un parámetro en las redes neuronales convolucionales que determina la cantidad de pasos con el que se aplica un filtro a la imagen de entrada [4].

Otra forma de aplicar el filtro podría ser con un desplazamiento = 2, el cual implica hacer un salto de a dos píxeles a la derecha y dos píxeles hacia abajo. Continuando con el ejemplo anterior, esto generaría un total de cuatro cálculos únicamente, por lo que el proceso de aprendizaje requiere menos tiempo.

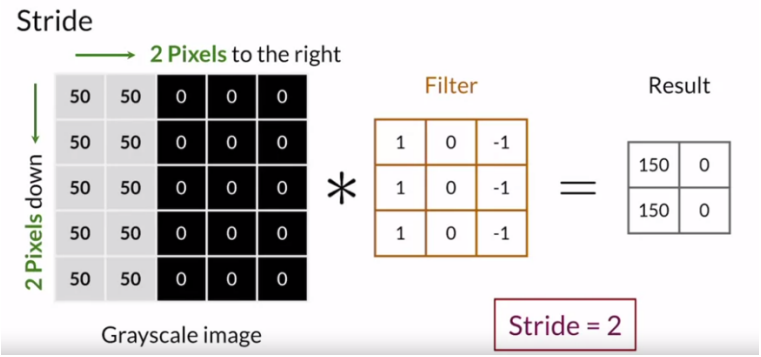


Figura 36: Filtro aplicado con un desplazamiento de dos pasos[4]

También se podría optar por elegir distintos desplazamientos de forma vertical y horizontal, esto se lo conoce como desplazamiento mixto. Un ejemplo podría ser dos hacia la derecha y cuatro hacia abajo. Cuanto mayores sean sus pasos, menor será la cobertura de su imagen para su filtro, pero al mismo tiempo hará que el cálculo sea más rápido.

Enmarcado de imágenes en redes de convolución

Continuando con el ejemplo anterior, al calcular una convolución con un desplazamiento de uno, se visitan cuatro secciones de la imagen: una en la esquina superior izquierda, una en la esquina superior derecha, una en la parte inferior izquierda y la última en la parte inferior derecha, como se muestra en la figura 37. El píxel en el centro se visita cuatro veces; los píxeles en las esquinas una vez y los restantes dos veces. Esto significa que la información en el centro recibe más atención que la información que se encuentra en los bordes; lo que puede representar un problema si las características en los bordes fueran de igual interés que el resto de la imagen. Para resolverlo se aplica el concepto de enmarcado o "padding" en inglés.

En el enmarcado se agrega un marco alrededor de la imagen para que toda la información aparezca en el centro de esta nueva imagen. El tamaño del marco puede variar según el tamaño de su filtro y el tamaño de la imagen. El valor usado dentro de los píxeles del marco a menudo se establece en cero, lo que se denomina "zero-padding" en inglés.

Cuando se calcula la convolución, el filtro escanea la imagen junto con su marco, pero esta vez el resultado es que cada píxel de la imagen se visita la misma cantidad de veces. El enmarcado se coloca alrededor de la imágen, con el fin de dar la misma importancia a los píxeles de los bordes y del centro.

Método de "Pooling" o agrupación

El método de "Pooling" (o agrupación) en una red neuronal convolucional, es una operación utilizada para reducir la dimensión de entrada y al mismo tiempo conservar la información más relevante. Esta técnica se aplica típicamente después de las capas de convolución para reducir el tamaño de la imagen y así el número de parámetros en la red, lo que ayuda a controlar el sobreajuste y a mejorar la eficiencia computacional.

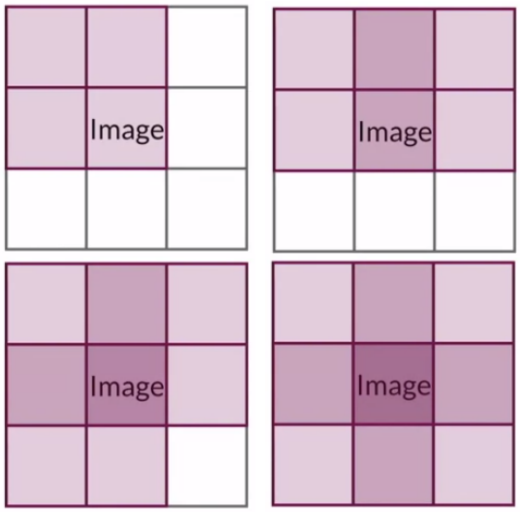


Figura 37: Ejemplo de aplicación de filtro donde el píxel en el centro se visita cuatro veces; los píxeles en las esquinas una vez y los restantes dos veces [4].

Como se muestra en la figura 38 se presenta el resultado de la aplicación de una capa de "Pooling", que resulta en otra imagen borrosa con menor tamaño y resolución. En este ejemplo, se observa que la paleta de colores y la distribución de colores en ambas imágenes sigue siendo similar. Si bien la imagen se ve borrosa, mantiene los bordes y formas originales. Será mucho menos costoso hacer cálculos en esta imagen que en la imagen original. De esta forma se puede ver que el método de "Pooling" reduce el tamaño de la imagen manteniendo las características más relevantes de la misma.

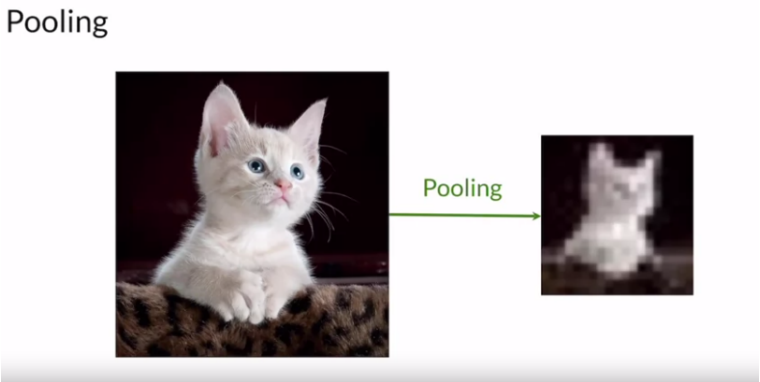


Figura 38: Convolución - Aplicación de una capa de agrupación sobre una imagen [4]. Aquí se observa que en ambas imágenes se mantienen la distribución de colores, los bordes y sus formas.

Uno de los tipos más populares de "Pooling" se denomina "Max-Pooling". Dada una imagen de tamaño 4x4, como se muestra en la figura 39, se busca obtener otra de tamaño 2x2. Para ello, se recorre la imagen tomando regiones de dicho tamaño de forma similar a la operación

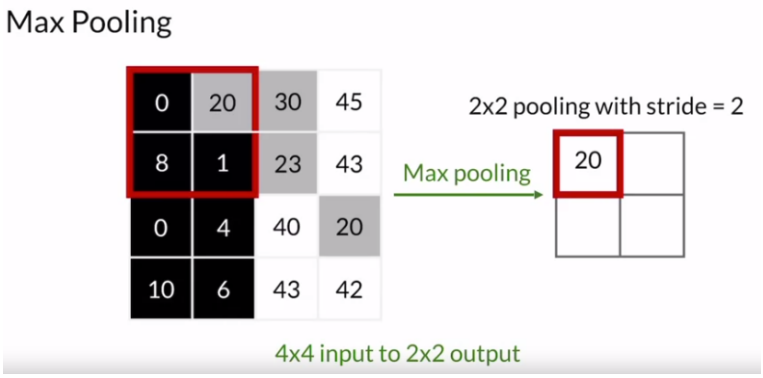


Figura 39: Ejemplo de aplicación de Max Pooling con un filtro de tamaño 2x2 con 2 pasos de desplazamiento [4].

de convolución. Se define un filtro de "Pooling" de tamaño 2x2 y en este ejemplo, se aplicará un desplazamiento de dos pasos para de este modo visitar cada píxel solamente una vez. Este proceso iterativo comienza ubicando el filtro en la esquina superior izquierda de la imagen y se encuentra el valor máximo de la ventana, que corresponde al valor 20 finalizando así la primera iteración. Luego, se desplaza el filtro dos pasos hacia la derecha, ubicando el mismo en la esquina superior derecha. Observar que no hay superposición de píxeles. En este caso, el valor máximo encontrado es 45. En la tercera iteración se vuelve al extremo izquierdo pero dos filas hacia abajo y se continúa con el mismo procedimiento hasta cubrir toda la imagen. El resultado final será una matriz de tamaño 2x2, que contiene el máximo de cada una de estas regiones procesadas de la imagen.

Existen otras operaciones de "pooling" como ser:

- Average Pooling: se calcula la media de cada sección. Es una operación utilizada con frecuencia.
- Minimum Pooling: se calcula el mínimo de cada sección.

Es importante destacar que durante la aplicación de una capa de "Pooling" no hay parámetros para ajustar; solo se aplica una regla simple en la imagen (o capa intermedia). Este aspecto es donde se diferencia respecto a la operación de convolución.

2.6.1. Método de Upsampling

La operación de "Upsampling" en inglés, aumenta la resolución espacial de una imagen o característica al expandirla en dimensiones más grandes, tal como se aprecia en la figura 40. Este método tiene el efecto opuesto al método de "Pooling". Para realizar esta tarea, se requiere inferir valores para los píxeles adicionales mediante la operación de "Upsampling" o de convolución traspuesta (también conocida como deconvolución) [4].

Upsumpling de vecinos más cercanos

Una forma de aumentar la muestra se conoce como "Upsumpling" de los vecinos más cercanos. Con este método se copian los valores de los píxeles entrada varias veces para completar

Figura 40: Ejemplo efecto de aplicar Upsampling a una imagen [4]

su salida [4]. Para ilustrar, se presenta una imagen en escala de grises de tamaño 2x2 que se desea aumentar a una salida de 4x4 como se muestra en la figura 41.

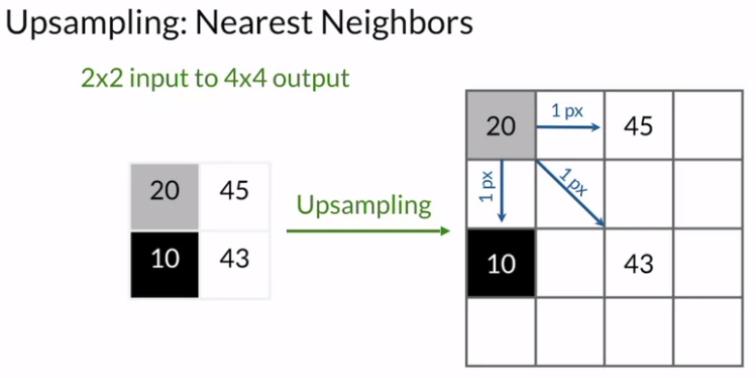


Figura 41: Ejemplo de Upsampling vecinos más cercanos de una matriz 2x2 a una 4x4 [4]

Primero, se asigna el valor de la esquina de la imagen 2x2, en el píxel superior izquierdo de la imagen 4x4. Los otros píxeles "vecinos" a este primer vértice superior izquierdo se añaden a una distancia determinada (en este caso dos píxeles de distancia). Habrá exactamente un píxel entre cada uno de ellos. Luego, se asigna el mismo valor a todos los demás píxeles que a su vecino más cercano y se llega al resultado que se observa en la figura 42. Al igual que en el método de "pooling", las capas de "upsampling" no tienen ningún parámetro. Es solo una regla fija.

Convolución traspuesta

La convolución traspuesta es otra forma de aplicar "upsampling". La principal diferencia radica en la utilización de un filtro con parámetros que se entrenan a efectos de ampliar el tamaño de su entrada. Para explicar el concepto se utilizará una matriz de tamaño 2x2 a la que se le aplica una operación de convolución traspuesta para obtener una matriz de 3x3. Se comienza tomando el valor superior izquierdo de su entrada y obteniendo su producto con cada valor en el filtro 2x2. Luego, se guarda este valor en la región superior izquierda de tamaño

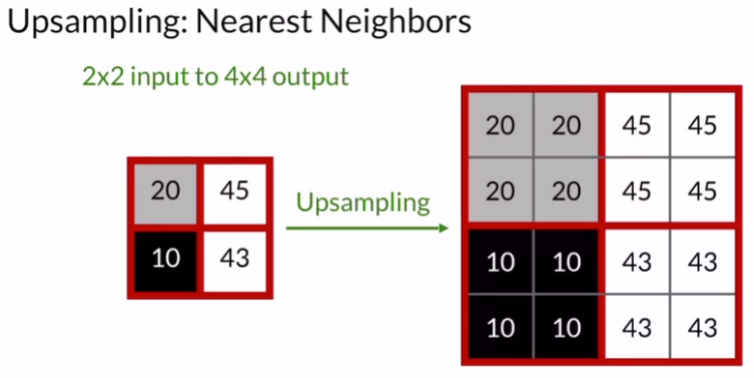


Figura 42: Ejemplo de Upsampling vecinos más cercanos de una matriz 2x2 a una 4x4 parte 2 [4]

2x2 en la matriz resultado como se muestra en la figura 43.

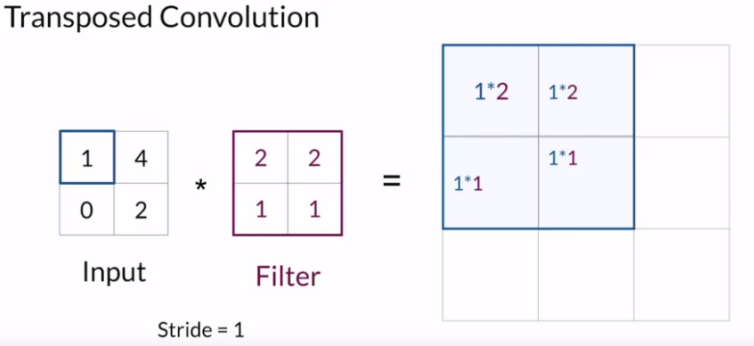


Figura 43: Convolución - Ejemplo de convolución traspuesta en una matriz 2x2 con un filtro 2x2 [4]

Se mueve el filtro un paso hacia la derecha (recordar que se está aplicando un desplazamiento de un paso) y se aplica el filtro con el siguiente valor a la derecha (en este caso corresponde al 4).Existen píxeles que se solapan con las operaciones anteriores; en ese caso se suman los productos como se muestra en la figura 44.

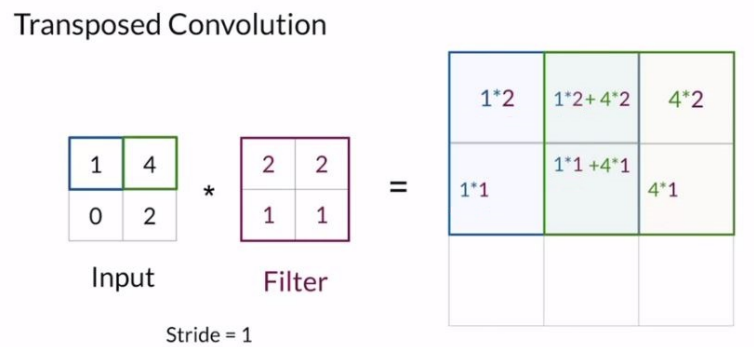


Figura 44: Convolución - Ejemplo de Transposed Convolutions Parte 3 [4]

Por último, se mueve el filtro a la esquina inferior izquierda de su entrada y toma el producto nuevamente con el filtro. Este proceso iterativo continúa hasta que se haya cubierto toda su entrada obteniendo el resultado de la figura 45:

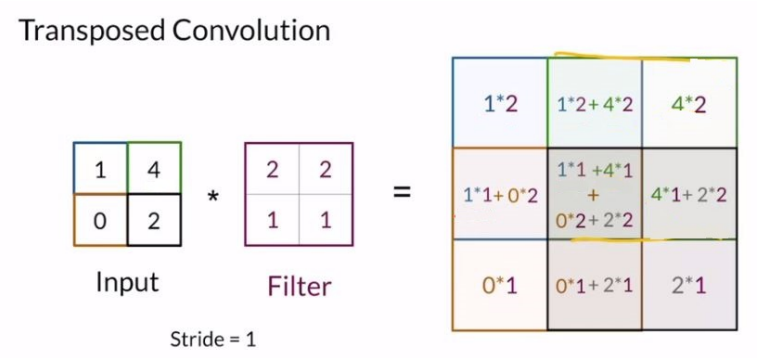


Figura 45: Convolución - Ejemplo de Transposed Convolutions Parte 4 [4]

Con este procedimiento, se puede ver que algunos de los valores de la salida están más influenciados por los valores de la entrada. Por ejemplo, el píxel central en la salida está influenciado por todos los valores en la entrada, mientras que las esquinas están influenciadas por un solo valor. El hecho de que existan píxeles mucho más influenciados que otros genera un inconveniente conocido como el "checker board problem" en inglés [4].

3. Redes generativas adversarias (GAN)

En el marco del aprendizaje profundo, uno de los avances más importantes de los últimos años ha sido el desarrollo de las redes generativas adversarias (GAN, por sus siglas en inglés) [15]. Esta técnica ha permitido a las computadoras generar contenido artificial con un nivel de precisión tal que pueda engañar al ojo humano [1]. Las GAN están compuestas por dos redes neuronales: un Generador y un Discriminador, que se entrenan enfrentándose entre sí. Este proceso se lleva a cabo en múltiples pasos, donde en cada uno de ellos se ajustan los parámetros de las redes para que el Discriminador pueda clasificar correctamente los datos reales y el Generador pueda "engañar" al Discriminador [3].

En este capítulo, se presentará de manera intuitiva el funcionamiento de las GAN, describiendo la tarea de cada componente y algunas de sus aplicaciones en el mundo real. Posteriormente, se definirá la arquitectura de los modelos que componen las GAN y se describe el proceso de entrenamiento. Por último, se discutirán algunos de los problemas típicos asociados con el entrenamiento de las GAN.

Aunque sólo existen desde 2014 [2], las GAN han logrado un destacado rendimiento en múltiples tareas, entre las que se incluyen la generación de rostros humanos y la animación de obras de arte [4]. Ian Goodfellow, ampliamente considerado como su creador [2], destaca la velocidad con la que han mejorado a lo largo de los años. Por ejemplo, en sus inicios eran capaces de generar solamente rostros en blanco y negro de aspecto bastante inhumano, pero hacia el año 2018 ya eran capaces de reproducir rostros realistas en colores de muy buena calidad [1]. En la actualidad, estas imágenes tienen alta resolución a nivel de fotografías profesionales. Como se puede ver en "This Person Does Not Exist", es fácil pensar que estas fotografías pertenecen a personas reales, pero en realidad no existen. También pueden ser gatos, como se muestra en "These Cats Do Not Exist". Las GAN no solamente pueden reproducir rostros humanos o animales, sino que también tienen la habilidad de aprender de cualquier juego de datos que se les proporcione.

Las GAN también son capaces de realizar lo que se conoce como transferencia de estilos. Por ejemplo, pueden transformar la imagen de un caballo en una cebra y viceversa, sin necesidad de contar con la imagen de una cebra y de un caballo haciendo las mismas acciones, sino simplemente transfiriendo el estilo [1]. Además, pueden tomar un retrato como la Mona Lisa y animarlo usando el movimiento de la cara de cualquier persona [16]. Estas técnicas no se limitan a imágenes 2D, sino que también pueden generar objetos 3D, como se puede ver en [17], al pasar de una imagen 2D a una interpretación tridimensional de la misma.

Varias empresas destacadas han comenzado a utilizar las GAN para una variedad de aplicaciones [4]: Por ejemplo, Adobe plantea una próxima generación de Photoshop, donde los artistas novatos puedan actuar a un nivel experto sin tener tales conocimientos. Google utiliza GAN en gran medida para la generación de texto, así como para generar imágenes. IBM las utiliza para el aumento de datos, generando ejemplos sintéticos y, de esta manera, aumentar el conjunto de datos de entrenamiento para un clasificador. Esto es muy útil si no se cuenta con suficiente cantidad de datos de una determinada clase o de un determinado tipo de imagen. En el ámbito de la medicina, también se utilizan para la generación de datos médicos artificiales o incluso para detectar anomalías en radiografías [18]. Snapchat y TikTok trabajan en nuevos

filtros creativos que son muy utilizados en la actualidad. Incluso Disney las utiliza para crear imágenes con súper resolución. Se presentan más ejemplos en el siguiente enlace: "Impressive Applications of Generative Adversarial Networks".

Imagínese una situación en una galería de arte en la que se exhiben las mejores obras de famosos artistas. La galería cuenta con un inspector encargado de revisar cada obra antes de ser expuesta, con el fin de asegurarse de que todas las obras de arte sean auténticas. Sin embargo, existe un falsificador que se presenta regularmente para vender obras nunca antes vistas de artistas reales. Aunque los estilos y técnicas son muy similares a las obras originales, en realidad son falsificaciones. Cuando el falsificador crea su primera obra, no tiene idea de cómo imitar una obra real, del mismo modo que el inspector no es bueno diferenciando arte debido a su falta de experiencia. Después de cada exposición, el inspector recibe retroalimentación sobre si sus decisiones fueron correctas para aprender a distinguir técnicas o patrones utilizados por el falsificador. A su vez, el falsificador obtiene información valiosa sobre su trabajo a partir de las decisiones del inspector. ¿El falsificador está generando obras que parecen reales? ¿Existen rasgos en las pinturas que hacen creer al inspector que son reales, o por el contrario hay patrones que delatan al falsificador?

Esta situación sirve como una buena analogía para entender el funcionamiento de las GAN, ya que el Generador se puede ver como el falsificador de arte y el Discriminador como el inspector de la galería. En cada nueva presentación, el inspector (Discriminador) obtiene más experiencia y mejora su precisión en distinguir las obras reales de las falsas, mientras que el falsificador (Generador) aprende los detalles que el inspector observa para replicarlos cada vez mejor [4].

3.1. Definición y arquitectura de los modelos

Las GAN son una de las arquitecturas más interesantes y exitosas en el campo del aprendizaje profundo y la generación de imágenes y videos. La arquitectura de una GAN consta de dos componentes fundamentales: el Generador y el Discriminador. Estos dos componentes trabajan en conjunto mediante un proceso de entrenamiento de adversarios que se verá en detalle a lo largo del trabajo. A continuación, se explorará en profundidad cada uno de estos componentes y su papel en la arquitectura de una GAN.

Discriminador

El Discriminador [15] es una de las dos redes neuronales que componen una GAN, y es el encargado de distinguir entre imágenes reales y falsas generadas por el Generador. Es un modelo de clasificación binaria, es decir, que clasifica las imágenes en dos categorías: real o falsa.

El objetivo del Discriminador es aprender a diferenciar entre imágenes generadas por el Generador y aquellas que provienen de un conjunto de datos real. Para ello, se le proporciona una imagen de entrada y debe producir una salida que indique si la imagen es real o falsa[19].

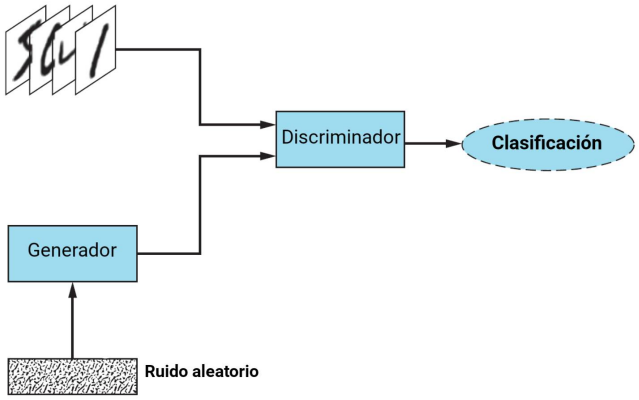


Figura 46: Arquitectura modelo Discriminador [1]

Los datos de entrada del Discriminador son un conjunto de imágenes, ya sean reales o las generadas por el Generador. La salida es un valor que indica la probabilidad de que la imagen de entrada sea real (o falsa). Los parámetros de la red se ajustan con el objetivo de minimizar la diferencia entre los valores reales Y y las predicciones $\hat{Y}$; se utiliza una función de costo, denominada $L(\theta)$, que mediante el método de descenso por gradiente modifica los parámetros θ. Durante el entrenamiento, existen varios hiperparámetros que se pueden ajustar para mejorar el desempeño del Discriminador, como la tasa de aprendizaje, el tamaño del lote, la función de activación, el número de capas, entre otros. Más adelante se profundizará cómo realizar cada uno de estos ajustes[1].

En resumen, el Discriminador es un modelo de clasificación binaria que se encarga de distinguir entre imágenes reales y falsas generadas por el Generador. Es esencial en la arquitectura de una GAN, ya que permite que el Generador aprenda a generar imágenes más realistas a medida que el Discriminador mejora su capacidad para distinguir entre imágenes reales y falsas.

Generador

El Generador es una red neuronal que tiene por objetivo aprender a generar imágenes lo suficientemente realistas para "engañar" al Discriminador. Como entrada toma un vector de dimensión fija, el cual se denomina ruido aleatorio (z) que muestrea aleatoriamente de una distribución de probabilidad normal estándar que se llamará espacio latente. De modo opcional, se puede utilizar como argumento de entrada la clase de objeto que se desea generar. Este modelo no utilizará datos reales en ninguna etapa de su entrenamiento [1]. Típicamente, el Generador se compone de una red neuronal de tipo convolucional. Las capas transpuestas convolucionales, son capaces de aumentar la resolución de una imagen permitiendo así la creación de imágenes de mayor calidad a medida que se profundiza en la red [3]. Su aprendizaje se basa en el resultado de la función de pérdida y al igual que el Discriminador, los parámetros θ se ajustan por medio del descenso por gradiente. A diferencia del Discriminador, que el modelo debe aprender a reconocer patrones en los datos, el Generador debe aprender a sintetizarlos [1].

Uno de los principales desafíos al entrenar un Generador es evitar que genere imágenes demasiado similares, es decir, que no tenga suficiente variabilidad en las imágenes generadas. Para abordar este problema, se puede utilizar una técnica llamada "normalización por lotes" [4] que normaliza los valores de entrada para cada capa en la red y ayuda a garantizar que la red aprenda características más diversas y significativas en los datos.

Función de costo

La función de costo, o función de pérdida, es una parte fundamental de las GAN, ya que permite al Generador y al Discriminador aprender y mejorar sus habilidades[1]. Su objetivo es medir la diferencia entre las distribuciones de probabilidad de las imágenes generadas por el Generador y las imágenes reales del conjunto de datos de entrenamiento. De una forma genérica se puede expresar la función de costo de la siguiente manera [5]:

$$Ł(G, \theta_G, z, D, \theta_D, x) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x, \theta_D)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z, \theta_G)))]$$

Donde G representa al Generador, D al Discriminador, x son los datos reales del conjunto de datos de entrenamiento, $p_{\text{data}}(x)$ su distribución de probabilidad; z vectores de ruido y $p_z(z)$ la distribución de probabilidad del espacio latente utilizado como entrada del Generador. La función de costo busca minimizar la diferencia entre la distribución de probabilidad real y la distribución de probabilidad generada por G. La primera parte de la fórmula mide la capacidad del Discriminador de distinguir correctamente entre imágenes reales. Mientras que la segunda parte mide la capacidad del Generador de "engañar" al Discriminador para que crea que las imágenes generadas son reales. En otras palabras, el Generador intenta maximizar la segunda parte de la fórmula, mientras que el Discriminador intenta maximizar la primera parte y minimizar la segunda.

Existen varios tipos de funciones de costo que se pueden utilizar en las GAN, la más común es la de entropía binaria cruzada o BCE por sus siglas en inglés. Esta se puede representar de la siguiente manera:

$$L_{\text{BCE}} = -\frac{1}{N} \sum_{i=1}^{N} [y_i \log(D(\phi_i)) + (1 - y_i) \log(1 - D(\phi_i))]$$

Donde:

- N es el número total de ejemplos en el conjunto de datos (obsérvese que se tiene la misma cantidad de ejemplos reales y falsos).
- ϕ_i es el i-ésimo ejemplo proveniente del conjunto de datos de entrenamiento (conjunto que tiene imágenes reales y falsas).
- y_i es la etiqueta real correspondiente al i-ésimo ejemplo. Si $y_i = 1$, indica que el ejemplo es real, mientras que si $y_i = 0$, indica que el ejemplo es generado.
- $D(\phi_i)$ es la salida del Discriminador para el i-ésimo ejemplo real. Representa la probabilidad estimada por el Discriminador de que el ejemplo sea real.

La función de costo BCE se compone de dos términos, uno para los ejemplos reales y otro para los ejemplos generados. Si $y_i = 1$, es decir, si el ejemplo es real, el término $y_i \log(D(\phi_i))$ mide la discrepancia entre la probabilidad estimada por el Discriminador para el ejemplo real y la etiqueta real (1). El objetivo es maximizar esta probabilidad para los ejemplos reales. Por otro lado, si $y_i = 0$, es decir, si el ejemplo es generado, el término $(1 - y_i) \log(1 - D(\phi_i))$ mide la discrepancia entre la probabilidad estimada por el Discriminador para el ejemplo generado y la etiqueta real (0). El objetivo es minimizar esta probabilidad para los ejemplos generados.

También se suele utilizar la función de error medio cuadrático (MSE Loss) o la función de pérdida de Wasserstein (la cual se verá posteriormente), entre otras. Cada una de ellas tiene sus propias ventajas y desventajas por lo que es importante destacar que la elección de la función de pérdida adecuada depende del tipo de problema que se esté abordando y de las características de los datos.

Entrenamiento del modelo

Como se mencionó previamente, el entrenamiento de una GAN implica entrenar tanto el Generador como el Discriminador en una configuración de competencia, donde el objetivo es que el Generador genere muestras lo más parecidas posible a las muestras reales, mientras que el Discriminador debe ser capaz de distinguir las muestras generadas del Generador de las muestras reales.

Para entrenar una GAN[1], se inicializan los pesos del Generador y el Discriminador con valores aleatorios y en cada iteración se actualizarán los parámetros de ambos modelos. Los parámetros de Discriminador se actualizan mediante el método de propagación hacia atrás y recibe como entrada datos reales y sintéticos creados por el Generador. Luego de entrenado el Discriminador, el Generador toma elementos del espacio latente para generar nuevas imágenes. Estás son tomadas por el Discriminador y evalúa si son reales o falsas. Este resultado es el utilizado por el Generador para actualizar sus parámetros. Es decir que el Generador utiliza únicamente la información brindada por el Discriminador para medir la calidad de la generación de datos.

En cada paso de entrenamiento, se actualizan los pesos del Discriminador para minimizar la función de pérdida del Discriminador y se actualizan los pesos del Generador para maximizar la función de pérdida del Discriminador. Este proceso se repite iterativamente hasta que se alcanza un cierto criterio de parada (convergencia a una solución óptima o número máximo de iteraciones). Además, se pueden ajustar otros hiperparámetros como la tasa de aprendizaje, el tamaño del lote y el número de épocas de entrenamiento para evitar problemas como el colapso del modo o sobreajuste y así obtener un mejor rendimiento de la red. Estos conceptos se estudiarán a lo largo de esta sección.

Solo se entrena un modelo a la vez, mientras que el otro se mantiene constante [1]. Es importante tener en cuenta que ambos modelos deben mejorar juntos y deben mantenerse en niveles de habilidad similares desde el comienzo del entrenamiento.

El razonamiento detrás de esto es que al tener un Discriminador muy superior al Genera-

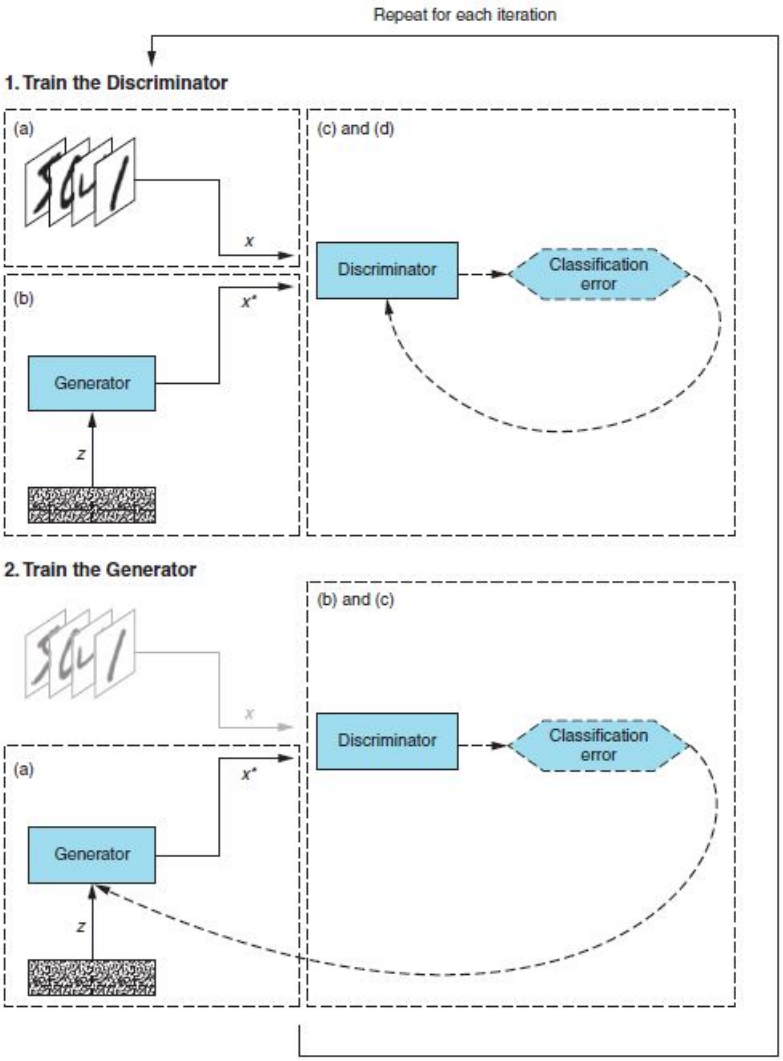


Figura 47: Pasos del entrenamiento de una GAN [1]

dor, se obtendrán predicciones donde todos los ejemplos falsos son 100 % falsos. Esto no será útil para el Generador ya que no tendrá indicaciones de como mejorar. Lo mismo sucede en el caso que el Generador que supere por completo al Discriminador, donde todas las imágenes serán 100 % reales.

A continuación se presenta el algoritmo 1 de entrenamiento de una GAN.

Algorithm 1 Entrenamiento de una GAN

Definir dimensión de espacio latente z.
Definir arquitectura de Discriminador (D).
Definir arquitectura de Generador (G).
Definir arquitectura GAN. Un modelo auxiliar que secuencia la arquitectura de G y D. Recibe como entradas muestras de z, genera imágenes con G y luego las clasifica con D (Obs: En esta arquitectura los parámetros de D no son entrenables).
Cargar juego de datos con observaciones reales.
Definir cantidad de épocas de entrenamiento, tasa de aprendizaje y tamaño de lote (n).
Calcular cantidad de lotes por época = 2 $\frac{Observaciones\ total\ de\ la\ muestra}{tamaño\ de\ lote}$.
for de i = 1 a total de épocas **do**
    for de j = 1 a cantidad de lotes por época **do**
        Muestrear minibatch de tamaño $\frac{n}{2}$ de muestras de imágenes reales.
        Muestrear minibatch de tamaño $\frac{n}{2}$ de muestras a partir del espacio latente z.
        Generar a partir de las muestras de z imágenes con el modelo G.
        Consolidar ambos minibatches de imágenes con su correspondiente etiqueta (real o falso).
        Entrenar D y actualizar sus parámetros.
        Muestrear minibatch de tamaño n de muestras a partir del espacio latente z.
        Usar arquitectura de modelo GAN para generar a partir de las muestras de z imágenes con el modelo G.
        Etiquetarlos como reales.
        Entrenar arquitectura GAN para actualizar únicamente los parámetros de G.
    end for
end for

¿Cuál es el valor óptimo para D?

Tal como se muestra en [2] y [20] en una arquitectura GAN dado un Generador fijo, el óptimo del Discriminador es $D^*(x) = \frac{p_{data}(x)}{p_{data}(x) + p_g(x)}$. El generador G define implícitamente una distribución de probabilidad p_g como la distribución de las muestras G(z) obtenidas cuando $z \sim p_z$. Una vez que este se encuentre bien entrenado podría considerarse como un buen estimador de p_{data}. En este caso el óptimo de G se alcanza cuando $p_g = p_{data}$ siendo $D^*(x) = \frac{1}{2}$.

3.2. Desafíos y soluciones al entrenar una GAN

En este capítulo se abordarán algunos de los desafíos en el campo de las redes generativas adversarias. Se explorarán los desafíos y soluciones principales a la hora de entrenar una GAN como es el problema del colapso de modo y el desvanecimiento del gradiente. Cada uno de estos temas representa una contribución importante al campo de las GAN y su comprensión es fundamental para el desarrollo de modelos más efectivos y sofisticados en la generación de imágenes.

Colapso de modo

El colapso de modo, es un problema frecuente que ocurre durante el entrenamiento de una GAN, cuando el generador produce una variedad limitada de salidas, ignorando la diversidad presente en los datos de entrenamiento, incluso cuando se le suministran diferentes valores aleatorios de entrada. El colapso de modo se considera uno de los principales desafíos en la construcción de GAN[15]. Esto ocurre porque el generador intenta engañar al discriminador maximizando su capacidad para clasificar las imágenes generadas como auténticas, y si las imágenes generadas son muy similares entre sí, es menos probable que el discriminador detecte la falta de diversidad en el conjunto de datos generado.

Para ilustrar esta situación, consideremos un conjunto de imágenes de dígitos escritos a mano representados en dos dimensiones por las variables x_1 y x_2. Como se muestra en la figura 48 existen 10 concentraciones distintas de datos donde cada una de estas representa a un dígito. Siendo los círculos más oscuros la representación de aquellas observaciones con un mayor grado de similitud.

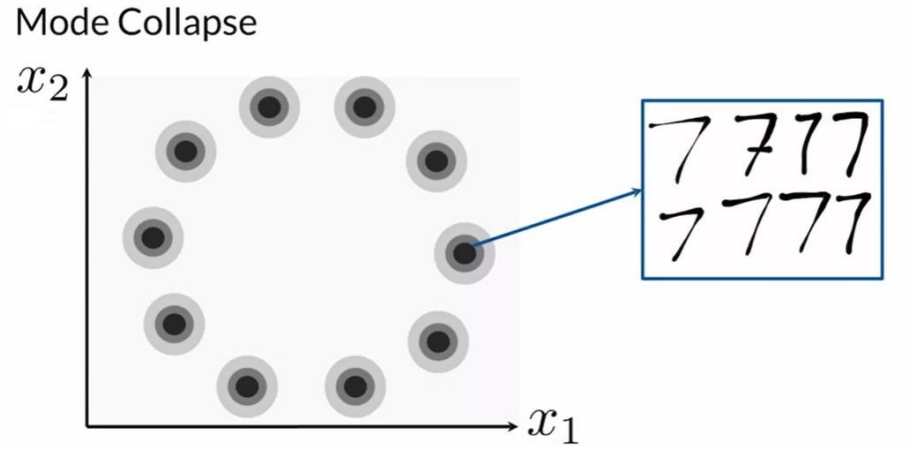


Figura 48: Ejemplo de representación de dígitos en dos dimensiones [4]

Supóngase que a partir de la etapa de entrenamiento k el Discriminador ha aprendido a identificar qué dígitos son escritos a mano y cuales son del Generador, excepto los que representan un uno o un siete. Desde un punto de vista técnico, esto podría indicar que nos encontramos en un mínimo local de la función de costo del Discriminador. Esta información se transmite al Generador, lo que le permite tener una buena idea de cómo engañar al Discriminador en el siguiente ciclo: generando únicamente imágenes que se asemejen a los dígitos uno o siete.

El colapso de modo en GAN es un desafío importante que requiere abordarse para mejorar la calidad y diversidad de las muestras generadas. A continuación, se presentan algunas posibles soluciones para mitigar este problema:

- Diseño de arquitectura del generador: se puede experimentar con arquitecturas más complejas y profundas para el generador, como el uso de capas adicionales. Esto permite al generador capturar de manera más efectiva la diversidad de características presentes en los datos reales.
- Regularización: la aplicación de técnicas de regularización, como la regularización L1 o L2, puede ayudar a evitar el colapso de modo al penalizar la generación de muestras demasiado similares. Estas técnicas promueven una mayor diversidad en las salidas del generador.
- Funciones de pérdida adicionales: se pueden agregar términos de pérdida adicionales que fomenten la diversidad en las muestras generadas. Por ejemplo, la información mutua máxima (Maximum Mutual Information, MMI) o la diversidad de Jensen-Shannon (Jensen-Shannon Diversity, JSD) se pueden utilizar como objetivos de optimización para evitar el colapso de modo.
- Aumento de datos: al aplicar técnicas de aumento de datos durante el entrenamiento, se introducen pequeñas variaciones en los datos de entrada, lo que puede ayudar a evitar que el generador se enfoque en generar solo un conjunto limitado de muestras.
- Actualización del discriminador: modificar la frecuencia con la que se actualiza el discriminador en relación con el generador puede tener un impacto en la superación del

colapso de modo. Ajustar la relación de actualización puede permitir que el generador aprenda de manera más efectiva y evite generar muestras repetitivas.

Es importante destacar que no existe una solución única para el colapso de modo, y las técnicas mencionadas pueden funcionar de manera diferente según el caso y el conjunto de datos. La combinación y experimentación de varias estrategias es fundamental para encontrar la mejor solución en cada escenario.

Desvanecimiento de gradiente

En redes neuronales el desvanecimiento de gradiente es un fenómeno que puede ocurrir durante el proceso de entrenamiento en aquellos algoritmos que usen métodos de aprendizaje basados en gradientes y retropropagación. En dichos métodos, durante cada iteración de entrenamiento, cada uno de los pesos de las redes neuronales recibe una actualización proporcional a la derivada parcial de la función de error con respecto al peso actual. El problema es que en algunos casos los gradientes se vuelven extremadamente pequeños generando que los pesos no cambien su valor o en los peores casos impedir por completo el entrenamiento de la red. Este problema es especialmente problemático en redes neuronales profundas, ya que los gradientes deben "atravesar" varias capas antes de llegar a las capas iniciales.

Según se expone en [21] el desvanecimiento de gradiente es un obstáculo común en el entrenamiento de una GAN. El generador puede tener dificultades para aprender de la señal de retroalimentación del Discriminador cuando este último se vuelve demasiado competente, y en particular al utilizar la función de pérdida de entropía binaria cruzada. La función sigmoide tiene gradientes cercanos a cero en las regiones de saturación, llevando a que durante la retropropagación del error, los gradientes se vuelven cada vez más pequeños a medida que se propagan hacia atrás a través de sus múltiples capas.

Para abordar este problema, se han propuesto diferentes enfoques, como el uso de otras funciones de pérdida, como la pérdida de Wasserstein o la divergencia de Jensen-Shannon. Estas funciones de pérdida tienen propiedades diferentes que pueden ayudar a evitar el desvanecimiento del gradiente y mejorar el rendimiento de las GAN. La pérdida de Wasserstein, en particular, se ha destacado por su capacidad para proporcionar gradientes más estables y evitar el problema del desvanecimiento del gradiente.

3.3. Variantes especializadas de GAN

En esta sección, se presenta un conjunto de variantes especializadas de las GAN que han surgido como respuestas innovadoras a desafíos específicos. Entre estas variantes se encuentran las WGAN, GAN condicional y controlada, cada una con su enfoque único y aplicaciones particulares. Se verán aspectos clave de estas variantes, analizando sus características y contribuciones distintivas. Desde la mejora de la estabilidad del entrenamiento con el modelo WGAN hasta la capacidad de generar imágenes condicionadas.

Wasserstein GAN (WGAN)

Wasserstein GAN o también denominadas WGAN fue presentado por Martin Arjovsky en 2017 en su artículo titulado Wasserstein GAN [22]. Se trata de una variante especializada de las GAN diseñada para abordar problemas de estabilidad tales como el desvanecimiento de gradiente a la hora de entrenar un modelo. A diferencia de las GAN tradicionales que utilizan la función de pérdida de entropía cruzada binaria para medir la probabilidad de que un dato sea real, WGAN emplea la distancia de Wasserstein (Earth Movers Distance - EMD) como medida de calidad de los datos generados [20]. Esta distancia cuantifica la cantidad de esfuerzo necesario para transformar una distribución en otra, teniendo en cuenta tanto la distancia como la cantidad.

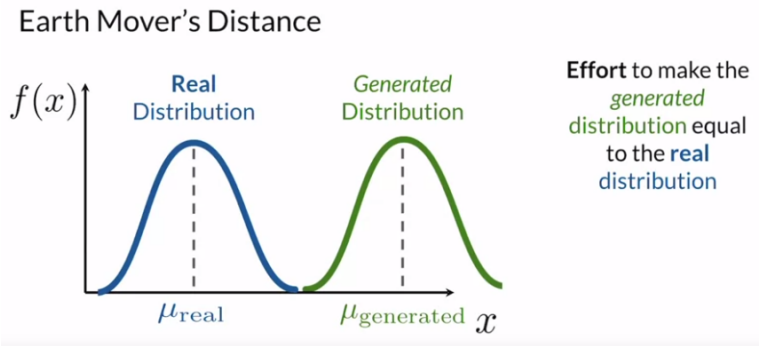


Figura 49: Earth Mover's Distance [4]

En términos de arquitectura, las WGAN introducen un componente llamado Crítico ($C(x)$) en lugar del Discriminador de las GAN tradicionales. Este componente puede ser entrenado para aproximarse a la distancia de Wasserstein y ser utilizado para el entrenamiento del Generador. El componente Crítico proporciona una puntuación numérica que representa la calidad de una muestra generada, en lugar de una probabilidad binaria; este resultado se puede interpretar como cuán real es considerada una imagen por el valor crítico. Este cambio se basa en el argumento matemático de que el generador debe buscar minimizar la distancia entre la distribución de los datos reales observada durante el entrenamiento y la distribución observada de los datos generados. La expresión de la función de costos es:

$$\min_G \max_c E(C(x)) \; - \; E(C(G(z)))$$

donde $C(x)$ son las evaluaciones del crítico sobre las muestras reales y $C(G(z))$ las evaluaciones del crítico de las muestras generados por el Generador . La ventaja principal de las WGAN es su capacidad para proporcionar gradientes más estables durante el entrenamiento, lo que evita problemas como el colapso de modo y el desvanecimiento de gradiente. En lugar de maximizar una divergencia como lo hacen las GAN tradicionales, las WGAN minimizan la distancia de Wasserstein, lo que equivale a evaluar el nivel de error en las observaciones clasificadas. Este resultado, al no estar condicionado a un valor entre cero y uno evita los problemas antes mencionados. Vale señalar que la distancia de Wasserstein es continua y diferenciable, lo que implica que el Crítico puede ser entrenado hasta alcanzar un valor óptimo. Cuanto mayor sea el entrenamiento, más confiable es el valor de su gradiente. A diferencia de las DCGAN, el beneficio de las WGAN es que el proceso de entrenamiento es menos sensible a la arquitectura del modelo y a la configuración de sus hiperparámetros [22].

Para entrenar una WGAN, el Crítico debe cumplir la condición de continuidad 1-Lipschitz. Esto implica que la norma de su gradiente debe ser como máximo uno. Lo que significa que la pendiente no puede ser en módulo mayor que uno en ningún punto. Esta condición es importante porque asegura que la función de costos sea continua y diferenciable, permitiendo un mayor grado de estabilidad (en referencia a la continuidad del cálculo del gradiente) durante el entrenamiento [1]. Para asegurar el cumplimiento de continuidad 1-L se puede emplear la técnica de restricción de peso ("weight clipping" en inglés). Este limita los valores de los pesos del crítico dentro de un rango específico permitiendo mantener los gradientes bajo control y evitar que los valores de los pesos se disparen durante el entrenamiento. Si bien esta es una técnica efectiva y de fácil implementación, forzar los pesos del crítico a un rango limitado de valores podría provocar la pérdida de su capacidad de aprendizaje, esto se lo conoce como "clipping collapse" en inglés [4].

Otra alternativa es utilizar algún método de regularización; penalizando la magnitud de aquellos pesos que exceden cierto umbral. Una técnica común es la regularización de gradiente, que agrega un término de penalización en la función de costo basado en la norma L2 de los gradientes. A continuación se presenta la expresión de la función de costos regularizada en una WGAN:

$$\min_G \max_C E(C(x)) \; - \; E(C(G(z))) + \lambda_{reg}$$

La regularización permite un mayor control sobre la penalización de los gradientes y puede evitar problemas de "clipping collapse". Sin embargo, puede requerir una configuración más cuidadosa de los hiperparámetros [4].

Función de pérdida de Wasserstein

La función de pérdida de Wasserstein busca incrementar la distancia entre elementos reales y falsos. Se puede resumir como:

1. Función de pérdida del Crítico = Promedio de puntaje sobre datos reales - Promedio de puntaje sobre datos generados.

2. Pérdida del generador = - Promedio puntaje sobre datos generados

Los promedios se calculan sobre los lotes de datos. Esto es precisamente cómo se implementa la función de pérdida. En el caso del generador, una puntuación mayor del crítico resultará en una pérdida menor para el generador, lo que alentará al crítico a producir mayores puntuaciones para imágenes falsas. Por ejemplo, una puntuación media de 10 se convierte en -10, una puntuación media de 50 se convierte en -50, que es más pequeño, y así sucesivamente.

En el caso del crítico, por ejemplo, una puntuación mayor para imágenes reales resulta en una pérdida mayor para el crítico, penalizando al modelo. Esto anima al crítico a producir puntuaciones más pequeñas para imágenes reales. Por ejemplo, una puntuación media de 20

para imágenes reales y 50 para imágenes falsas da como resultado una pérdida de -30; una puntuación media de 10 para imágenes reales y 50 para imágenes falsas da como resultado una pérdida de -40, que es mejor, y así sucesivamente. El signo de la pérdida no importa en este caso, siempre que la pérdida de imágenes reales sea una cantidad pequeña y la pérdida de imágenes falsas sea una cantidad grande. La pérdida de Wasserstein anima a los críticos a separar estas cifras. También podemos revertir la situación y hacer que el crítico genere una puntuación grande para imágenes reales y una puntuación pequeña para imágenes falsas y lograr el mismo resultado. Algunas implementaciones realizan este cambio.

Matemáticamente, si se toma por un lado la distribución de probabilidad asociada al conjunto de datos p_{data} y una variable $x_{real} \sim p_{data}$. Por otro lado la distribución de probabilidad asociada a los datos generados por el generador p_g y otra variable $x_g \sim p_g$. La distancia de Wasserstein $W(p_{data}, p_g)$ de orden p entre ambas distribuciones, se define como:

$$W_p(p_{data}, p_g) = \left(\inf_{\pi \in \Pi(p_{data}, p_g)} \iint_{supp(p_{data}) \times supp(p_g)} \rho(x_{data}, x_g)^p \, d\pi(x_{data}, x_g) \right)^{\frac{1}{p}}$$

donde $\rho(x, y)$ es una función de distancia, $\Pi(p_{data}, p_g)$ es el conjunto de todas las medidas de probabilidad sobre la distribución conjunta $p_{data} \times p_g$ con marginales p_{data} y p_g [23].

En resumen, las WGAN son una variante de las GAN que utilizan la distancia de Wasserstein como métrica de evaluación y aplican restricciones en los pesos para mejorar la estabilidad del entrenamiento. Estas mejoras han demostrado ser efectivas en la generación de imágenes de alta calidad y en la solución de desafíos asociados con las GAN tradicionales.

Generación condicional

Una vez que un modelo GAN genera un conjunto de datos de calidad y con buena distribución respecto a la variabilidad del dominio, podría no ser de utilidad si no es posible indicar al generador características específicas sobre los datos. Es de interés contar con herramientas de control que permitan especificar características particulares de la salida deseada. Este procedimiento se denomina generación condicional [24].

En la generación condicional se deberá indicar al modelo que genere diferentes elementos bajo ciertas especificaciones o condiciones y adaptar el proceso de entrenamiento. De este modo, como se verá a lo largo del capítulo, se obtendrá una generación condicionada. Es preciso descubrir cómo adaptar las entradas del modelo sin cambiar el mismo y ni su proceso de entrenamiento.

La generación condicional requiere un conjunto de datos etiquetados y de alguna manera transferir esta información tanto al generador como al discriminador durante el entrenamiento. Se utilizará un vector que llamaremos vector de clase para indicar al generador la clase de la cual deben provenir los ejemplos generados. Por lo general, este es un vector denominado "one-hot", en donde se utilizan ceros en cada posición, excepto en la posición correspondiente a la clase deseada donde va un 1. Este vector se concatena con el vector de ruido permitiendo generar datos aleatorios dentro de un subconjunto específico del dominio.

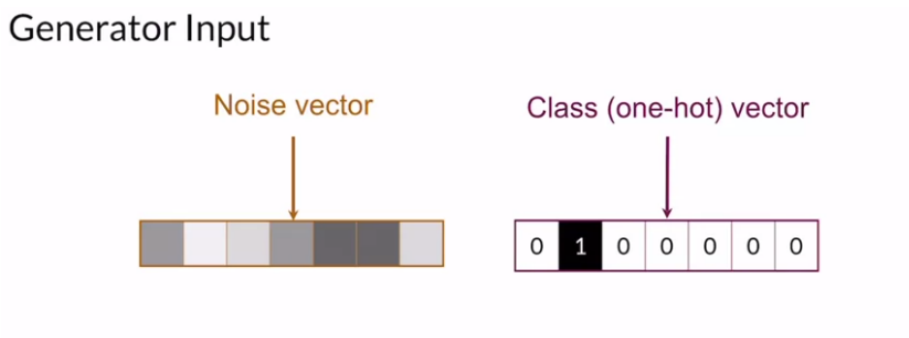
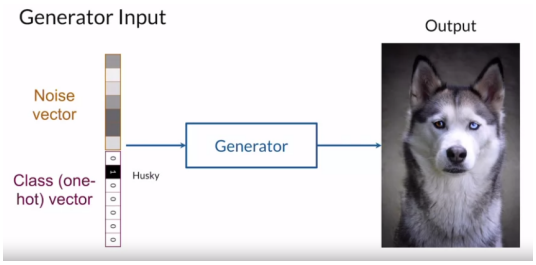
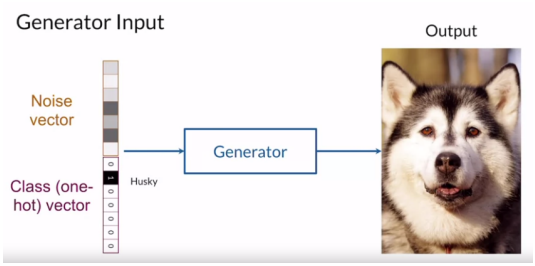


Figura 50: Vectores de entrada necesarios para la generación condicional [4]

Supongamos que el vector de clase especifica razas de perros y que un uno en la segunda posición del mismo, indica la raza Husky. Si se tuviera un uno en otra posición y un cero en la segunda, se indicaría una raza diferente; por ejemplo, un Golden Retriever. El vector de ruido agrega aleatoriedad permitiendo producir distintos tipos de perros Husky.



Generando un perro de una raza Husky [4]



Generando nuevo perro de una raza Husky [4]

Si cambia ese vector de ruido manteniendo fijo el vector de clase, se producirá un perro diferente de la misma raza, como se muestra en la imagen 3.3.

Pero, ¿cómo se asegura de que el generador realmente produzca un ejemplo de raza Husky? ¿Por qué no se produce cualquier perro al azar? Eso se debe a que el discriminador también recibe información respecto a la clase del objeto. El discriminador empareja los ejemplos recibidos con la información de la clase a la que pertenecen para determinar si son representaciones reales o falsas de esa clase en particular.

Supongamos la clase de Golden Retriever, pero en imagen de perros Beagle. El resultado del discriminador será falso porque no es un Golden Retriever de aspecto real, aunque sea una imagen muy real de un Beagle.

Como se muestra en la figura 51 se aprecia la imagen de un Beagle real a la que el Discriminador le asigna solo un 5 % de veracidad. Por lo tanto, el generador necesita producir un Golden Retriever realista como se muestra en 52.

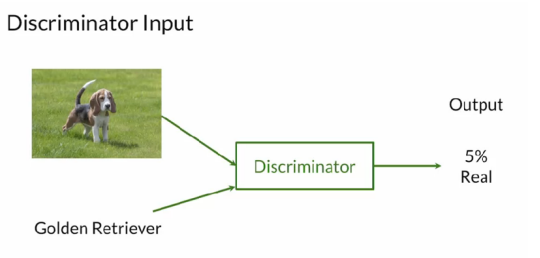


Figura 51: Discriminando un perro de una raza Golden Retriever [4]

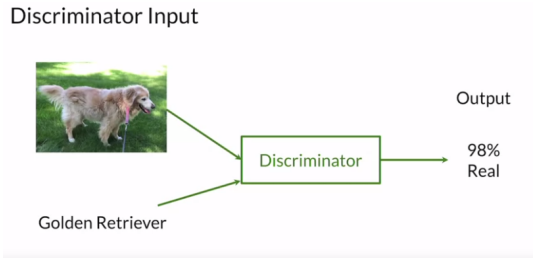


Figura 52: Discriminando un perro de una raza Golden Retriever [4]

Generación controlada

La generación de datos controlada es otra forma de controlar las salidas producidas por una GAN a diferencia de la generación condicional que aprovecha las etiquetas durante el entrenamiento, la generación controlada otorga el poder de manipular las características deseadas en los ejemplos de salida sin la necesidad de tener los datos etiquetados. Este control o ajuste de características puede hacerse a través de modificaciones en el vector de ruido de entrada z, que alimenta al generador después de entrenar el modelo [4].

Por ejemplo, supongamos que con el vector de ruido de entrada z, obtenemos una imagen de una mujer pelirroja como se muestra en la figura 54 se podría controlar la edad, si usa lentes, el color de cabello, etc.

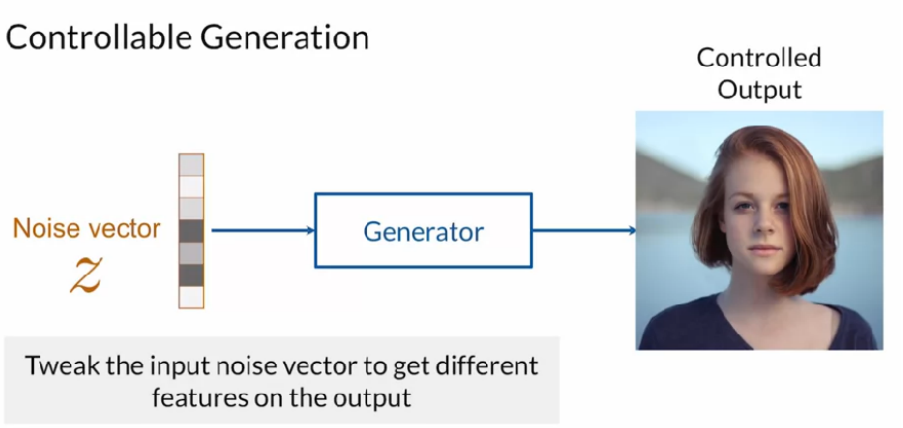


Figura 53: Ejemplo generación de un rostro humano a partir de una GAN [4]

Digamos que se modifica una de las características de este vector de ruido y ahora se obtiene la misma mujer, pero esta vez con cabello azul como se muestra en la figura 54.

La generación controlada implica controlar determinadas características de los datos. Generalmente se utiliza sobre variables de dominio continuo, mientras que la generación condicional permite especificar qué clase de observación se desea generar de lo utilizado para entrenar el modelo.[25]. Con la generación controlada, es posible obtener ejemplos de características singulares, mientras que con la generación condicional, se obtienen ejemplos de una clase determinada. La generación controlada funciona ajustando el vector de ruido de entrada z que

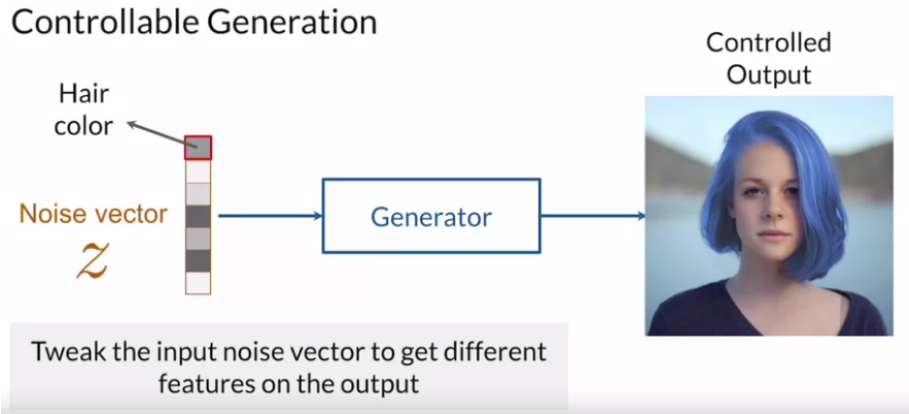


Figura 54: Ejemplo control de lo generado a partir de una GAN [4]

alimenta al generador, mientras que con la generación condicional, se debe pasar información adicional que represente la clase de los datos.

La generación controlada se logra manipulando el vector de ruido z que alimenta al generador. Al interpolar dos observaciones generadas podemos ver cómo una imagen se transforma en otra. Al obtener ejemplos intermedios manipulando las entradas del espacio Z tal como se muestra en la figura 55, donde el dígito ocho se transforma en nueve.

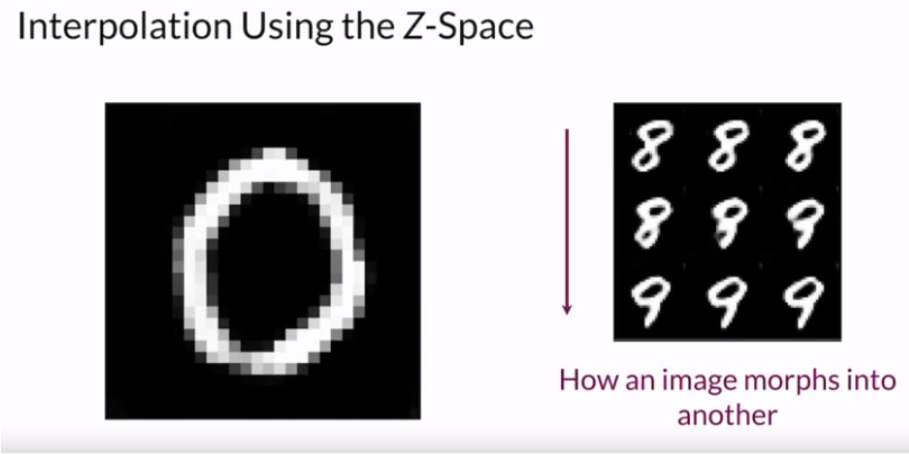


Figura 55: Interpolación de dígitos [4]

A modo de ejemplo, sea el vector z de dimensión 2 con ejes z_1 y z_2. Se tienen dos vectores de ruido v_1 y v_2, con las coordenadas $(z_1, z_2) = (5, 10)$ y $(z_1, z_2) = (4, 2)$ respectivamente. Los vectores v_1 y v_2 representan valores vectoriales concretos en el espacio Z. Cuando v_1 alimenta al generador, se produce la imagen etiquetada con $g(v_1)$ y v_2 produce la imagen etiquetada con $g(v_2)$.

Se desea obtener valores intermedios entre estas dos imágenes. Para ello, se interpolan sus dos vectores de entrada, v_1 y v_2 en el espacio Z. Generalmente se utiliza una interpolación lineal, aunque existen otros métodos. Por lo tanto, es posible obtener un conjunto de vectores intermedios y luego ingresarlos al generador y ver las imágenes resultantes como se muestra

en la figura 56.

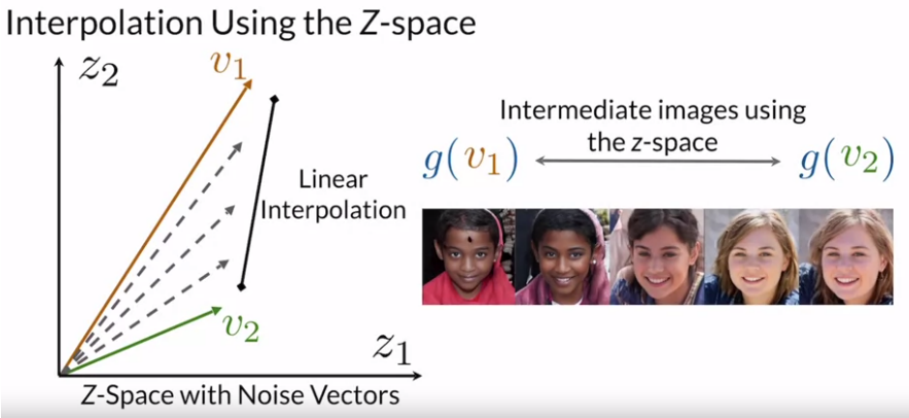


Figura 56: Interpolación de dígitos [4]

Por otro lado, la generación controlada también usa y aprovecha cambios en el espacio Z como las modificaciones de los vectores de ruido, que también se verán reflejadas en la salida del generador. Como se muestra en la figura 57, con el vector de ruido se podría obtener una imagen de una mujer con cabello rojo o la misma mujer con cabello azul.

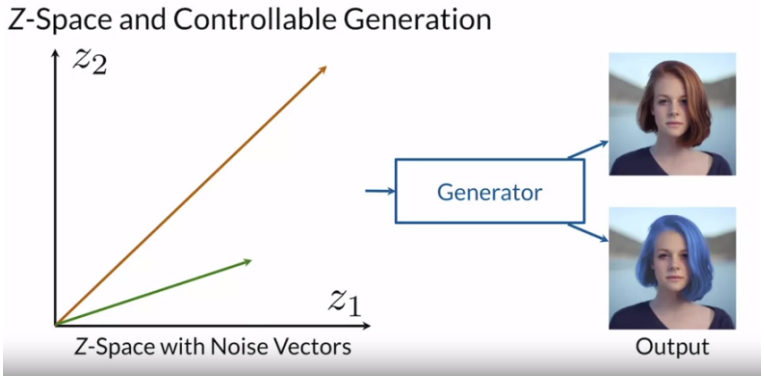


Figura 57: Generación controlada - Modificando el color de pelo [4]

Modificar el color del cabello de las imágenes generadas, requiere conocer la dirección en el espacio Z para ir del vector origen que genera el cabello rojo hacia el vector destino que genera el cabello azul. Con la dirección apropiada ,que se denota d, es posible controlar este cambio.

Para encontrar la dirección en el espacio Z que modifica ciertas características en la salida, se puede utilizar un clasificador que identifique si una imagen tiene esa característica. Se podría tomar un lote de vectores de ruido, pasarlos por el generador y obtener algunas imágenes. Luego se pasan estas imágenes por un clasificador que determinará cuáles imágenes tienen la característica deseada. Se repite este proceso hasta que las imágenes se clasifiquen correctamente. Este método se realiza después de haber entrenado la GAN. Tiene como desventaja la necesidad de un clasificador previamente entrenado. Para encontrar las direcciones usando los gradientes de los clasificadores, es preciso modificar los vectores de ruido sin cambiar el

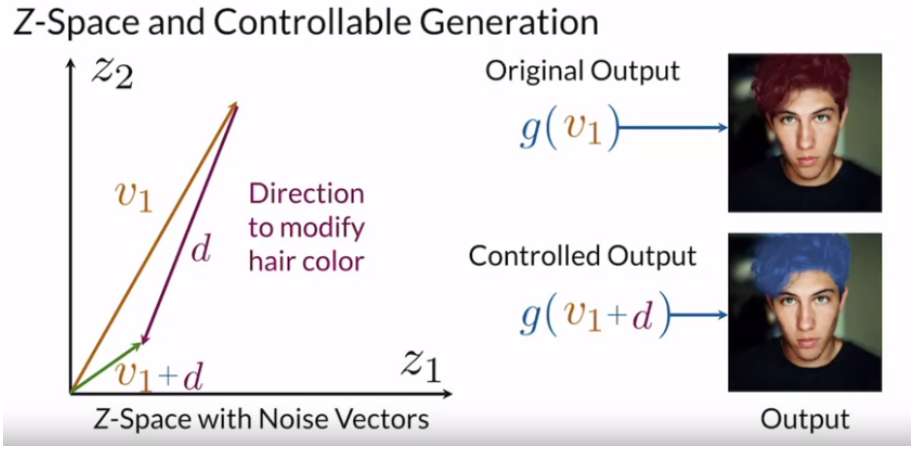


Figura 58: Generación controlada - Modificando el color de pelo con la dirección d [4]

generador. La generación controlada presenta desafíos a la hora de modificar una característica deseada. Específicamente, la correlación de características afecta la realización de estas modificaciones. Cuando diferentes características tienen una alta correlación en el conjunto de datos de entrenamiento, se vuelve difícil controlar características específicas sin modificar otras correlacionadas.

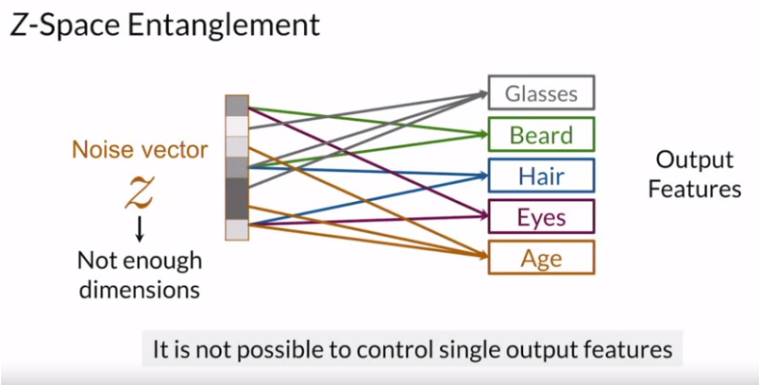


Figura 59: Generación controlada - Efecto del espacio entrelazado Z [4]

Idealmente sería posible controlar características únicas, como la cantidad de barba en la cara de una persona. Si las características en el conjunto de datos no tienen una alta correlación, podría tomar la imagen de una mujer y agregarle barba moviéndose en alguna dirección en el espacio Z. Sin embargo, es muy probable que en el conjunto de datos de entrenamiento, características como la presencia de una barba y el aspecto masculino de la cara estén fuertemente correlacionadas. Por lo tanto, si deseamos agregar una barba a la imagen de una mujer, terminaremos modificando otras características correlacionadas como se muestra en la siguiente figura 60. Es posible que la GAN devuelva una imagen bastante realista con barba, pero con modificando también otras características debido a la correlación entre la presencia de barba y la masculinidad de las imágenes dentro del conjunto de entrenamiento. Es posible que este efecto no sea el deseado, dado que se busca encontrar direcciones donde se pueda cambiar únicamente una característica y de esa manera editar imágenes de manera confiable [4].

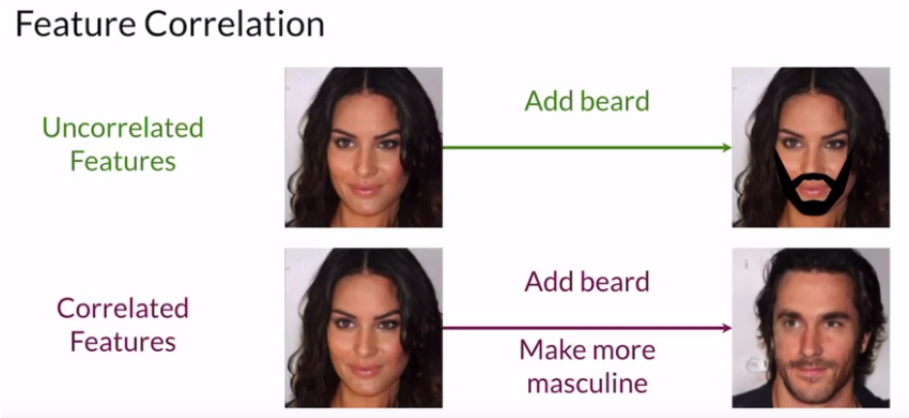


Figura 60: Generación controlada - Efecto de agregar barba al rostro de una mujer [4]

Otro desafío que enfrenta la generación controlada, se conoce como entrelazamiento en el espacio Z. Cuando el espacio Z está entrelazado, el movimiento en las diferentes direcciones tiene un efecto en múltiples características simultáneamente. Incluso si estas características no están necesariamente correlacionadas en el conjunto de datos de entrenamiento. Esto se debe a la forma en que aprendió el modelo en relación al espacio Z y su efecto es que el Z quede enredado o entrelazado.

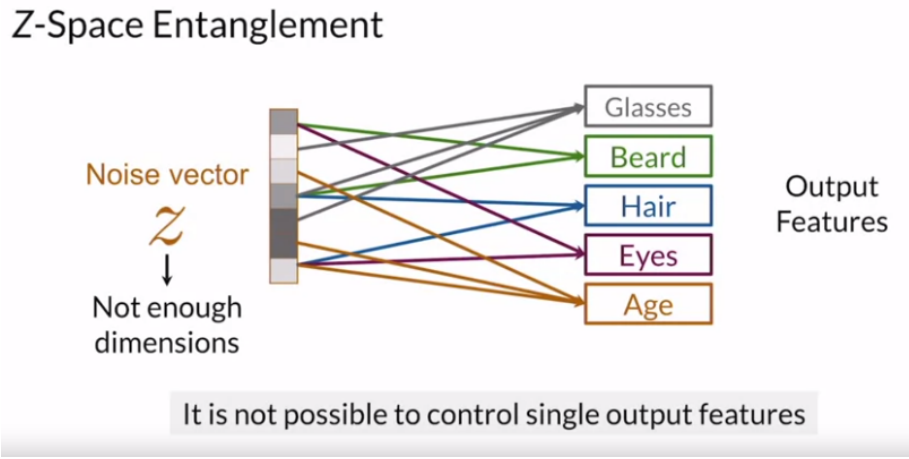


Figura 61: Generación controlada - Efecto del espacio entrelazado Z [4]

En un espacio Z entrelazado, si se controla la presencia de gafas en la salida también se podría modificar si tiene barba. Eso no es del todo deseable pues significa que los cambios en algunos de los componentes de los vectores de ruido cambian múltiples características en la salida al mismo tiempo. Este es un problema muy común cuando el espacio Z no tiene suficientes dimensiones en relación a la cantidad de características que desea controlar debido a que no se pueden mapear uno a uno.

En resumen, la generación controlada enfrenta desafíos importantes. Uno refiere a cuando las características del conjunto de datos tienen una alta correlación entre sí. El siguiente,

refiere a que incluso si las características que se desea controlar no tienen correlación entre sí en el conjunto de datos de entrenamiento, el control de una característica también puede ser difícil si su espacio Z se encuentra entrelazado. Esto sucederá comúnmente si el número de dimensiones en el espacio Z no es lo suficientemente grande, aunque existen otras razones por las que eso también sucede [4].

Desentrelazado del espacio Z

En esta sección, se presentan métodos de desenredo y algunas formas de incorporarlo en un modelo GAN. En primer lugar, se plantea el significado de un espacio Z entrelazado en comparación con el espacio Z desentrelazado. Luego se mencionan algunas de las formas más populares de modificar el modelo a efectos de tener un espacio Z desentrelazado.

A continuación se presenta un ejemplo que plantea estos conceptos. Se toman los siguientes vectores de ruido v_1 y v_2, en el espacio Z y supongamos que provienen de un espacio Z desentrelazado. A la derecha en la figura 62 se visualizan dos de las n dimensiones del espacio Z. Dado que este es un espacio desentrelazado, cada una de las posiciones corresponde a una sola característica. La primera dimensión corresponde al color del cabello y la segunda dimensión a su longitud. Si se quisiera cambiar el color, se debe modificar el primer elemento en su vector de ruido en la dirección z_1. Mientras que para modificar su longitud, se modifica la dimensión z_2.

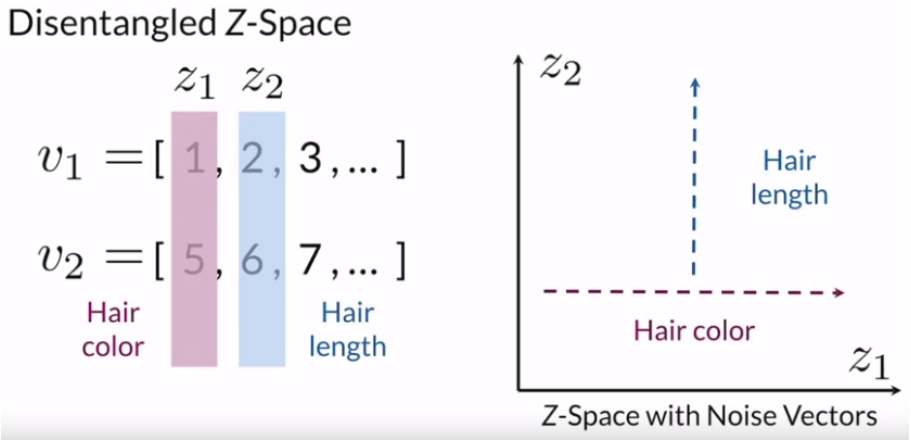


Figura 62: Generación controlada - desentrelazado de espacio Z [4]

Para facilitar el aprendizaje del modelo, es deseable contar con una alta dimensión en el espacio Z. Por ejemplo, si solo se desea controlar estas dos características, como el color y la longitud del cabello, probablemente sea prudente hacer que su vector de ruido tenga más de dos dimensiones. Estos otros valores no controlarán una característica específica, sólo ayudarán al modelo a adaptarse durante el entrenamiento debido a que los componentes de los vectores de ruido en el espacio Z desentrelazado, permiten cambiar las características deseadas en la salida. A esto se lo conoce como factores de variación latentes [4].

Una forma de incorporar en el modelo espacios Z desentrelazados, es etiquetar sus datos y seguir un proceso similar al que se usa para la generación condicional. Pero en este caso,

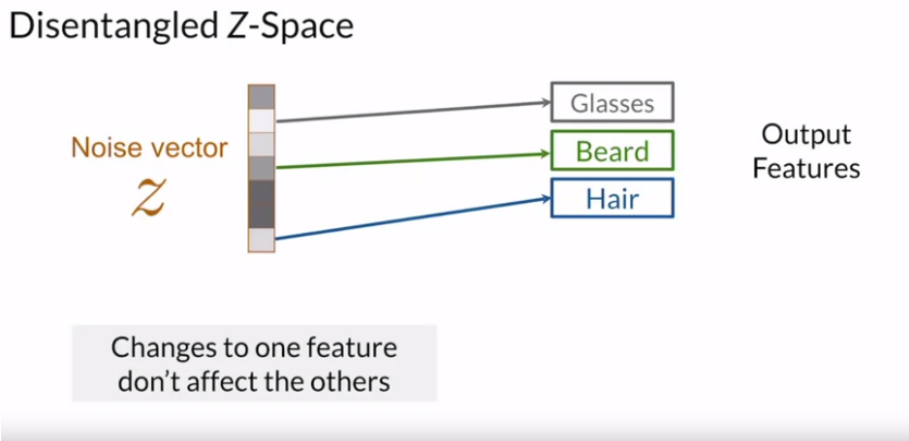


Figura 63: Generación controlada - desentrelazado de espacio Z [4]

la información de la clase está incrustada en el vector de ruido; por lo que no necesita esta información adicional de una sola clase o vector de clase. Sin embargo, el uso de este método podría resultar problemático para las clases continuas; por ejemplo, si tenemos que etiquetar miles de rostros humanos con la longitud de su cabello. Otra forma de utilizar un espacio Z desenredado sin etiquetar ninguno de sus ejemplos, es agregar un término de regularización a la función de pérdida para incorporar en el modelo la asociación con cada índice de los vectores de ruido a las diferentes características [4].

4. Puesta en práctica de redes neuronales y GAN

En esta sección, se detalla paso a paso la implementación de redes neuronales para tareas de clasificación, así como la implementación de redes neuronales adversarias (GAN) para la generación de dígitos manuscritos utilizando el conjunto de datos MNIST.

El inicio de la sección introduce una breve descripción de la librería Keras y del conjunto de datos MNIST, que se emplearán para implementar los modelos. Luego se presenta una sección dedicada a la implementación de un clasificador de imágenes; el objetivo principal es implementar una red neuronal que permita identificar a partir de la imagen de un número escrito a mano, cuál es el dígito correspondiente. La siguiente sección detalla la implementación de una red neuronal adversaria convolucional, también conocida como DCGAN para la generación de dígitos manuscritos. Posteriormente, se presenta una sección dedicada a mencionar criterios para la implementación de una versión modificada del modelo, que generan una GAN con características más estables en comparación con el primer modelo.

Para cada modelo, se analizará detalladamente la implementación de cada componente, su proceso de entrenamiento, evaluación y los resultados obtenidos. Se publica un repositorio que contiene el código completo del trabajo realizado, en el siguiente enlace: https://github.com/MATHIASFUIDIO/PROYECTO_GAN.

Interfaz de programación de aplicaciones y conjunto de datos

Keras es una API (del inglés, application programming interface) de alto nivel de la plataforma TensorFlow que proporciona una interfaz para resolver problemas de Aprendizaje Automático, con un enfoque en el aprendizaje profundo permitiendo el diseño de redes neuronales de forma sencilla y eficiente. Keras está escrito en el lenguaje de programación Python y permite implementar todo el ciclo de vida del desarrollo de modelos de aprendizaje profundo como las redes neuronales. En el ciclo de vida del desarrollo de una red neuronal encontramos las siguientes etapas: definición, compilación, ajuste y evaluación de un modelo. Así mismo Keras nos permitirá hacer predicciones a través de los modelos desarrollados [22].

El conjunto de datos MNIST es un acrónimo que significa Instituto Nacional Modificado de Estándares y Tecnología (en inglés, Modified National Institute of Standards and Technology). Es un conjunto de datos de 70.000 imágenes cuadradas pequeñas en escala de grises de 28×28 píxeles de dígitos únicos escritos a mano entre 0 y 9.

Keras proporciona acceso al conjunto de datos MNIST a través de la función *loaddataset*(). La misma devuelve dos tuplas, una con los elementos de entrada y salida para el conjunto de datos de entrenamiento, y otra con los elementos de entrada y salida para el conjunto de datos de prueba.

Figura 64: Ejemplo observaciones del conjunto MNIST

Cada elemento de estas tuplas representa una imagen de 28×28 píxeles en escala de grises y su correspondiente etiqueta. El conjunto de datos de entrenamiento tiene 60.000 observaciones y el de prueba 10.000. Cada píxel puede tomar un valor entre 0 y 255 que representan los valores blanco o negro respectivamente. Se muestran algunos ejemplos en la figura 64

4.1. Clasificador de dígitos manuscritos

El siguiente modelo tiene por objetivo predecir cuál es el dígito contenido en la imagen. Para ello, comenzaremos utilizando una red neuronal simple mediante dos capas densas totalmente conectadas. Luego intentaremos mejorar los resultados con una segunda red neuronal que utilice capas de convolución y de "pooling". Finalmente haremos una breve comparación de resultados.

Previo a entrenar el modelo se realizan tres ajustes fundamentales en los datos para permitir un correcto entrenamiento.

- **"One hot encoding":** aplicaremos esta técnica a la etiqueta de nuestros datos. La misma consiste en crear una columna binaria para cada valor único que existe en la variable respuesta (todos los dígitos del 0 al 9), y marcar con un 1 la columna correspondiente al valor presente en cada registro, dejando las demás columnas con un valor de 0.

Etiqueta
3
1
0
7
9

-->

0	1	2	3	4	5	6	7	8	9
0	0	0	1	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	1

Figura 65: Ejemplo de una transformación de etiquetas por el método One Hot Encoding

- **Escalado entre 0 y 1:** los modelos presentan mejores características de eficiencia cuando los datos están en una escala acotada. Por lo que transformaremos el valor de cada píxel de una escala de 0 a 255 a otra escala entre 0 y 1.

- **Redimensionamiento de los datos:** pasamos de tener ordenados los píxeles de las imágenes en una matriz de dos dimensiones (28×28) a un vector de 784 elementos. Dado que nuestro primer modelo será una red conteniendo dos capas densas, deberemos aplanar el conjunto de datos de modo de ingresar cada imagen como un vector de píxeles.

Arquitectura del clasificador

El clasificador será una red neuronal que consta de dos capas de 128 nodos cada una. La función de activación a la salida de cada capa, será ReLu. Entre cada una de estas, se utiliza una capa de "Dropout" a efectos de mejorar la capacidad de generalización del modelo. La capa de salida tendrá 10 nodos y como activación la función softmax (utilizada en casos de clasificación de más de dos categorías). Se utilizará la función de pérdida entropía cruzada categórica dado que se trata de una clasificación de más de dos categorías, Accuracy como métrica y el optimizador Adam para calcular el descenso por gradiente (en el apéndice 6.2 se describe el funcionamiento de este optimizador).

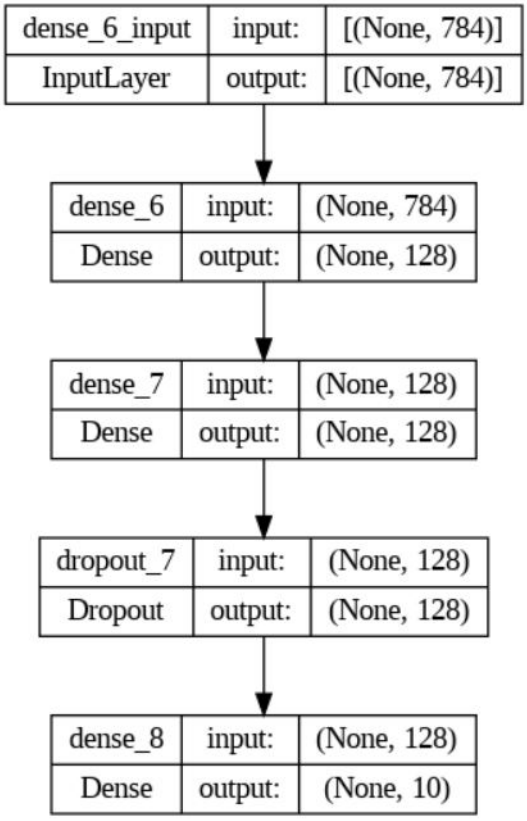


Figura 66: Arquitectura red neuronal clasificadora de dígitos manuscritos

El modelo tiene un total de 118.282 parámetros entrenables. Para el entrenamiento utilizaremos un tamaño de lote de 512 imágenes y 10 épocas. Estos hiperparámetros pueden variar y su valor afecta el resultado del entrenamiento.

Evaluación del clasificador

El modelo entrenado devuelve como resultado un vector que contiene una estimación de la probabilidad de que la nueva observación corresponda a cada una de las clases. Aquella clase con mayor probabilidad es la clase predicha por el modelo. Para evaluar el modelo, se observará el nivel de acierto frente a un nuevo conjunto de datos no utilizado previamente para su entrenamiento. El nivel de accuracy del modelo es de 97,6 %.

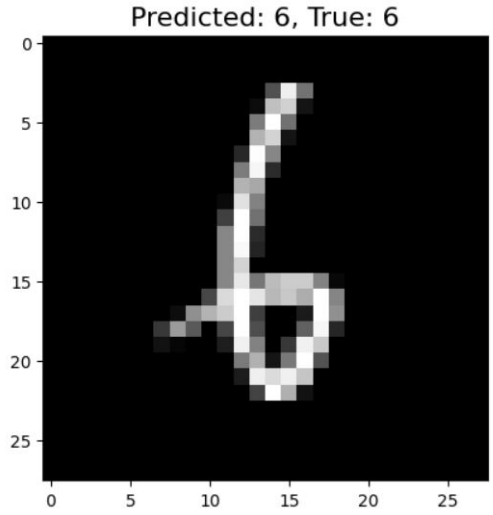


Figura 67: Ejemplo predicción clasificador

Si bien se conoce el nivel global de acierto del modelo, no se conoce el nivel de precisión para un dígito específico. Es posible que el modelo tenga un nivel de precisión diferente frente a distintos dígitos. Para ello, se presenta en la figura 68 una matriz de confusión a efectos de realizar este estudio. A modo de ejemplo, al analizar las predicciones de las imágenes correspondientes al dígito 0, se observa que:

- 972 elementos fueron correctamente predichos.
- ningún elemento fue predicho como 1.
- 1 elemento fue predicho como 2.
- 1 elemento fue predicho como 3.
- ningún elemento fue predicho como 4.
- 1 elemento fue predicho como 5.
- 1 elemento fue predicho como 6.
- 1 elemento fue predicho como 7.
- 2 elementos fueron predichos como 8.
- 1 elemento fue predicho como 9.

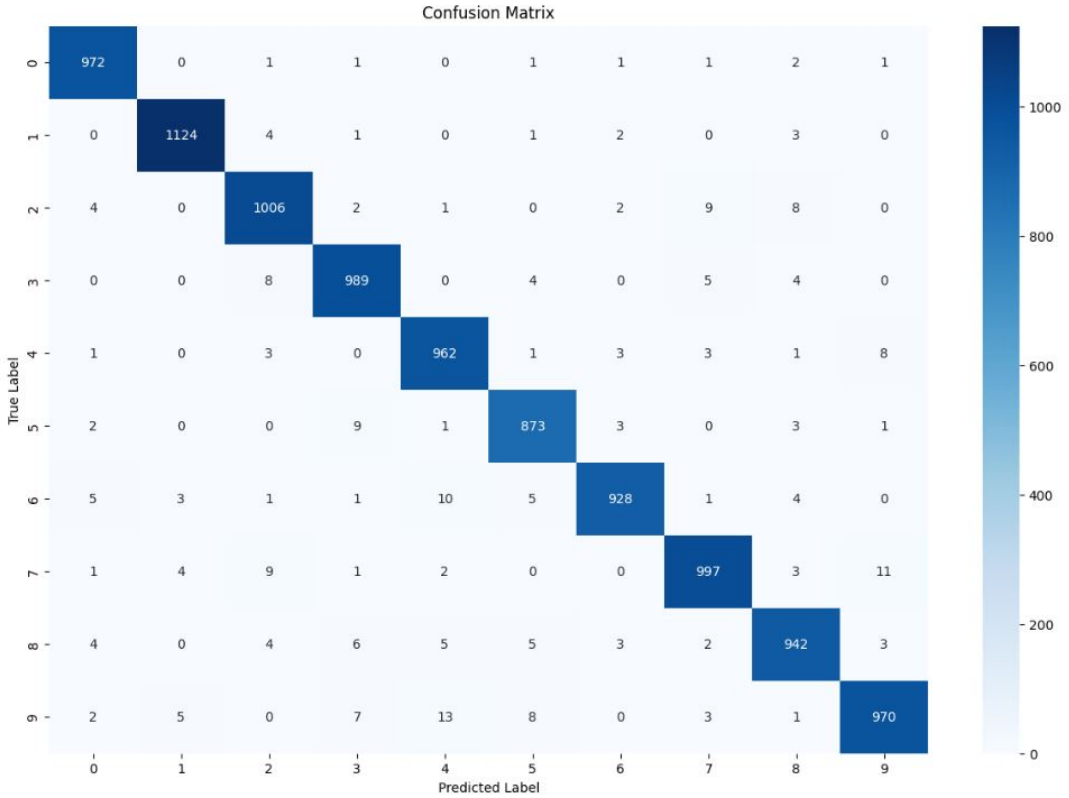


Figura 68: Matriz de confusión de modelo clasificador en muestra de evaluación

Para completar el análisis, nos resultó de interés poder observar las imágenes en las que el modelo se equivocó con mayor probabilidad. En otras palabras, nos referimos a aquellas que confundieron más al modelo.

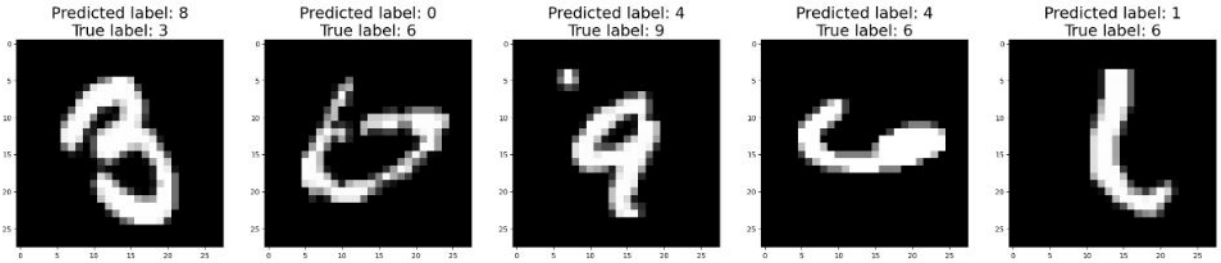


Figura 69: Ejemplos de errores del clasificador

Observando las imágenes en la figura 69 se puede decir que se trata de casos que no son del todo claros y es razonable que el modelo se equivoque. La primera imagen corresponde al número 3, el modelo se confunde con un 8 dado que las curvas son bastante cerradas; la segunda, cuarta y la quinta imagen, corresponden al dígito 6, aunque a simple vista se puede comprender porque el modelo propone clasificarlos en 0, 4 y 1 respectivamente. La imagen más clara es la tercera que corresponde a un 9, pero tiene un píxel atípico que podría haber afectado la clasificación del modelo.

Clasificador convolucional de dígitos manuscritos

Dado que nuestro segundo modelo recibirá directamente una imagen de 28×28, será necesario en lugar de aplanar los datos como en el modelo anterior, ajustar los datos para que tengan una dimensión más y queden de 28×28×1. Esto se debe a que la función de Keras que utiliza la operación de Convolución, está preparada para recibir imágenes de colores. Es decir, de 28×28×3, por su formato RBG de canales de colores en imágenes. Por lo que cuando se trata de imágenes en blanco y negro, se debe indicar que el conjunto cuenta con un único canal y por lo tanto, estas deberán estar en un formato de 28×28×1.

La operación Conv2D de Keras utiliza distintos filtros a efectos de destacar distintas características de una imagen. Esto permite hacer una extracción de características de la imagen como ser los bordes, líneas verticales y horizontales, líneas curvas, etc. Estos elementos mejoran la eficiencia de la red neuronal cuando se pasa toda la información a las capas densas del modelo. El entrenamiento de la red, permite que se calculen parámetros específicamente asociados a las características obtenidas en las capas anteriores de convolución y "pooling", facilitando el aprendizaje de la red y mejorando la eficiencia del modelo.

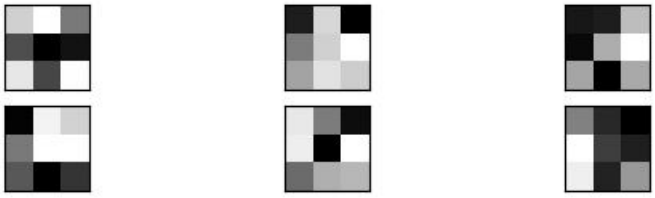


Figura 70: Ejemplo de filtros en Conv2D

Este segundo clasificador tiene por objetivo mejorar la precisión del modelo anterior. En este caso, será una red neuronal que consta de:

- una capa de convolución de 64 filtros y un núcleo de (3,3), seguido de una capa de "pooling" de (2,2).
- una segunda capa de convolución de 64 filtros y un núcleo de (3,3), seguido de una capa de "pooling" de (2,2).
- una capa de "Droput" con parámetro 0.25 que permite mejorar la capacidad de generalización del modelo desactivando de forma aleatoria el 25 % de las neuronas de la capa anterior inmediata.
- la operación Flatten() simplemente transforma la salida de la capa anterior a un único vector lineal. Ahora la salida se ve como una tira de neuronas.
- una capa densa de 64 nodos con función de activación ReLu.
- finalmente una capa de salida de 10 nodos que utilizará la función softmax.

Para la compilación del modelo se utiliza la función de pérdida entropía cruzada categórica dado que se trata de una clasificación de más de dos clases, Accuracy como métrica y el

optimizador Adam. Tamaño de lote 128 imágenes y 15 épocas. Totalizando 280.010 parámetros entrenables.

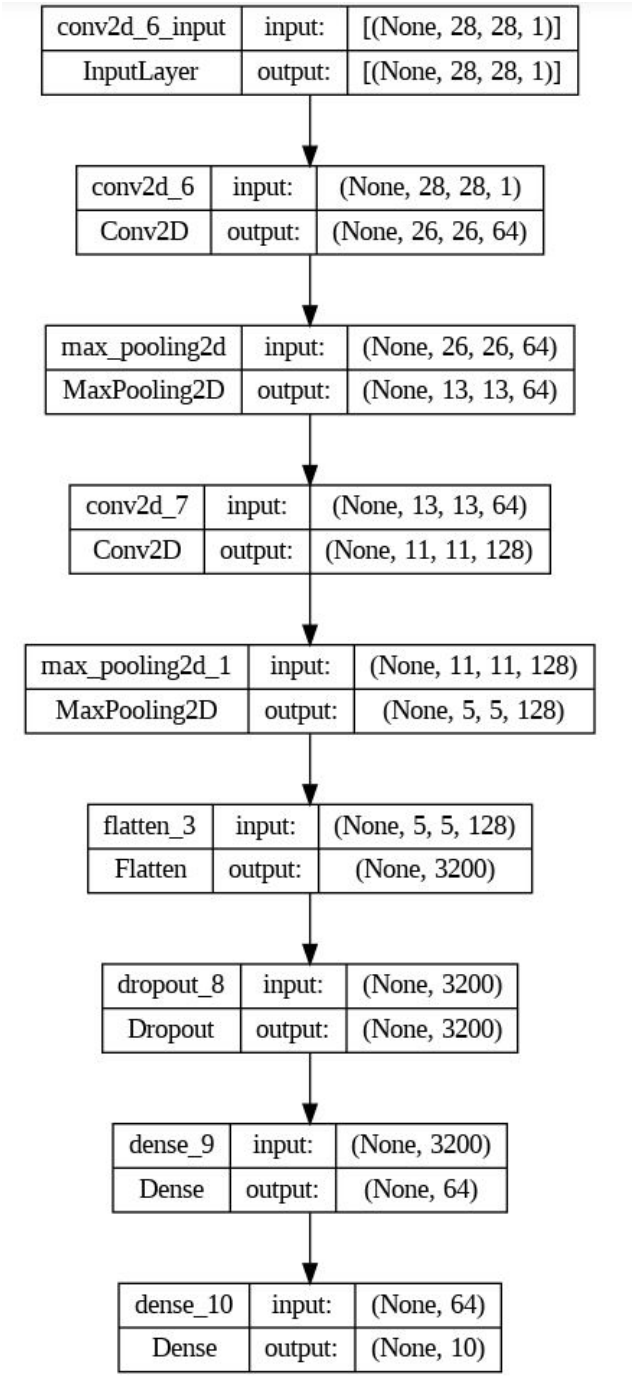


Figura 71: Arquitectura de clasificador convolucional

Luego de entrenar el modelo se observa una notoria mejoría a nivel de accuracy pasando de 97 % a 99,3 % en esta red neuronal convolucional. Como comentario adicional se destaca que al revisar los casos con mayor probabilidad de error, resultaron ser los mismos del modelo anterior.

4.2. GAN convolucional profunda

En esta sección se presenta el diseño e implementación de un modelo GAN utilizando redes neuronales de convolución. Este tipo de GAN, también se las conoce como DCGAN se debe a sus siglas en inglés de Deep Convolutional Generative Adversarials Networks. Tomaremos como ejemplo una arquitectura similar a la propuesta en [22].

El procedimiento que se verá para el desarrollo del modelo consta de las siguientes tareas:

- Definición y entrenamiento del Discriminador.
- Definición del Generador.
- Entrenamiento de la red GAN
- Evaluación del modelo

Al igual que en el caso anterior, utilizaremos el conjunto de datos MNIST y la API de Keras para desarrollar y entrenar el modelo. También se escalan los píxeles de las imágenes entre 0 y 1 y se redimensiona el conjunto de imágenes para que sean de la forma 28×28×1.

Discriminador: arquitectura y entrenamiento

El primer paso es definir el modelo Discriminador. En este caso, el modelo debe tomar una imagen y generar una predicción de clasificación sobre si la muestra es real o falsa (no predice la etiqueta correspondiente como en los modelos vistos anteriormente). Se trata entonces de un problema de clasificación binaria donde:

- Como entrada tendremos una imagen de un único canal de 28×28 píxeles.
- Como salida una probabilidad asociada a si la imagen es real o falsa.

El modelo discriminador tiene dos capas convolucionales con 64 filtros cada una, un tamaño de núcleo de 3×3 y un valor de "stride" de 2×2. Este modelo no tiene capas de "pooling" y un solo nodo en la capa de salida con la función de activación Sigmoide para predecir si la muestra de entrada es real o falsa. El modelo está entrenado para minimizar la función de pérdida de entropía cruzada binaria, apropiada para la clasificación binaria. Usaremos algunas de las mejores prácticas vistas anteriormente para definir el modelo Discriminador, como el uso de LeakyReLU en lugar de ReLU, el uso de "Dropout" y el uso de la versión Adam de descenso de gradiente estocástico con una tasa de aprendizaje de 0,0002 y un impulso de 0,5.

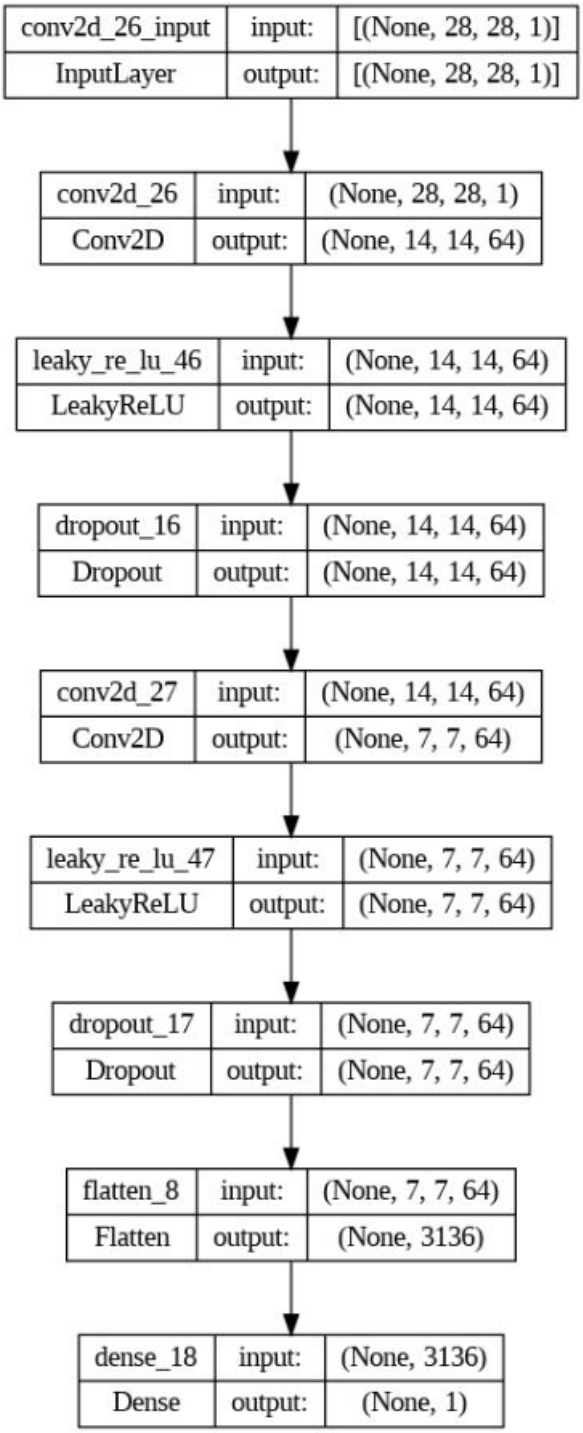


Figura 72: Arquitectura discriminador DCGAN

La figura 72 permite ver la arquitectura y muestra la entrada y la salida del modelo de cada capa. Se puede ver el efecto de utilizar un "stride" 2×2 y cómo reduce la salida de la capa de convolución pasando de una imagen de 28×28 a 14×14 y luego a 7×7, antes de que el modelo haga una predicción de salida. Este patrón es por diseño, genera un efecto similar a cuando utilizamos capas de "pooling".

Por último destacar que los parámetros del discriminador se ajustan en un proceso por lotes, de forma independiente y en momentos distintos al generador. Más adelante veremos cómo se realiza esta tarea. En algunos casos se suele realizar un paso cero de entrenamiento donde solo se actualizan los parámetros del discriminador (en nuestro ejemplo no implementamos esta estrategia).

Generador: Arquitectura y entrenamiento

El modelo generador es responsable de crear imágenes nuevas, falsas pero plausibles de dígitos escritos a mano. Para ello, toma un punto del espacio latente como entrada y genera una imagen cuadrada en escala de grises. El espacio latente es un espacio vectorial definido arbitrariamente de distribución gaussiana.

Al final del entrenamiento, el espacio vectorial latente se lo puede ver como una representación comprimida del espacio de salida que sólo el generador conoce cómo convertir.

- Entradas: punto en el espacio latente.
- Salidas: imagen cuadrada bidimensional en escala de grises de 28×28 píxeles con valores de píxeles en [0,1].

Desarrollar un modelo Generador requiere que transformemos un vector del espacio latente que en nuestro caso será de dimensión 100, a una matriz de dos dimensiones tamaño 28×28 o 784 elementos. Hay varias formas de lograrlo, pero hay un enfoque que ha demostrado ser eficaz en redes adversarias generativas convolucionales profundas. El mismo implica dos elementos principales:

1. la primera es una capa densa como primera capa oculta que tenga suficientes nodos para representar una versión de baja resolución de la imagen de salida. Específicamente, una imagen de la mitad del tamaño (un cuarto del área) de la imagen de salida que será de 14×14 o 196 nodos; y una imagen de un cuarto del tamaño (un octavo del área) que será de 7×7 o 49 nodos. Esto nos permite obtener muchas versiones o interpretaciones paralelas de la entrada. Este es un patrón en las redes neuronales convolucionales donde tenemos muchos filtros paralelos que dan como resultado múltiples mapas de activación, llamados mapas de características, con diferentes interpretaciones de la entrada. El modelo necesita espacio para inventar, crear o generar. Por lo tanto, la primera capa oculta necesita suficientes nodos para múltiples versiones de baja resolución de nuestra imagen de salida. La primera capa densa será de $7 \times 7 \times 128$.
2. la siguiente implica aumentar la resolución de la imagen a una versión de mayor resolución. Hay dos formas comunes de realizar este proceso de muestreo ascendente, llamado deconvolución. Una forma es utilizar una capa UpSampling2D (como una capa de agrupación inversa) seguida de una capa Conv2D normal. La otra forma, quizás más moderna, es combinar estas dos operaciones en una sola capa, llamada Conv2DTranspose. Usaremos este último enfoque para nuestro generador. La capa Conv2DTranspose se puede configurar con un paso de (2×2) que cuadruplicará el área de los mapas de características de entrada (duplicará sus dimensiones de ancho y alto). Esto se puede repetir

para llegar a nuestra imagen de salida de 28×28. Nuevamente, usaremos la activación LeakyReLU con una pendiente predeterminada de 0,2, lo que se considera una práctica recomendada al entrenar modelos GAN.

La capa de salida del modelo es Conv2D con un filtro y tamaño de núcleo de 7 × 7, diseñado para crear un mapa de características único y preservar sus dimensiones de 28 × 28 píxeles. Se utiliza una activación Sigmoide para garantizar que los valores de salida estén en el rango deseado de [0,1].

A diferencia del Discriminador, el modelo Generador no está compilado y no especifica una función de pérdida ni un algoritmo de optimización. Esto se debe a que el Generador no se entrena directamente.

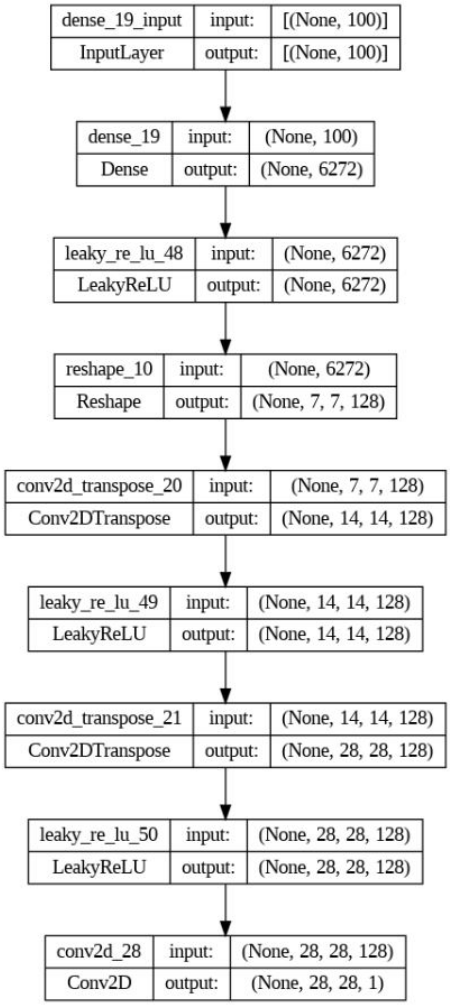


Figura 73: Arquitectura generador DCGAN

Observando la estructura del modelo Generador en 73, se muestran las capas del modelo y su forma de salida. Podemos ver que, tal como está diseñada, la primera capa oculta tiene 6.272 parámetros o 128×7×7, cuyas activaciones se remodelan en 128 mapas de características de 7×7. Este se actualiza a través de las dos capas Conv2DTranspose a la forma de salida deseada de 28×28×1. Totalizando 1.164.289 parámetros entrenables.

Los pesos en el modelo Generador se actualizan en función del rendimiento del modelo Discriminador. Cuando este último es bueno para detectar muestras falsas, el Generador se actualiza más y cuando el modelo discriminador es relativamente pobre o confuso al detectar muestras falsas, el modelo del generador se actualiza menos.

Para entrenar el Generador se crea una nueva arquitectura la cual llamaremos GAN donde se apila el generador y Discriminador de manera tal que el Generador reciba como entrada puntos aleatorios en el espacio latente y genere muestras que se introducen directamente en el modelo Discriminador. Estas muestras se clasifican y la salida de este modelo más grande se utiliza para actualizar los parámetros del modelo del Generador. Es importante destacar que en esta arquitectura los parámetros del Discriminador se encuentran fijos por lo cual en el proceso de entrenamiento no se actualizaran dichos parámetros. Al entrenar el Generador mediante este modelo GAN lógico, hay un cambio importante. Queremos que el Discriminador piense que las muestras generadas por el Generador sean reales, no falsas. Por lo tanto, cuando el Generador se entrena como parte del modelo GAN, marcaremos las muestras generadas como reales (clase = 1); para que el Discriminador clasifique entonces las muestras generadas que no son reales (clase = 0) o tienen una baja probabilidad de ser reales (0,3 o 0,5). Al pasarle imágenes falsas etiquetadas como verdaderas, el proceso de retropropagación (backpropagation) utilizado para actualizar los pesos del modelo verá esto como un error grande y actualizará los pesos del modelo (es decir, solo los pesos en el Generador) para corregir este error, lo que a su vez hará que el generador produzca muestras de mejor calidad.

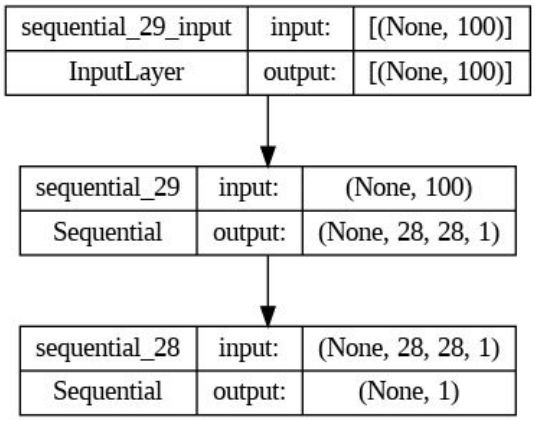


Figura 74: Arquitectura GAN combinación discriminador y generador

En resumen lo que se requiere es que primero actualicemos el modelo Discriminador con muestras reales y falsas, luego actualicemos el Generador a través del modelo compuesto. También requiere que enumeremos tanto las épocas como los lotes dentro de una época. En la función de entrenamiento completa hay elementos a tener en cuenta para la ejecución del modelo. Primero, la cantidad de lotes dentro de una época se define por cuantas veces el tamaño de lote entra en el conjunto de datos de entrenamiento. En este ejemplo se tiene un conjunto de datos de 60.000 muestras y un tamaño de lote de 256, redondeando hacia abajo tendremos 234 lotes por época. El modelo discriminador se actualiza una vez por lote combinando medio lote (128) de ejemplos falsos y reales (128) en un solo lote. Puede actualizar el Discriminador con cada medio lote por separado (recomendado para conjuntos de datos más complejos), pero

combinar las muestras en un solo lote será más rápido a largo plazo, especialmente cuando se entrena en hardware GPU. Finalmente, informamos la pérdida de cada lote. [22]

Es fundamental estar atento a la pérdida de lotes. La razón de esto es que una falla en la pérdida del Discriminador indica que el modelo Generador ha comenzado a generar ejemplos basura que el Discriminador puede detectar fácilmente. Es necesario supervisar la pérdida del Discriminador, Se espera que oscile entre 0,5 y 0,8 por lote en este conjunto de datos [22]. La pérdida del Generador es menos crítica y puede oscilar entre 0,5 y 2 o más. Se podría incluso intentar detectar la pérdida del Discriminador, detener y luego reiniciar el proceso de entrenamiento.

Evaluación resultados de DCGAN en MNIST

Generalmente, no existe una forma objetiva de evaluar el rendimiento de un modelo GAN. No podemos calcular una medida de error objetiva una imagen generada. Podría ser posible en el caso de imágenes MNIST porque las imágenes están muy restringidas, pero en general no es posible.

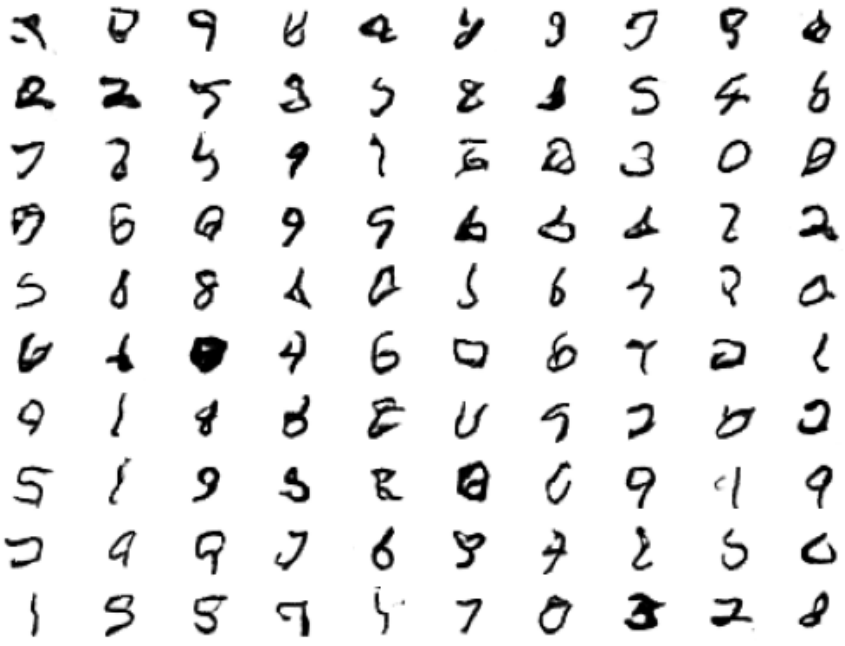


Figura 75: Ejemplos generados por modelo Generador con 100 épocas

Por lo tanto, un operador humano debe evaluar subjetivamente la calidad de las imágenes. Esto significa que no podemos saber cuándo detener el entrenamiento sin mirar ejemplos de imágenes generadas. A su vez, la naturaleza adversa del proceso de entrenamiento significa

que el Generador cambia después de cada lote, lo que significa que una vez que se pueden generar imágenes suficientemente buenas, la calidad subjetiva de las imágenes puede comenzar a variar, mejorar o incluso degradarse con actualizaciones posteriores.

Hay tres formas de manejar esta compleja situación de entrenamiento.

- Evaluar periódicamente el acierto de la clasificación del discriminador en imágenes reales y falsas.
- Generar periódicamente muchas imágenes y guardarlas en un archivo para revisión subjetiva.
- Guardar periódicamente el modelo del Generador.

Estas tres acciones se pueden realizar al mismo tiempo para una época de entrenamiento determinada, como cada cinco o diez épocas de entrenamiento. El resultado será un modelo Generador guardado para el cual tenemos una manera de evaluar subjetivamente la calidad de su salida y saber objetivamente qué tan bien se engañó al discriminador en el momento en que se guardó dicho modelo.

Finalizamos el capitulo enumerando algunas ideas para mejorar el modelo GAN antes presentado y que se recomienda explorar:

- **Activación y escalamiento de TanH**. Actualizar el ejemplo para usar la función de activación de Tanh en el generador y escalar todos los valores de píxeles al rango [-1, 1].
- **Cambiar espacio latente** Actualizar el ejemplo para utilizar un espacio latente mayor o menor y comparar la calidad de los resultados y la velocidad del entrenamiento.
- **Normalización por lotes** (Batch Normalization). Actualizar el Discriminador y/o el Generador para hacer uso de la normalización por lotes, recomendado para modelos DCGAN.
- **Configuración del modelo** Actualizar la configuración del modelo para utilizar Discriminadores y/o Generadores más profundos o más superficiales, experimentar con capas UpSampling2D en el generador

5. Conclusiones

Se estudió en profundidad la arquitectura, los métodos entrenamiento y evaluación de performance de las GAN, analizando distintos conceptos relativos al desarrollo y uso de este modelo. Esto requirió de un importante estudio de las redes neuronales profundas, en particular las redes de convolución en el contexto del procesamiento de imágenes.

Se aplicaron los conocimientos teóricos adquiridos a lo largo del trabajo, a través de la realización de casos prácticos utilizando un conjunto de datos real como es el conjunto de datos MNIST y nos permitió aprender a programar en el lenguaje de Python, utilizando una librería potente y en actual desarrollo como Keras. Estas herramientas son actualmente de las más utilizadas tanto en la industria como en la comunidad científica en las áreas de ciencia de datos e inteligencia artificial.

Se desarrollaron dos lineas de trabajo: por un lado se implementó un modelo discriminador de imágenes, capaz de clasificar las mismas con un alto nivel de precisión 99,3 % y, en segundo lugar, una arquitectura GAN convolucional con la capacidad de generar dígitos manuscritos plausibles.

A partir del estudio y la experiencia realizada con este modelo creemos que existe una variedad creciente en el ámbito de la inteligencia artificial para el soporte de la generación de datos (data augmentation) que favorece la realización de experimentos en problemas cuyos datos de entrenamiento son escasos o de alto costo de adquisición.

Índice de figuras

Índice de cuadros

Referencias

[1] Jakub Langr and Vladimir Bok. *GANs in action: deep learning with generative adversarial networks*. Manning Publications, Shelter Island, New York, 2019. OCLC: on1050335878.

[2] Ian J. Goodfellow, Jean Pouget-Abadie, M. Mirza, B. Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, and Yoshua Bengio. Generative adversarial networks. *ArXiv*, abs/1406.2661, 2014.

[3] Kailash Ahirwar. *Generative Adversarial Networks Projects*. Packt, 01 2019.

[4] Sharon Zhou. *Build Basic Generative Adversarial Networks*. Coursera, 2020. https://www.coursera.org/learn/build-basic-generative-adversarial-networks-gans/.

[5] Mauricio Delbracio, José Lezama, and Guillermo Carbajal. *Deep Learning for Computer Vision*. Fing, 09 2018. https://iie.fing.edu.uy/~mdelbra/DL2018/.

[6] Sebastián Raschka and Vahid Mirjalili. *Python Machine Learning*. Packt, 2019.

[7] Haiyan Liu Hailong Xi and Yu Zhang. Recognition and optimization algorithm of mnist dataset based on lenet5 network structure. *Army Armored Force Academy, Beijing 100072, China*, pages 323–324, 2018.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[9] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991.

[10] Moshe Leshno, Vladimir Ya. Lin, Allan Pinkus, and Shimon Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6(6):861–867, 1993.

[11] Allan Pinkus. Approximation theory of the mlp model in neural networks. *Acta Numerica*, 8:143–195, 1999.

[12] Anastasis Kratsios. The universal approximation property. *Annals of Mathematics and Artificial Intelligence*, 89(5-6):435–469, jan 2021.

[13] Dirk Neumann. Ridge regression and lasso. *Albert Ludwigs University of Freiburg - Business Analytics summer seminar*, 2014.

[14] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[15] David Foster. *Generative Deep Learning*. O'Reilly Media, Inc., 2019.

[16] Egor Zakharov, Aliaksandra Shysheya, Egor Burkov, and Victor Lempitsky. Few-shot adversarial learning of realistic neural talking head models, 2019.

[17] Evgeny Zamyatin and Andrey Filchenkov. Learning to generate chairs with generative adversarial nets, 2017.

[18] Diana Davletshina, Valentyn Melnychuk, Viet Tran, Hitansh Singla, Max Berrendorf, Evgeniy Faerman, Michael Fromm, and Matthias Schubert. Unsupervised anomaly detection for x-ray images, 2020.

[19] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: with Applications in R*. Springer, 2013.

[20] Lilian Weng. From gan to wgan. *ArXiv*, abs/1904.08994, 2019.

[21] Martin Arjovsky and Léon Bottou. Towards principled methods for training generative adversarial networks. *arXiv preprint arXiv:1701.04862*, 2017.

[22] Jason Brownlee. *Generative Adversarial Networks with Python Deep Learning Generative Models for Image Synthesis and Image Translation*. 2019.

[23] Xiaoteng Ma Li Xia Jung Yang Zhiheng Li Xiu Li Ming Zhang, Yawei Wang. Wasserstein distance guided adversarial imitation learning with reward shape exploration. *The Department of Automation, Tshinghua University, Shenzhen 518055, P.R, China*, 2020.

[24] K. Smith and A. Smith. Conditional gan for timeseries generation. *ArXiv*, abs/2006.16477, 2020.

[25] Minhyeok Lee and Junhee Seok. Controllable generative adversarial network. *IEEE Access*, 7:28158–28169, 2019.

[26] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

6. Apéndice

6.1. Descenso por gradiente

La mayoría de los algoritmos de aprendizaje profundo implican la optimización de algún tipo. Optimización se refiere a la tarea de encontrar el mínimo o máximo global de una función $f(x)$, si existen. Lo que implica hallar los valores de x que cumplen dicha condición. Por lo general, la mayoría de los problemas de optimización se expresan en términos de minimizar $f(x)$. Para resolver un problema de maximización, se puede aplicar un algoritmo de minimización sobre -$f(x)$ [8].

La función que se desea minimizar o maximizar se le denomina función objetivo [8]. El valor óptimo se denota como $x^* = argmin(f(x))$.

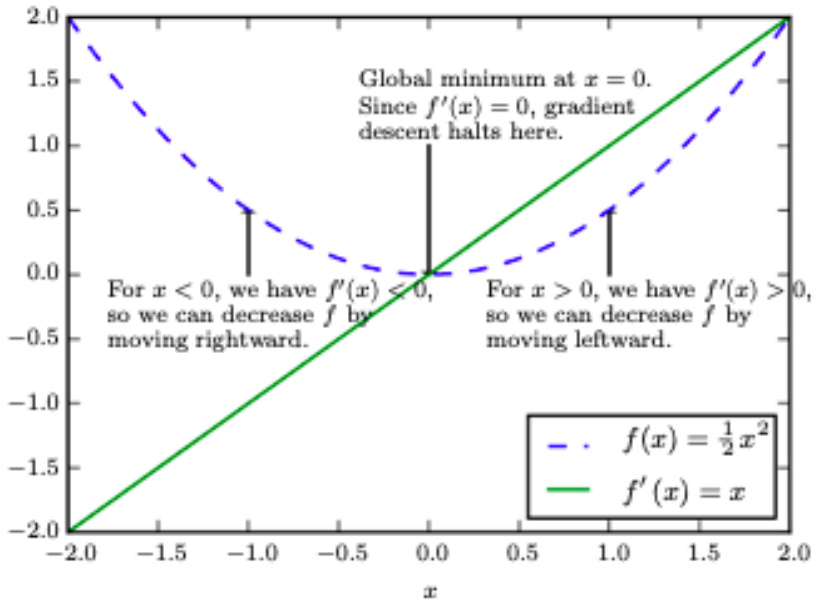


Figura 76: Descenso de gradiente. Una ilustración de cómo el algoritmo de descenso de gradiente utiliza las derivadas de una función para seguir la función cuesta abajo hasta un mínimo [5]

A partir de la figura 76 supóngase que se tiene una función $y = f(x)$ donde x e y son reales. Se denota como $f'(x)$ a la derivada de $f(x)$. La derivada brinda la pendiente en el punto x; indicando cómo afecta un pequeño cambio en x sobre la función $f(x)$. Es decir que se cumple que $f(x + \epsilon) \approx f'(x)\epsilon + f(x)$.

Esto lo se puede utilizar para encontrar un mínimo local de la función $f(x)$ porque dice cómo hacer una pequeña mejora en $y = f(x)$ a partir de una modificación en x. Sabemos que $f(x - \epsilon * sign(f'(x)))$ es menor que $f(x)$ para un ϵ lo suficientemente pequeño. Por lo tanto, es posible moverse en pequeños pasos en sentido opuesto al signo de $f'(x)$ y encontrar nuevos valores de x que minimicen de la función $f(x)$. Esta técnica se denomina **Descenso por Gradiente** [8]. Cuando la derivada es cero, no tenemos información sobre la dirección

hacia dónde moverse y el algoritmo queda estancado. En ese punto se dice que está en un punto crítico. Estos puntos pueden ser mínimos o máximos locales o bien pueden ser puntos silla. A continuación se presentan ejemplos.

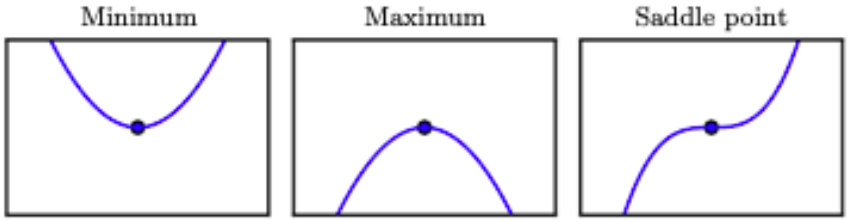


Figura 77: Ejemplo de puntos críticos - Mínimo, máximo y punto silla [5]

El tamaño del paso ϵ que se utiliza para moverse se denomina tasa de aprendizaje o "Learning Rate" en inglés [5]. Este paso puede ser fijo o variable. Existen distintas variantes sobre cómo tratar y variar el "Learning Rate". Si el paso es muy grande puede que no se llegue al mínimo y el algoritmo comience a "rebotar" indefinidamente. Si el paso es demasiado pequeño se va a demorar mucho en llegar un mínimo (o máximo). Si la función no es convexa, el punto critico alcanzado por el algoritmo puede no ser el mínimo global de la función. Para el caso de funciones de varias variables el planteo es análogo. El gradiente estará compuesto por el vector formado por las derivadas parciales de la función como se muestra a continuación.

$$x_{t+1} \; = \; x_t \; - \; \nabla_x f(x_t) \;\; \eta \; > \; 0 \text{ (step size/ learning rate)}$$

El algoritmo de descenso por gradiente finaliza cuando se llega a un punto estacionario. Es decir que $\Delta * f(x^*) = 0$. El gradiente total en base al conjunto de datos de entrada será el promedio de la la suma de todos los gradientes calculados para cada elemento del conjunto como se muestra a continuación:

$$\nabla_W L(W) = \frac{1}{n} \sum_{i=1}^{n} \nabla_W L(x_i; y_i; W) + \lambda \nabla_W R(W)$$

Luego es preciso moverse en sentido opuesto a esta dirección resultante y se vuelve a calcular hasta que el algoritmo converge a un punto estacionario.

Hay dos formas de calcular el gradiente:

- De manera analítica: esto es calcular explícitamente la expresión de la derivada de la función.
- Mediante aproximación numérica: se hace la definición del gradiente. Se toma un incremento de la función, se le resta el valor de la función y luego se divide sobre el incremento, obteniendo un valor aproximado de la derivada en ese punto.

Según [5], en la práctica cuando se trabaja con redes neuronales se calculará de manera analítica. La idea es que al conocer todas las operaciones aplicadas, es posible entonces calcular de forma automática y analítica el gradiente. Gran parte de las librerías que manejan redes neuronales se dedican a implementar eficientemente esta operación de calcular el gradiente de las redes neuronales. Se suele utilizar la aproximación numérica como una forma de "Sanity Check" a efectos de verificar que el resultado analítico del gradiente es correcto [5].

6.2. Optimizador Adam

El algoritmo de optimización Adam es una extensión del descenso de gradiente estocástico que puede ser utilizado en lugar del procedimiento clásico para actualizar de forma iterativa los pesos de la red neuronal basándose en los datos de entrenamiento. Este fue presentado por Diederik Kingma y Jimmy Ba en su articulo [26].

A diferencia del descenso de gradiente estocástico, que mantiene una tasa de aprendizaje única (llamada α) para todas las actualizaciones de peso y que no cambia durante el entrenamiento, Adam calcula tasas de aprendizaje adaptativas individuales para diferentes parámetros a partir de estimaciones de los primeros y segundos momentos de los gradientes. Los autores describen a Adam como la combinación de dos extensiones del descenso de gradiente estocástico: "Algoritmo de Gradiente Adaptativo" (AdaGrad) y la "Propagación de la Raíz Media Cuadrática" (RMSProp)[26].

En lugar de adaptar las tasas de aprendizaje de los parámetros basadas en el promedio del primer momento (la media) como en RMSProp, Adam también utiliza el promedio de los segundos momentos de los gradientes (la varianza no centrada). Específicamente, el algoritmo calcula un promedio móvil exponencial del gradiente y del gradiente al cuadrado. Los parámetros β_1 y β_2 controlan las tasas de decaimiento de estos promedios móviles. El valor inicial de los promedios móviles y los valores de β_1 y β_2 cercanos a 1.0 (recomendados), resultan en un sesgo de las estimaciones del momento hacia cero. Este sesgo se supera al calcular primero las estimaciones sesgadas antes de calcular las estimaciones corregidas[26].

Parámetros de configuración de Adam:

1. α. También conocida como tasa de aprendizaje o tamaño del paso. La proporción con la que se actualizan los pesos (por ejemplo, 0.001). Valores más grandes (por ejemplo, 0.3) resultan en un aprendizaje inicial más rápido mientras que valores más pequeños (por ejemplo, 10^{-5}) ralentizan el aprendizaje durante el entrenamiento.
2. β_1. La tasa de decaimiento exponencial para las estimaciones del primer momento (por ejemplo, 0.9).

3. β_2 La tasa de decaimiento exponencial para las estimaciones del segundo momento (por ejemplo, 0.999). Este valor debe establecerse cerca de 1.0 en problemas con un gradiente disperso.

4. ϵ. Es un número muy pequeño para evitar cualquier división por cero en la implementación (por ejemplo, 10^{-8}).

Dentro de los principales beneficios se detaca:

1. Fácil de implementar.

2. Eficiente computacionalmente.

3. Requisitos de memoria reducidos.

4. Invariante a la reescala diagonal de los gradientes.

5. Bien adaptado para problemas que son grandes en términos de datos y/o parámetros.

6. Apropiado para objetivos no estacionarios.

7. Apropiado para problemas con gradientes muy ruidosos o dispersos.

8. Los hiperparámetros tienen una interpretación intuitiva y típicamente requieren poco ajuste.

A continuación se presenta el algoritmo en pseudocódigo.

Algorithm 8.7 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \boldsymbol{0}$, $\boldsymbol{r} = \boldsymbol{0}$
  Initialize time step $t = 0$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
    Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
    Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
    Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
    Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while

Figura 78: pseudocódigo de optimizador de Adam [8, cap 8]