Deep Learning

Jorge Alcalde Vesteiro

1. Definición y Conceptos Clave

El Deep Learning (aprendizaje profundo) es una rama del aprendizaje automático (machine learning), que a su vez pertenece a la inteligencia artificial (IA). La IA busca crear sistemas capaces de realizar tareas que requieren inteligencia humana. El aprendizaje automático permite a las máquinas aprender de datos sin programación explícita. El Deep Learning utiliza redes neuronales artificiales profundas (con múltiples capas) para extraer automáticamente patrones complejos de los datos.

Definición 1.1 (Red Neuronal Artificial (RNA))**.** *Modelo computacional inspirado en el cerebro biológico. Compuesto por nodos (neuronas) interconectados en capas. Las conexiones tienen pesos ajustables.*

Definición 1.2 (Red Neuronal Profunda (RNP))**.** *RNA con múltiples capas ocultas entre la entrada y la salida. Permite aprender representaciones jerárquicas y modelar relaciones complejas.*

El Deep Learning aprende representaciones de los datos. La red extrae características relevantes de los datos brutos a través de sus capas, aprendiendo características más abstractas en cada nivel.

2. Historia DL

Algunos hitos:

- **1958: Perceptrón.** Primer modelo de red neuronal (clasificación lineal).
- **1986: Retropropagación (Backpropagation).** Algoritmo para entrenar redes multicapa.
- **1989: Redes Neuronales Convolucionales (CNNs).** Procesamiento de imágenes (patrones locales).
- **1997: Redes LSTM (Long Short-Term Memory).** Procesamiento de secuencias (memoria a largo plazo).
- **2012: AlexNet.** CNN profunda que marcó el inicio del auge actual.
- **2014: Redes Generativas Antagónicas (GANs).** Generación de datos sintéticos.
- **2017: Transformers.** Arquitectura para procesamiento de lenguaje (mecanismo de atención).
- **2020+: Modelos Fundacionales.** Modelos grandes pre-entrenados (GPT, BERT).

3. Factores del Auge Actual

1. **Big Data:** Gran disponibilidad de datos para entrenamiento.
2. **Hardware:**
   - **GPUs:** Eficientes para cálculos matriciales paralelos.
   - **TPUs:** Chips diseñados para Deep Learning.
3. **Software:** Frameworks de código abierto (PyTorch, TensorFlow) que simplifican el desarrollo.

4. Aplicaciones Principales

- **Procesamiento del Lenguaje Natural (PLN):** Traducción, análisis de sentimiento, reconocimiento de voz, generación de texto, chatbots.
- **Visión por Computador:** Clasificación/detección/segmentación de imágenes, reconocimiento facial, generación de imágenes.
- **Audio:** Reconocimiento de voz, clasificación de sonidos, generación de música.
- **Otras áreas:** Robótica, análisis de datos (biología, medicina, finanzas), predicción meteorológica, juegos, conducción autónoma.

5. Desafíos y Ética

5.1. Desafíos Técnicos

- **Necesidad de datos:** Requiere grandes conjuntos de datos etiquetados (la aumentación de datos ayuda).
- **Costo computacional:** Entrenar modelos grandes es costoso en hardware y energía.
- **Explicabilidad:** Dificultad para entender las decisiones de la red (investigación en IA explicable - XAI).

5.2. Consideraciones Éticas

- **Sesgos:** Los modelos pueden perpetuar sesgos presentes en los datos.
- **Privacidad:** Riesgos asociados al análisis de datos personales.
- **Impacto en el empleo:** Potencial automatización de trabajos.
- **Deepfakes:** Generación de contenido falso realista.

6. El Perceptrón: La Unidad Básica de las Redes Neuronales

El perceptrón, introducido por Frank Rosenblatt en 1957, es el componente fundamental de las redes neuronales y, por lo tanto, del deep learning. Es un modelo inspirado en la biología, específicamente en el funcionamiento de las neuronas.

6.1. Estructura del Perceptrón

Un perceptrón recibe un conjunto de entradas, realiza una combinación lineal de estas entradas y luego aplica una función de activación para producir una salida. Visualmente, se puede representar como un nodo con:

- **Entradas** $(x_1, x_2, ..., x_n)$: Valores numéricos que representan la información de entrada. Pueden ser características de un objeto, píxeles de una imagen, palabras de una frase, etc.
- **Pesos** $(w_1, w_2, ..., w_n)$: Valores numéricos asociados a cada entrada. Representan la importancia de cada entrada en la decisión final del perceptrón. Un peso mayor indica una mayor influencia.
- **Bias** (w_0): Un peso adicional que no está asociado a ninguna entrada específica. El bias permite ajustar el umbral de activación del perceptrón, lo que le da más flexibilidad. Se puede considerar como el peso de una entrada ficticia siempre igual a 1.
- **Función de Suma Ponderada**: Calcula la suma ponderada de las entradas, los pesos y el bias:

$$\text{suma} = w_0 + w_1x_1 + w_2x_2 + ... + w_nx_n = w_0 + \sum_{i=1}^{n} w_ix_i$$

- **Función de Activación** (g): Una función que toma la suma ponderada como entrada y produce la salida del perceptrón. Introduce no linealidad en el modelo, lo que permite a la red neuronal aprender relaciones más complejas que simplemente lineales. Ejemplos comunes:
  - **Función escalón (o Heaviside):** Produce una salida binaria (0 o 1). Si la suma ponderada es mayor o igual a un umbral, la salida es 1; de lo contrario, es 0.
  - **Función sigmoide:** Produce una salida entre 0 y 1, que puede interpretarse como una probabilidad.
  - **Función ReLU (Rectified Linear Unit):** Si la suma ponderada es positiva, la salida es igual a la suma; si es negativa, la salida es 0. Es muy popular en redes neuronales profundas.

6.2. Salida del Perceptrón

La salida ($\hat{y}$) del perceptrón se calcula de la siguiente manera:

$$\hat{y} = g\left(w_0 + \sum_{i=1}^{n} w_i x_i\right)$$

Donde:

- $\hat{y}$ es la salida del perceptrón.
- g es la función de activación.
- w_0 es el bias.
- w_i es el peso asociado a la entrada x_i.
- x_i es el valor de la entrada i.

6.3. Forma Matricial

La ecuación del perceptrón se puede expresar de forma más compacta utilizando notación matricial:

$$\hat{y} = g(\mathbf{w}^T \mathbf{x} + w_0)$$

O, más comúnmente, incluyendo el bias dentro del vector de pesos:

$$\hat{y} = g(\mathbf{w}^T \mathbf{x})$$

Donde:

- $\mathbf{w} = [w_0, w_1, w_2, ..., w_n]^T$ es el vector de pesos (incluyendo el bias como w_0).
- $\mathbf{x} = [1, x_1, x_2, ..., x_n]^T$ es el vector de entradas (incluyendo un 1 para el bias).
- ^{T} denota la transpuesta del vector.

Esta forma matricial es más eficiente para la implementación en computadoras, ya que se pueden utilizar bibliotecas optimizadas para álgebra lineal.

6.4. Limitaciones del Perceptrón Simple

El perceptrón simple, con una función de activación escalón, solo puede aprender a clasificar datos que son *linealmente separables*. Esto significa que los datos se pueden separar en dos clases mediante una línea recta (en 2D), un plano (en 3D) o un hiperplano (en dimensiones superiores). No puede resolver problemas como la función XOR (o-exclusivo), que no es linealmente separable. Esta limitación se supera con las redes neuronales multicapa.

7. Funciones de Activación

La función de activación, denotada como g, es un componente crucial del perceptrón y, en general, de cualquier red neuronal. Determina la salida de una neurona dada una entrada o conjunto de entradas. No es simplemente un detalle técnico; sin ella, las redes neuronales serían extremadamente limitadas.

7.1. ¿Por Qué Necesitamos Funciones de Activación?

La razón principal es introducir no linealidad. Si no usáramos funciones de activación (o si usáramos solo funciones lineales), la salida de una red neuronal siempre sería una combinación lineal de las entradas, sin importar cuántas capas tenga la red. Esto limitaría drásticamente la capacidad de la red para aprender patrones complejos.

Imagina que tienes un perceptrón sin función de activación. Su salida sería simplemente:

$$\hat{y} = \mathbf{w}^T \mathbf{x}$$

Si conectas varios perceptrones de este tipo en capas, la salida de la última capa seguiría siendo una combinación lineal de las entradas originales. Podrías 'colapsar' toda la red en un único perceptrón equivalente. No ganarías nada con tener múltiples capas.

Las funciones de activación no lineales permiten a las redes neuronales aproximar cualquier función continua (esto se conoce como el teorema de aproximación universal). En otras palabras, con suficientes neuronas y la función de activación correcta, una red neuronal puede, en teoría, aprender cualquier relación entre entradas y salidas, por muy compleja que sea.

7.2. Características Clave de las Funciones de Activación

- **No linealidad:** Como ya se ha mencionado, esta es la propiedad más importante.
- **Diferenciabilidad:** La función de activación debe ser diferenciable (al menos en la mayor parte de su dominio). Esto es esencial para el algoritmo de retropropagación (backpropagation), que se basa en el cálculo de gradientes.
- **Rango:** El rango de valores de salida de la función de activación puede influir en el comportamiento del entrenamiento. Algunas funciones tienen un rango limitado (como la sigmoide, entre 0 y 1), mientras que otras tienen un rango ilimitado (como ReLU).
- **Complejidad computacional:** Algunas funciones de activación son más rápidas de calcular que otras. Esto puede ser importante en aplicaciones donde la velocidad es crítica.
- **Problema de la 'neurona muerta':** Algunas funciones, como ReLU, pueden sufrir el problema de la neurona muerta. Si la entrada a una neurona ReLU es siempre negativa, la salida será siempre 0, y el gradiente también será 0. Esto significa que los pesos de esa neurona no se actualizarán durante el entrenamiento, y la neurona quedará m̈uerta.

7.3. Funciones de Activación Comunes

- **Sigmoide:**

  $$\sigma(z) = \frac{1}{1 + e^{-z}}$$

  Produce una salida entre 0 y 1. Se usaba mucho en el pasado, pero ahora se usa menos en capas ocultas debido al problema del *desvanecimiento del gradiente* (vanishing gradient). Cuando la entrada es muy grande (positiva o negativa), la salida de la sigmoide se satura (se acerca a 1 o a 0), y el gradiente se vuelve muy pequeño. Esto dificulta el entrenamiento. Sin embargo es muy útil en la capa de salida cuando se necesita una probabilidad (clasificación binaria).

- **Tangente Hiperbólica (tanh):**

  $$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Similar a la sigmoide, pero su rango es entre -1 y 1. También sufre del problema del desvanecimiento del gradiente, pero al estar centrada en 0, suele converger más rápido que la sigmoide.

- **ReLU (Rectified Linear Unit):**

$$\text{ReLU}(z) = \text{máx}(0, z)$$

Simplemente devuelve la entrada si es positiva, y 0 si es negativa. Es muy popular en la actualidad porque es computacionalmente eficiente y ayuda a mitigar el problema del desvanecimiento del gradiente (aunque puede sufrir el problema de la neurona muerta).

- **Leaky ReLU:**

$$\text{Leaky ReLU}(z) = \begin{cases} z & \text{si } z > 0 \\ \alpha z & \text{si } z \leq 0 \end{cases}$$

Donde α es un valor pequeño (por ejemplo 0.01) Una variante de ReLU que intenta solucionar el problema de la neurona muerta. En lugar de devolver 0 para entradas negativas, devuelve un pequeño valor multiplicado por la entrada.

- **ELU (Exponential Linear Unit):** Otra variante que tiene la siguiente forma:

$$\text{ELU}(z) = \begin{cases} z & \text{si } z > 0 \\ \alpha(e^z - 1) & \text{si } z \leq 0 \end{cases}$$

- **Softmax:**

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Se utiliza en la capa de salida para problemas de clasificación multiclase. Toma un vector de valores reales y lo transforma en una distribución de probabilidad, donde cada elemento representa la probabilidad de que la entrada pertenezca a una clase específica. La suma de todos los elementos del vector de salida es 1. No es adecuada para clasificación binaria; en ese caso, se usa la sigmoide.

7.4. Uso en Capas de Salida vs. Capas Ocultas

La elección de la función de activación depende de la tarea y de la capa de la red:

- **Capas ocultas: ReLU y sus variantes (Leaky ReLU, ELU) son las opciones más comunes en la actualidad por su eficiencia y buen rendimiento.**
- **Capa de salida:**
  - **Clasificación binaria:** Sigmoide (produce una probabilidad entre 0 y 1).
  - **Clasificación multiclase:** Softmax (produce una distribución de probabilidad sobre las clases).
  - **Regresión:** Generalmente no se usa función de activación (o se usa una función lineal), ya que se quiere predecir un valor continuo sin restricciones.

8. Redes Neuronales Totalmente Conectadas (Fully Connected Networks)

Un solo perceptrón tiene una capacidad limitada para aprender patrones complejos. La verdadera potencia de las redes neuronales surge al conectar múltiples perceptrones entre sí. La forma más común de hacerlo es mediante capas totalmente conectadas.

8.1. Capas Totalmente Conectadas

En una capa totalmente conectada (también llamada capa densa), cada neurona de una capa está conectada a todas las neuronas de la capa anterior. Esto significa que cada neurona recibe como entrada las salidas de todas las neuronas de la capa precedente.

8.2. Construcción de una Red Totalmente Conectada

Una red neuronal totalmente conectada se construye apilando varias capas totalmente conectadas. Una red típica tiene:

- **Capa de entrada:** Recibe los datos de entrada (las características o "features"). No realiza ningún cálculo, simplemente distribuye las entradas a la siguiente capa.
- **Capas ocultas:** Una o más capas totalmente conectadas que realizan el procesamiento principal. Cada neurona en una capa oculta recibe las salidas de todas las neuronas de la capa anterior, realiza una suma ponderada (incluyendo el bias), y aplica una función de activación.
- **Capa de salida:** Produce la salida final de la red. El número de neuronas en la capa de salida depende de la tarea (por ejemplo, una neurona para regresión o clasificación binaria, múltiples neuronas para clasificación multiclase). La función de activación en la capa de salida también depende de la tarea (por ejemplo, sigmoide para clasificación binaria, softmax para clasificación multiclase).

8.3. Flujo de Información (Forward Pass)

El cálculo de la salida de la red, dado un conjunto de entradas, se llama 'forward pass' o propagación hacia adelante. La información fluye desde la capa de entrada, a través de las capas ocultas, hasta la capa de salida.

Para una red con una capa oculta, el cálculo se realiza de la siguiente manera (usando notación matricial):

1. **Capa oculta:**
$$\mathbf{h} = g(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)})$$
Donde:
   - $\mathbf{x}$ es el vector de entradas.
   - $\mathbf{W}^{(1)}$ es la matriz de pesos de la capa oculta (cada fila corresponde a una neurona de la capa oculta, y cada columna a una entrada).
   - $\mathbf{b}^{(1)}$ es el vector de bias de la capa oculta.
   - g es la función de activación (aplicada elemento a elemento al vector resultante).
   - $\mathbf{h}$ es el vector de salidas de la capa oculta.
2. **Capa de salida:**
$$\hat{\mathbf{y}} = g(\mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)})$$
Donde:
   - $\mathbf{h}$ es el vector de salidas de la capa oculta (que ahora sirve como entrada para la capa de salida).
   - $\mathbf{W}^{(2)}$ es la matriz de pesos de la capa de salida.
   - $\mathbf{b}^{(2)}$ es el vector de bias de la capa de salida.
   - g es la función de activación de la capa de salida (puede ser diferente de la función de activación de la capa oculta).
   - $\hat{\mathbf{y}}$ es el vector de salidas de la red.

8.4. Generalización a Múltiples Capas

Para una red con k capas ocultas, la propagación hacia adelante se puede generalizar como:

$$\mathbf{h}^{(l)} = g(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)})$$

Donde:

- l es el índice de la capa ($l = 1$ para la primera capa oculta, $l = k$ para la última capa oculta).
- $\mathbf{h}^{(l)}$ es el vector de salidas de la capa l.
- $\mathbf{W}^{(l)}$ es la matriz de pesos de la capa l.
- $\mathbf{b}^{(l)}$ es el vector de bias de la capa l.
- $\mathbf{h}^{(0)} = \mathbf{x}$ (la capa de entrada).
- g es la función de activación de cada capa.

Y la salida final es:

$$\hat{\mathbf{y}} = g(\mathbf{W}^{(k+1)}\mathbf{h}^{(k)} + \mathbf{b}^{(k+1)})$$

8.5. Multiplicación de Matrices

Como se puede ver en las ecuaciones anteriores, la operación fundamental en una red totalmente conectada es la multiplicación de matrices. Cada capa implica multiplicar una matriz de pesos por un vector de entradas (o salidas de la capa anterior), y sumar un vector de bias. Esta es la razón por la que las GPUs son tan eficientes para entrenar redes neuronales: están optimizadas para realizar multiplicaciones de matrices en paralelo.

8.6. Simplificación de la Notación (Opcional)

A menudo, para simplificar la notación, se incluye el bias dentro de la matriz de pesos. En este caso se debe incluir un '1' al principio del vector de activaciones de la capa anterior.

8.7. Número de Parámetros

El número de parámetros (pesos y bias) en una red totalmente conectada crece rápidamente con el número de capas y el número de neuronas por capa. Por ejemplo, si una capa tiene n entradas y m neuronas, la matriz de pesos tendrá $m \times n$ elementos, y el vector de bias tendrá m elementos. Un número elevado de parámetros puede llevar al sobreajuste (overfitting), donde el modelo se ajusta demasiado a los datos de entrenamiento y generaliza mal a nuevos datos.

9. Función de Pérdida (Loss Function): Midiendo el Error

Hasta ahora, hemos visto cómo una red neuronal totalmente conectada procesa las entradas para producir una salida. Pero, ¿cómo aprende la red a realizar predicciones precisas? La clave está en la función de pérdida (también llamada función de coste).

9.1. ¿Qué es la Función de Pérdida?

La función de pérdida es una función que cuantifica la diferencia entre la salida predicha por la red neuronal ($\hat{\mathbf{y}}$) y el valor real (o valor objetivo, $\mathbf{y}$). En otras palabras, mide qué tan 'mal' está funcionando la red en un determinado ejemplo o conjunto de ejemplos.

El objetivo del entrenamiento de una red neuronal es encontrar los pesos ($\mathbf{W}$) y los bias ($\mathbf{b}$) que minimicen la función de pérdida. Cuanto menor sea el valor de la función de pérdida, mejor será el rendimiento de la red.

9.2. Características de una Buena Función de Pérdida

- **Representativa del objetivo:** La función de pérdida debe reflejar el objetivo real del problema. Una mala elección de la función de pérdida puede llevar a un modelo que, aunque tenga una pérdida baja, no sea útil en la práctica.
- **Diferenciable:** La función de pérdida debe ser diferenciable (al menos en la mayor parte de su dominio). Esto es crucial para el algoritmo de retropropagación, que se basa en el cálculo de gradientes para ajustar los pesos de la red.
- **Convexa (idealmente):** Una función convexa tiene un único mínimo global, lo que facilita la optimización. Aunque no todas las funciones de pérdida utilizadas en deep learning son convexas (de hecho, la función de pérdida de una red neuronal profunda típicamente no es convexa), es una propiedad deseable.

9.3. Un Ejemplo Práctico: La Puerta XOR

La función XOR, no es linealmente separable. Para este ejemplo:

- Se define una red con: 2 neuronas de entrada, 2 ocultas y 1 de salida.
- Función de activación: Sigmoide (en todas las capas, aunque se podría usar otra en las ocultas como relu)
- Pesos aleatorios (sin bias, para simplificar).
- Se calcula un forward pass.

Con pesos aleatorios, la red comete errores. Es necesario utilizar una función de pérdida para cuantificar el error.

9.4. Tipos de Funciones de Pérdida

Las funciones de pérdida se pueden clasificar según el tipo de tarea de aprendizaje automático:

9.4.1. Clasificación

- **Binary Cross-Entropy (Entropía Cruzada Binaria):** Se utiliza para problemas de clasificación binaria (dos clases, 0 o 1). Mide la diferencia entre la probabilidad predicha por la red (un valor entre 0 y 1, típicamente obtenido aplicando una función sigmoide a la salida de la última capa) y la etiqueta real (0 o 1).

$$\text{BCELoss}(\hat{y}, y) = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

  Donde:

  - N es el número de ejemplos.
  - y_i es la etiqueta real del ejemplo i (0 o 1).
  - $\hat{y}_i$ es la probabilidad predicha por la red para el ejemplo i.

  Características:

  - Penaliza fuertemente las predicciones incorrectas con alta confianza.
  - Es diferenciable.
  - Proporciona una interpretación probabilística.
  - Trata ambas clases simétricamente.

- **Categorical Cross-Entropy (Entropía Cruzada Categórica):** Se utiliza para problemas de clasificación multiclase (más de dos clases). Es una generalización de la entropía cruzada binaria. La salida de la red es típicamente un vector de probabilidades (obtenido aplicando una función softmax a la salida de la última capa), y la etiqueta real se representa como un vector .ºne-hot"(un vector con un 1 en la posición correspondiente a la clase correcta y 0 en las demás posiciones).

$$\text{CCELoss}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

Donde:

- C es el número de clases.
- $\mathbf{y}$ es el vector 'one-hot' que representa la etiqueta real.
- $\hat{\mathbf{y}}$ es el vector de probabilidades predichas por la red.

9.4.2. Regresión

- **Mean Absolute Error (MAE, Error Absoluto Medio):** Calcula el promedio de las diferencias absolutas entre las predicciones y los valores reales.

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i|$$

Es fácil de interpretar, ya que el error está en la misma unidad que la variable objetivo.

- **Mean Squared Error (MSE, Error Cuadrático Medio):** Calcula el promedio de los cuadrados de las diferencias entre las predicciones y los valores reales.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Penaliza más los errores grandes que el MAE, debido al término cuadrático.

- **Mean Absolute Percentage Error (MAPE, Error Porcentual Absoluto Medio):** Calcula el promedio de los errores porcentuales absolutos.

$$\text{MAPE} = \frac{1}{n}\sum_{i=1}^{n} \left| \frac{y_i - \hat{y}_i}{y_i} \right| \times 100\,\%$$

Proporciona una medida relativa del error, expresada como un porcentaje. *No se puede usar si y_i puede ser cero.*

9.5. Interpretación de la Función de Pérdida

- Valores cercanos a cero: Indican que el modelo está haciendo buenas predicciones.
- Valores altos: Indican que el modelo está cometiendo errores significativos.
- Valores mayores que 1 en Cross-Entropy: Normalmente son un error, revisa el código (las probabilidades deben estar mal calculadas).

Importante: La función de pérdida se calcula generalmente sobre un lote (batch) de ejemplos, no sobre un solo ejemplo. El valor de la función de pérdida para el lote es el promedio de los valores para cada ejemplo del lote.

La elección de la función de pérdida adecuada es una decisión de diseño crucial en el desarrollo de modelos de aprendizaje automático.

10. Descenso de Gradiente y Retropropagación (Backpropagation)

Hemos visto cómo evaluar una red neuronal (forward pass) y cómo medir su error (función de pérdida). Ahora, la pregunta clave es: ¿cómo entrenamos la red para que aprenda a realizar predicciones precisas? La respuesta involucra dos conceptos fundamentales: el descenso de gradiente y la retropropagación.

10.1. Entrenamiento de una Red Neuronal: El Objetivo

El objetivo del entrenamiento es encontrar los valores óptimos de los pesos ($\mathbf{W}$) y los bias ($\mathbf{b}$) de la red que minimicen la función de pérdida ($\mathcal{L}$) sobre el conjunto de entrenamiento. Formalmente, buscamos:

$$\mathbf{W}^*, \mathbf{b}^* = \arg\min_{\mathbf{W},\mathbf{b}} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(y_i, f(\mathbf{x}_i; \mathbf{W}, \mathbf{b}))$$

Donde:

- N es el número de ejemplos en el conjunto de entrenamiento.
- $\mathbf{x}_i$ es el vector de entrada del ejemplo i.
- y_i es la etiqueta (valor objetivo) del ejemplo i.
- $f(\mathbf{x}_i; \mathbf{W}, \mathbf{b})$ es la salida de la red para la entrada $\mathbf{x}_i$, dados los pesos $\mathbf{W}$ y los bias $\mathbf{b}$.
- $\mathcal{L}$ es la función de pérdida.
- $\mathbf{W}^*$ y $\mathbf{b}^*$ son los valores óptimos de los pesos y los bias, respectivamente.

Este es un problema de optimización complejo, y en general, no se puede resolver de forma analítica (es decir, no hay una fórmula cerrada para encontrar $\mathbf{W}^*$ y $\mathbf{b}^*$). En su lugar, se utiliza un algoritmo iterativo llamado descenso de gradiente.

10.2. Descenso de Gradiente (Gradient Descent)

El descenso de gradiente es un algoritmo de optimización que busca el mínimo de una función de forma iterativa. La idea es 'descender' por la superficie de la función de pérdida en la dirección de máxima pendiente negativa (el gradiente negativo).

10.2.1. El Gradiente

El gradiente de una función, denotado como ∇f, es un vector que apunta en la dirección de máximo crecimiento de la función en un punto dado. Cada componente del gradiente es la derivada parcial de la función con respecto a una de sus variables. Para una función de pérdida $\mathcal{L}(\mathbf{W}, \mathbf{b})$, el gradiente con respecto a los pesos sería:

$$\nabla_{\mathbf{W}} \mathcal{L} = \left[\frac{\partial \mathcal{L}}{\partial w_1}, \frac{\partial \mathcal{L}}{\partial w_2}, \ldots, \frac{\partial \mathcal{L}}{\partial w_n} \right]$$

Donde n es el número total de pesos en la red. Análogamente, se calcula el gradiente con respecto a los bias.

10.2.2. El Algoritmo

El algoritmo de descenso de gradiente sigue estos pasos:

1. **Inicialización:** Se inicializan los pesos ($\mathbf{W}$) y los bias ($\mathbf{b}$) de la red con valores aleatorios (o siguiendo alguna otra estrategia de inicialización).
2. **Cálculo del gradiente:** Se calcula el gradiente de la función de pérdida con respecto a los pesos y los bias, utilizando los datos de entrenamiento.

3. **Actualización de los parámetros:** Se actualizan los pesos y los bias en la dirección opuesta al gradiente, utilizando una tasa de aprendizaje (η, "learning rate"):

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \nabla_{\mathbf{W}} \mathcal{L}$$

$$\mathbf{b} \leftarrow \mathbf{b} - \eta \nabla_{\mathbf{b}} \mathcal{L}$$

4. **Repetición:** Se repiten los pasos 2 y 3 hasta que se cumpla algún criterio de parada (por ejemplo, hasta que la función de pérdida deje de disminuir significativamente, o hasta que se alcance un número máximo de iteraciones).

10.2.3. Tasa de Aprendizaje (Learning Rate)

La tasa de aprendizaje (η) es un hiperparámetro crucial que controla el tamaño de los pasos que da el algoritmo en cada iteración.

- η **muy pequeña:** El algoritmo convergerá muy lentamente, y puede quedar atrapado en mínimos locales.
- η **muy grande:** El algoritmo puede oscilar alrededor del mínimo, o incluso divergir (no converger).

La elección de la tasa de aprendizaje adecuada es un desafío, y a menudo requiere experimentación.

10.2.4. Inicialización de Pesos

La inicialización de los pesos también es importante. Una mala inicialización puede llevar a que el algoritmo quede atrapado en un mínimo local o que tarde mucho en converger. Una estrategia común es inicializar los pesos aleatoriamente siguiendo una distribución normal con media 0 y una desviación estándar pequeña.

10.3. Retropropagación (Backpropagation)

El descenso de gradiente requiere calcular el gradiente de la función de pérdida con respecto a todos los pesos y bias de la red. En una red neuronal profunda, esto puede involucrar miles o incluso millones de parámetros. Calcular este gradiente directamente sería extremadamente ineficiente.

Aquí es donde entra en juego la retropropagación (backpropagation). Es un algoritmo eficiente para calcular el gradiente de la función de pérdida en una red neuronal, utilizando la regla de la cadena de cálculo.

10.3.1. La Regla de la Cadena

La regla de la cadena permite calcular la derivada de una función compuesta. Si tenemos una función $z = f(y)$ y $y = g(x)$, entonces la derivada de z con respecto a x es:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

10.3.2. Aplicación a la Red Neuronal

En una red neuronal, la función de pérdida es una función compuesta de los pesos, los bias y las funciones de activación de todas las capas. La retropropagación aplica la regla de la cadena repetidamente para calcular las derivadas parciales de la función de pérdida con respecto a cada peso y bias, propagando el error hacia atrás desde la capa de salida hasta la capa de entrada.

Supongamos que queremos calcular $\frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}}$, donde $w_{ij}^{(l)}$ es el peso que conecta la neurona j de la capa $l-1$ con la neurona i de la capa l. Usando la regla de la cadena:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} = \frac{\partial \mathcal{L}}{\partial a_i^{(l)}} \frac{\partial a_i^{(l)}}{\partial z_i^{(l)}} \frac{\partial z_i^{(l)}}{\partial w_{ij}^{(l)}}$$

Donde:

- $z_i^{(l)}$ es la suma ponderada de las entradas a la neurona i de la capa l (antes de aplicar la función de activación).
- $a_i^{(l)}$ es la activación (salida) de la neurona i de la capa l ($a_i^{(l)} = g(z_i^{(l)})$).

La retropropagación calcula estos términos de forma eficiente, reutilizando cálculos intermedios.

10.4. Tipos de Descenso de Gradiente

- **Descenso de Gradiente por Lotes (Batch Gradient Descent):** Calcula el gradiente utilizando todos los ejemplos del conjunto de entrenamiento en cada iteración. Es computacionalmente costoso, pero proporciona una estimación precisa del gradiente.
- **Descenso de Gradiente Estocástico (Stochastic Gradient Descent, SGD):** Calcula el gradiente utilizando un solo ejemplo (elegido aleatoriamente) en cada iteración. Es mucho más rápido que el descenso de gradiente por lotes, pero el gradiente estimado es más 'ruidoso'.
- **Descenso de Gradiente por Mini-Lotes (Mini-Batch Gradient Descent):** Un compromiso entre los dos anteriores. Calcula el gradiente utilizando un pequeño subconjunto (mini-lote) de ejemplos en cada iteración. Es el método más utilizado en la práctica, ya que combina la eficiencia del SGD con la estabilidad del descenso de gradiente por lotes.

11. El Problema del Desvanecimiento del Gradiente (Vanishing Gradient Problem)

Durante el entrenamiento de redes neuronales profundas, puede surgir un problema conocido como *desvanecimiento del gradiente*. Este fenómeno dificulta o impide el aprendizaje efectivo de las capas inferiores de la red.

11.1. ¿Qué es el Desvanecimiento del Gradiente?

El desvanecimiento del gradiente ocurre cuando los gradientes de la función de pérdida con respecto a los pesos de las capas anteriores se vuelven extremadamente pequeños a medida que se propagan hacia atrás a través de la red durante la retropropagación.

Recordemos que la actualización de los pesos en el descenso de gradiente se realiza de la siguiente manera:

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \nabla_{\mathbf{W}} \mathcal{L}$$

Si los componentes del gradiente $\nabla_{\mathbf{W}} \mathcal{L}$ son muy cercanos a cero, la actualización de los pesos será mínima o nula. Esto significa que las capas iniciales de la red aprenden muy lentamente o dejan de aprender por completo. La red se .estanca".

11.2. Causas del Desvanecimiento del Gradiente

Hay dos causas principales:

1. **Funciones de activación con gradientes pequeños en sus extremos:** Algunas funciones de activación, como la sigmoide y la tangente hiperbólica (tanh), tienen la propiedad de que sus derivadas se acercan a cero cuando la entrada es muy grande (en valor absoluto).

   Consideremos una red profunda que utiliza la función sigmoide como activación en todas sus capas. Durante la retropropagación, la regla de la cadena implica multiplicar las derivadas de las funciones de activación de cada capa. Si muchas de estas derivadas son cercanas a cero (porque las salidas de las neuronas están saturadas), el gradiente resultante se volverá exponencialmente pequeño a medida que se propaga hacia las capas iniciales.

   - **Sigmoide:** $\sigma(z) = \frac{1}{1+e^{-z}}$. Su derivada es $\sigma'(z) = \sigma(z)(1-\sigma(z))$. El valor máximo de la derivada es 0.25 (cuando $z = 0$), y se acerca a 0 rápidamente cuando $|z|$ es grande.

- **Tangente Hiperbólica (tanh):** $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$. Su derivada es $\tanh'(z) = 1 - \tanh^2(z)$. El valor máximo de la derivada es 1 (cuando $z = 0$), y se acerca a 0 rápidamente cuando $|z|$ es grande.

2. **Inicialización de pesos inadecuada:** Si los pesos iniciales de la red son demasiado pequeños, las activaciones de las neuronas también tenderán a ser pequeñas, y las derivadas de las funciones de activación (como la sigmoide) estarán en la región donde son cercanas a cero, exacerbando el problema.

11.3. Consecuencias

- **Convergencia lenta:** El entrenamiento se vuelve extremadamente lento, ya que las capas iniciales de la red apenas se actualizan.
- **No convergencia:** En casos extremos, la red puede no converger en absoluto, quedando atrapada en un mínimo local muy pobre.
- **Dificultad para entrenar redes profundas:** El problema se agrava a medida que aumenta la profundidad de la red, ya que hay más multiplicaciones de derivadas pequeñas en la regla de la cadena.

11.4. Soluciones

1. **Usar funciones de activación que no se saturen (o se saturen menos):**
   - **ReLU (Rectified Linear Unit):** $\text{ReLU}(z) = \text{máx}(0, z)$. Su derivada es 1 para $z > 0$ y 0 para $z < 0$. Esto evita el desvanecimiento del gradiente en la región positiva. Sin embargo, puede sufrir el problema de la "neurona muerta"si la entrada es siempre negativa.
   - **Leaky ReLU:** Una variante de ReLU que introduce una pequeña pendiente para valores negativos:

$$\text{Leaky ReLU}(z) = \begin{cases} z & \text{si } z > 0 \\ \alpha z & \text{si } z \leq 0 \end{cases}$$

     Donde α suele ser 0.01. Esto ayuda a evitar el problema de la neurona muerta.
   - **ELU (Exponential Linear Unit), SELU (Scaled Exponential Linear Unit)** y otras variantes de ReLU

2. **Inicialización de pesos cuidadosa:** Existen técnicas de inicialización de pesos diseñadas específicamente para mitigar el desvanecimiento del gradiente, como la *inicialización de Glorot* (también conocida como Xavier) y la *inicialización de He*. Estas técnicas ajustan la varianza de los pesos iniciales en función del número de entradas y salidas de cada capa.
3. **Redes Residuales (ResNets):** Las ResNets introducen conexiones de 'alto' (skip connections) que permiten que el gradiente fluya más fácilmente a través de las capas, evitando que se desvanezca. Una ResNet añade la entrada a una capa a la salida de esa capa (o de un bloque de capas). Esto crea un atajo para el gradiente.
4. **Batch Normalization:** La normalización por lotes (Batch Normalization) es una técnica que normaliza las activaciones de cada capa, lo que ayuda a mantener los gradientes en un rango razonable y acelera el entrenamiento

12. Entrenamiento de una Red Neuronal

12.1. Pasos Previos al Entrenamiento

1. **Preparación de los Datos:**
   - **Recopilación y limpieza:** Obtener un conjunto de datos relevante y de alta calidad. Esto puede implicar limpiar los datos (eliminar valores faltantes, corregir errores, etc.).
   - **División en conjuntos:** Dividir el conjunto de datos en tres subconjuntos:

- **Conjunto de entrenamiento (training set):** Se utiliza para ajustar los pesos y bias de la red.
- **Conjunto de validación (validation set):** Se utiliza para evaluar el rendimiento de la red durante el entrenamiento y ajustar hiperparámetros (como la tasa de aprendizaje, el número de capas, el número de neuronas por capa, etc.). No se utiliza para ajustar directamente los pesos.
- **Conjunto de prueba (test set):** Se utiliza para evaluar el rendimiento final de la red, una vez que el entrenamiento ha terminado. Proporciona una estimación imparcial de la capacidad de generalización de la red a datos no vistos. Es crucial que el conjunto de prueba no se utilice ni para entrenar ni para ajustar hiperparámetros.

- **Preprocesamiento:** Transformar los datos para que sean adecuados para la red neuronal. Esto puede incluir:
  - **Normalización/Estandarización:** Escalar los valores de las características para que tengan un rango similar (por ejemplo, entre 0 y 1, o con media 0 y desviación estándar 1). Esto ayuda a que el entrenamiento sea más estable y eficiente.
  - **Codificación de variables categóricas:** Convertir variables categóricas (como 'rojo', 'verde', 'azul') en representaciones numéricas (por ejemplo, usando one-hot encoding).
  - **Tratamiento de datos faltantes:** Si hay valores faltantes, decidir cómo manejarlos (eliminarlos, imputarlos, etc.).
  - **Aumento de datos (data augmentation):** Si el conjunto de datos es pequeño, se pueden generar artificialmente nuevas instancias de datos a partir de los datos existentes (por ejemplo, rotando, escalando o recortando imágenes).

2. **Definición de la Arquitectura:**
   - **Número de capas:** Decidir cuántas capas tendrá la red (incluyendo la capa de entrada y la capa de salida).
   - **Número de neuronas por capa:** Decidir cuántas neuronas tendrá cada capa.
   - **Funciones de activación:** Elegir las funciones de activación para cada capa (ReLU, sigmoid, tanh, etc.).
   - **Conexiones:** En una red totalmente conectada, todas las neuronas de una capa están conectadas a todas las neuronas de la capa siguiente. Existen otras arquitecturas (como las redes convolucionales o las redes recurrentes) con diferentes patrones de conexión.
3. **Elección de la Función de Pérdida:** Seleccionar una función de pérdida apropiada para el tipo de problema (clasificación binaria, clasificación multiclase, regresión).
4. **Elección del Optimizador:** Seleccionar un algoritmo de optimización para ajustar los pesos y bias de la red. El optimizador más común es el descenso de gradiente (en sus diferentes variantes: Batch, Stochastic, Mini-Batch), pero existen otros (Adam, RMSprop, etc.).

12.2. El Ciclo de Entrenamiento

El entrenamiento en sí es un proceso iterativo que se repite hasta que se cumple algún criterio de parada. Cada iteración completa a través del conjunto de entrenamiento se llama época (epoch).

Los pasos principales del ciclo de entrenamiento son:

1. **Forward Pass (Propagación hacia adelante):**
   - Se presentan los datos de entrada (un lote o 'mini-batch' de ejemplos) a la red.
   - La red calcula su salida, propagando la información a través de las capas, aplicando las funciones de activación en cada neurona.
2. **Cálculo de la Pérdida:**
   - Se compara la salida de la red con los valores objetivo (etiquetas reales) utilizando la función de pérdida.

- La función de pérdida produce un valor escalar que indica el error cometido por la red en ese lote.

3. **Backward Pass (Retropropagación):**
   - Se calcula el gradiente de la función de pérdida con respecto a todos los pesos y bias de la red, utilizando el algoritmo de retropropagación.
4. **Actualización de Parámetros:**
   - Se actualizan los pesos y bias de la red utilizando el optimizador (por ejemplo, el descenso de gradiente), en la dirección opuesta al gradiente.
5. **Evaluación (periódica):**
   - Cada cierto número de épocas (o iteraciones), se evalúa el rendimiento de la red en el conjunto de validación. Esto se hace para:
     - Monitorizar el progreso del entrenamiento.
     - Detectar el sobreajuste (overfitting). El sobreajuste ocurre cuando la red aprende demasiado bien los datos de entrenamiento, pero generaliza mal a nuevos datos. Se detecta cuando la pérdida en el conjunto de entrenamiento sigue disminuyendo, pero la pérdida en el conjunto de validación comienza a aumentar.
     - Ajustar hiperparámetros (como la tasa de aprendizaje).
     - Seleccionar el mejor modelo (el que tiene menor pérdida en el conjunto de validación).
6. **Criterio de Parada** Se detiene el proceso cuando se cumple un criterio, cómo por ejemplo:
   - La función de pérdida deja de disminuir.
   - Se alcanza un número máximo de iteraciones.
   - Se detecta sobreajuste.
   - Temprana (Early stopping): Se para cuando el error de validación deja de disminuir (o empieza a aumentar).

12.3. Evaluación Final

Una vez que el entrenamiento ha terminado, se evalúa el rendimiento final de la red en el conjunto de prueba. Esto proporciona una estimación imparcial de la capacidad de generalización de la red a datos no vistos. Es crucial no utilizar el conjunto de prueba durante el entrenamiento, ni para ajustar pesos ni para ajustar hiperparámetros.

13. Algoritmos de Optimización

En secciones anteriores, introdujimos el descenso de gradiente como el algoritmo fundamental para entrenar redes neuronales. Sin embargo, el descenso de gradiente básico (en sus variantes batch, mini-batch y estocástico) tiene algunas limitaciones. Existen algoritmos de optimización más avanzados que mejoran la velocidad y la estabilidad del entrenamiento.

13.1. Repaso del Descenso de Gradiente

Recordemos que la actualización de los pesos en el descenso de gradiente se realiza de la siguiente manera:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_t)$$

Donde:

- $\boldsymbol{\theta}_t$ representa el vector de todos los parámetros (pesos y bias) de la red en el instante t.
- η es la tasa de aprendizaje (learning rate).

- $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_t)$ es el gradiente de la función de pérdida con respecto a los parámetros, evaluado en $\boldsymbol{\theta}_t$.

En la práctica, a menudo se usa el descenso de gradiente por mini-lotes, donde el gradiente se calcula sobre un subconjunto de los datos de entrenamiento:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \frac{1}{n} \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} \mathcal{L}(\mathbf{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}_t)$$

Donde n es el tamaño del mini-lote.

La idea clave es moverse en la dirección opuesta al gradiente, que es la dirección de máximo descenso de la función de pérdida.

13.2. Descenso de Gradiente con Momento (Momentum)

El descenso de gradiente con momento (Momentum-Based Gradient Descent, MBGD) es una modificación del descenso de gradiente que introduce un término de inercia o momento. Este término ayuda a acelerar el entrenamiento en regiones donde el gradiente es consistente, y a suavizar las oscilaciones en regiones donde el gradiente cambia rápidamente de dirección.

La actualización de los parámetros en el descenso de gradiente con momento se realiza de la siguiente manera:

$$\begin{aligned} \mathbf{v}_t &= \gamma \mathbf{v}_{t-1} + \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_t) \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \mathbf{v}_t \end{aligned}$$

Donde:

- $\mathbf{v}_t$ es el vector de velocidad o momento en el instante t.
- γ es el coeficiente de momento (un hiperparámetro, típicamente entre 0.9 y 0.99). Controla la influencia de las actualizaciones anteriores en la actualización actual.
- η es la tasa de aprendizaje.

La intuición es la siguiente: si el gradiente ha estado apuntando consistentemente en la misma dirección durante varias iteraciones, el término de momento $\gamma \mathbf{v}_{t-1}$ se acumulará, y la actualización de los parámetros será mayor. Es como una bola rodando por una pendiente: gana velocidad a medida que desciende.

Ventajas del descenso de gradiente con momento:

- Acelera la convergencia en regiones con gradientes consistentes.
- Reduce las oscilaciones en regiones con gradientes ruidosos o que cambian rápidamente de dirección.
- Puede ayudar a escapar de mínimos locales poco profundos.

13.3. Adam (Adaptive Moment Estimation)

Adam es un algoritmo de optimización más sofisticado que combina las ideas del descenso de gradiente con momento y de otros algoritmos que adaptan la tasa de aprendizaje para cada parámetro individualmente (como RMSprop).

Adam mantiene dos estimaciones de momentos para cada parámetro:

- $\mathbf{m}_t$: Una estimación del primer momento del gradiente (la media). Captura la dirección promedio del gradiente.
- $\mathbf{v}_t$: Una estimación del segundo momento del gradiente (la varianza no centrada). Captura la magnitud de las variaciones del gradiente.

Las ecuaciones de actualización de Adam son:

$$
\begin{aligned}
\mathbf{g}_t &= \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_t) \\
\mathbf{m}_t &= \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t \\
\mathbf{v}_t &= \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)\mathbf{g}_t^2 \\
\hat{\mathbf{m}}_t &= \frac{\mathbf{m}_t}{1-\beta_1^t} \\
\hat{\mathbf{v}}_t &= \frac{\mathbf{v}_t}{1-\beta_2^t} \\
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon}
\end{aligned}
$$

Donde:

- $\mathbf{g}_t$ es el gradiente en el instante t.
- β_1 y β_2 son hiperparámetros que controlan la tasa de decaimiento exponencial de las medias móviles (típicamente, $\beta_1 = 0{,}9$ y $\beta_2 = 0{,}999$).
- $\hat{\mathbf{m}}_t$ y $\hat{\mathbf{v}}_t$ son estimaciones corregidas del sesgo de los momentos (importantes en las primeras iteraciones, cuando t es pequeño).
- ϵ es un valor pequeño (por ejemplo, 10^{-8}) para evitar la división por cero.
- η es la tasa de aprendizaje
- La notación $\mathbf{g}_t^2$ significa que cada elemento de $\mathbf{g}_t$ se eleva al cuadrado. Los cálculos con vectores son elemento a elemento.

Características de Adam:

- Adapta la tasa de aprendizaje para cada parámetro individualmente, basándose en las estimaciones de los momentos del gradiente.
- Combina las ventajas del descenso de gradiente con momento (aceleración en direcciones consistentes) y de algoritmos como RMSprop (adaptación de la tasa de aprendizaje por parámetro).
- Es uno de los algoritmos de optimización más populares y efectivos en la práctica para entrenar redes neuronales profundas.
- Generalmente requiere menos ajuste de hiperparámetros que otros algoritmos.

13.4. Otros Algoritmos de Optimización

Existen muchos otros algoritmos de optimización, como:

- **Adagrad:** Adapta la tasa de aprendizaje para cada parámetro, pero tiende a disminuirla demasiado rápido.
- **RMSprop:** Una mejora de Adagrad que evita la disminución agresiva de la tasa de aprendizaje.
- **Adadelta:** Similar a RMSprop.
- **Nadam:** Combina Adam con el momento Nesterov (una variante del momento estándar).

La elección del algoritmo de optimización puede tener un impacto significativo en el rendimiento del entrenamiento. Adam es una buena opción por defecto en muchos casos, pero vale la pena experimentar con otros algoritmos y ajustar sus hiperparámetros. Los frameworks de deep learning como PyTorch y TensorFlow proporcionan implementaciones de todos estos algoritmos.

14. El Compromiso Sesgo-Varianza (Bias-Variance Tradeoff)

El compromiso sesgo-varianza (bias-variance tradeoff) es un concepto fundamental en el aprendizaje automático, y es especialmente relevante en el contexto de las redes neuronales profundas. Se refiere al equilibrio entre dos tipos de errores que un modelo puede cometer: el error debido al sesgo y el error debido a la varianza.

14.1. Sesgo (Bias)

El sesgo se refiere al error sistemático que comete un modelo debido a suposiciones simplificadoras sobre los datos. Un modelo con alto sesgo subajusta (underfits) los datos; es decir, no es lo suficientemente complejo para capturar las relaciones subyacentes entre las características y la variable objetivo. Es como si el modelo fuera 'demasiado simple' para los datos.

Características de un modelo con alto sesgo:

- No se ajusta bien a los datos de entrenamiento (alto error de entrenamiento).
- No generaliza bien a los datos de prueba (alto error de prueba).
- Tiende a cometer el mismo tipo de error en diferentes conjuntos de datos.

14.2. Varianza (Variance)

La varianza se refiere a la sensibilidad del modelo a las fluctuaciones en los datos de entrenamiento. Un modelo con alta varianza sobreajusta (overfits) los datos; es decir, se ajusta demasiado bien a los datos de entrenamiento, incluyendo el ruido y las peculiaridades específicas de ese conjunto de datos. Es como si el modelo fuera 'demasiado complejo' para los datos. Aprende el ruido de los ejemplos.

Características de un modelo con alta varianza:

- Se ajusta muy bien a los datos de entrenamiento (bajo error de entrenamiento).
- No generaliza bien a los datos de prueba (alto error de prueba).
- Tiende a cometer diferentes tipos de errores en diferentes conjuntos de datos.

14.3. El Compromiso

El objetivo ideal es encontrar un modelo que tenga bajo sesgo y baja varianza. Sin embargo, en la práctica, existe un compromiso entre ambos.

- **Al aumentar la complejidad del modelo** (por ejemplo, añadiendo más capas o más neuronas a una red neuronal), el sesgo tiende a disminuir (el modelo se vuelve más capaz de capturar relaciones complejas), pero la varianza tiende a aumentar (el modelo se vuelve más susceptible a sobreajustar).
- **Al disminuir la complejidad del modelo**, la varianza tiende a disminuir (el modelo se vuelve menos susceptible a sobreajustar), pero el sesgo tiende a aumentar (el modelo se vuelve menos capaz de capturar relaciones complejas).

14.4. Representación Gráfica

La relación se visualiza mejor con un gráfico. Imagina que estás disparando a una diana:

- **Bajo sesgo, baja varianza:** Los disparos están agrupados cerca del centro de la diana.
- **Alto sesgo, baja varianza:** Los disparos están agrupados, pero lejos del centro de la diana.
- **Bajo sesgo, alta varianza:** Los disparos están dispersos alrededor del centro de la diana.
- **Alto sesgo, alta varianza:** Los disparos están dispersos y lejos del centro de la diana.

También es muy ilustrativa la gráfica que relaciona el Error total, con el error debido al Bias, y el error debido a la varianza:

- El error total tiene forma de U.
- A medida que aumenta la complejidad del modelo, el error debido al sesgo disminuye, pero el error debido a la varianza aumenta.
- El punto óptimo es donde el error total es mínimo.

14.5. Compromiso Sesgo-Varianza y Redes Neuronales

Las redes neuronales profundas, debido a su gran número de parámetros, tienen una alta capacidad de representación y, por lo tanto, son propensas al sobreajuste (alta varianza). Es decir, tienden a tener bajo sesgo, pero alta varianza si no se toman medidas.

14.6. Técnicas para Abordar el Compromiso Sesgo-Varianza

Existen varias técnicas para mitigar el sobreajuste y encontrar un buen equilibrio entre sesgo y varianza:

- **Regularización:** Añade un término de penalización a la función de pérdida que penaliza los modelos con pesos grandes. Esto ayuda a evitar que el modelo se ajuste demasiado a los datos de entrenamiento. Las técnicas de regularización más comunes son:
  - **Regularización L1 (Lasso):** Penaliza la suma de los valores absolutos de los pesos.
  - **Regularización L2 (Ridge):** Penaliza la suma de los cuadrados de los pesos.
- **Dropout:** Desactiva aleatoriamente neuronas durante el entrenamiento. Esto obliga a la red a aprender representaciones más robustas y evita que dependa demasiado de neuronas individuales.
- **Parada temprana (early stopping):** Detiene el entrenamiento cuando el error en el conjunto de validación comienza a aumentar, incluso si el error en el conjunto de entrenamiento sigue disminuyendo.
- **Aumento de datos (data augmentation):** Genera artificialmente nuevas instancias de datos a partir de los datos existentes. Esto aumenta el tamaño del conjunto de entrenamiento y ayuda a que el modelo generalice mejor.
- **Batch Normalization:** Normaliza las activaciones dentro de cada mini-lote.
- **Reducir la complejidad del modelo:** Disminuir el número de capas o neuronas si el modelo está sobreajustando.
- **Aumentar la complejidad del modelo:** Aumentar el número de capas o neuronas si el modelo está subajustando.
- **Conseguir más datos de entrenamiento:** Si es posible, obtener más datos de entrenamiento es la mejor manera de reducir la varianza.

15. Regularización: L1, L2 y Dropout

La regularización es un conjunto de técnicas que se utilizan para prevenir el sobreajuste (overfitting) en modelos de aprendizaje automático, especialmente en redes neuronales profundas. El sobreajuste ocurre cuando un modelo se ajusta demasiado bien a los datos de entrenamiento, aprendiendo incluso el ruido y las peculiaridades específicas de ese conjunto de datos, y generaliza mal a nuevos datos.

15.1. ¿Qué es la Regularización?

La idea fundamental de la regularización es añadir una penalización a la función de pérdida que depende de la complejidad del modelo. De esta forma, se desincentiva que el modelo se vuelva demasiado complejo y se ajuste en exceso a los datos de entrenamiento. En la práctica, la regularización aumenta el sesgo del modelo para reducir su varianza.

15.2. Regularización L1 (Lasso)

La regularización L1, también conocida como Lasso (Least Absolute Shrinkage and Selection Operator), añade un término de penalización a la función de pérdida que es proporcional a la suma de los valores absolutos de los pesos.

Si la función de pérdida original es $\mathcal{L}$, la función de pérdida con regularización L1 se convierte en:

$$\mathcal{L}_{\text{L1}} = \mathcal{L} + \lambda \sum_{i=1}^{n} |w_i|$$

Donde:

- $\mathcal{L}$ es la función de pérdida original (por ejemplo, MSE para regresión o entropía cruzada para clasificación).
- λ es un hiperparámetro que controla la fuerza de la regularización (también conocido como *coeficiente de regularización*). Un valor mayor de λ implica una mayor penalización por pesos grandes.
- n es el número total de pesos en el modelo.
- w_i es el i-ésimo peso del modelo.
- $\sum_{i=1}^{n} |w_i|$ es la norma L1 de los pesos (la suma de sus valores absolutos). A veces se escribe como $||\mathbf{w}||_1$

Características de la regularización L1:

- **Promueve la dispersión (sparsity):** La regularización L1 tiende a producir soluciones dispersas, donde muchos de los pesos son exactamente cero. Esto se debe a la forma de la norma L1 (un diamante en 2D, un poliedro en dimensiones superiores). Las soluciones óptimas tienden a encontrarse en los vértices de este poliedro, donde muchos de los pesos son cero.
- **Selección de características:** Al hacer que muchos pesos sean cero, la regularización L1 actúa como un mecanismo de selección de características. Las características correspondientes a los pesos que se han hecho cero se consideran irrelevantes para el modelo.
- **Interpretación:** La dispersión inducida por la regularización L1 puede hacer que el modelo sea más interpretable, ya que se puede identificar un subconjunto más pequeño de características importantes.

15.3. Regularización L2 (Ridge)

La regularización L2, también conocida como Ridge o weight decay, añade un término de penalización a la función de pérdida que es proporcional a la suma de los *cuadrados* de los pesos.

La función de pérdida con regularización L2 se convierte en:

$$\mathcal{L}_{\text{L2}} = \mathcal{L} + \lambda \sum_{i=1}^{n} w_i^2$$

Donde:

- λ es el hiperparámetro que controla la fuerza de la regularización.
- $\sum_{i=1}^{n} w_i^2$ es la norma L2 al cuadrado de los pesos. A veces se escribe como $||\mathbf{w}||_2^2$

Características de la regularización L2:

- **Pesos pequeños:** La regularización L2 tiende a producir soluciones con pesos pequeños, pero no necesariamente cero. La penalización por pesos grandes es menos agresiva que en la regularización L1.
- **Estabilidad numérica:** La regularización L2 puede mejorar la estabilidad numérica del entrenamiento, especialmente cuando las características están altamente correlacionadas.
- **No promueve la dispersión:** A diferencia de L1, L2 no induce dispersión en los pesos.

15.4. Comparación entre L1 y L2

- L1 produce soluciones dispersas (muchos pesos iguales a cero), mientras que L2 produce soluciones con pesos pequeños pero no necesariamente cero.
- L1 realiza selección de características, mientras que L2 no.
- L2 suele tener mejor rendimiento predictivo que L1, especialmente cuando hay muchas características y no se espera que la solución sea dispersa.

¿Por qué funcionan?
Al añadir una penalización por pesos, se favorecen modelos más 'sencillos', ya que modelos complejos necesitan ajustar muchos pesos a valores altos. Un modelo más simple generaliza mejor.

15.5. Dropout

Dropout es una técnica de regularización diferente a L1 y L2. En lugar de modificar la función de pérdida, dropout modifica la arquitectura de la red neuronal durante el entrenamiento.

Durante cada iteración de entrenamiento, dropout desactiva aleatoriamente una fracción de las neuronas de la red. Esto significa que esas neuronas no participan ni en el forward pass ni en el backward pass. La fracción de neuronas desactivadas se controla mediante un hiperparámetro p (la tasa de dropout, dropout rate), que típicamente se establece entre 0.2 y 0.5.

En cada iteración, se selecciona un subconjunto diferente de neuronas para ser desactivadas. Esto es equivalente a entrenar un gran número de redes neuronales diferentes, cada una con una arquitectura ligeramente distinta. En la fase de prueba (inferencia), se utilizan todas las neuronas, pero sus salidas se escalan por un factor de $1 - p$ para compensar el hecho de que no se aplicó dropout durante el entrenamiento.

- **Evita la co-adaptación de neuronas:** Al desactivar aleatoriamente neuronas, dropout evita que las neuronas se vuelvan demasiado dependientes unas de otras. Cada neurona se ve obligada a aprender características útiles por sí misma, en lugar de depender de la presencia de otras neuronas específicas.
- **Entrena un 'ensemble' de modelos:** Dropout puede interpretarse como una forma de entrenar un ensemble (conjunto) de redes neuronales diferentes. Cada red neuronal del ensemble tiene una arquitectura ligeramente distinta debido a la desactivación aleatoria de neuronas. En la fase de prueba, se promedian las predicciones de todas estas redes (implícitamente, al escalar las salidas).
- **Mejora la generalización:** Al evitar la co-adaptación y entrenar un ensemble de modelos, dropout mejora la capacidad de generalización de la red y reduce el sobreajuste.
- Es una técnica de regularización muy efectiva.
- Es computacionalmente eficiente.
- Es fácil de implementar.

16. Batch Normalization

Batch Normalization (BN) es una técnica utilizada en redes neuronales profundas para mejorar la estabilidad y la velocidad del entrenamiento, y en muchos casos, mejorar también el rendimiento del modelo. No se considera estrictamente una técnica de regularización como L1, L2 o Dropout (aunque a menudo tiene un efecto regularizador), sino más bien una técnica de normalización de activaciones.

16.1. El Problema del Internal Covariate Shift

Durante el entrenamiento de una red neuronal, la distribución de las entradas a cada capa cambia a medida que se actualizan los pesos de las capas anteriores. Este fenómeno se conoce como internal covariate shift. Este cambio constante en la distribución de las entradas dificulta el entrenamiento, ya que cada capa tiene que adaptarse continuamente a una nueva distribución.

16.2. ¿Cómo Funciona Batch Normalization?

Batch Normalization aborda el problema del internal covariate shift normalizando las activaciones de cada capa. En esencia, hace que las activaciones tengan una media cercana a 0 y una desviación estándar cercana a 1 para cada mini-lote (batch) de datos.

Los pasos para aplicar Batch Normalization a una capa específica son los siguientes:

1. **Calcular la media y la varianza del mini-lote:** Para cada neurona j de la capa, se calcula la media (μ_j) y la varianza (σ_j^2) de sus activaciones *a través de los ejemplos del mini-lote*.

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} h_{ij}$$

$$\sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(h_{ij} - \mu_j)^2$$

   Donde:

   - m es el tamaño del mini-lote.
   - h_{ij} es la activación de la neurona j para el ejemplo i del mini-lote (antes de la normalización). Es la salida de la capa anterior, antes de aplicar la función de activación, o bien, la salida de la función de activación si BN se aplica después de ella (menos común).

2. **Normalizar las activaciones:** Se normalizan las activaciones de cada neurona restando la media del mini-lote y dividiendo por la desviación estándar del mini-lote (más un pequeño valor ϵ para evitar la división por cero):

$$\hat{h}_{ij} = \frac{h_{ij} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

   Donde $\hat{h}_{ij}$ es la activación normalizada.

3. **Escalar y desplazar:** Se introduce un escalado (γ_j) y un desplazamiento (β_j) para cada neurona. Estos parámetros son *aprendidos* durante el entrenamiento, junto con los pesos y bias de la red. Permiten a la red recuperar la capacidad de representación original de la capa, si es necesario. Si $\gamma_j = \sqrt{\sigma_j^2}$ y $\beta_j = \mu_j$, la transformación se revierte.

$$\tilde{h}_{ij} = \gamma_j \hat{h}_{ij} + \beta_j$$

   Donde $\tilde{h}_{ij}$ es la salida final de la capa de Batch Normalization.

16.3. Batch Normalization en la Fase de Inferencia (Test)

Durante el entrenamiento, la media y la varianza se calculan para cada mini-lote. Sin embargo, en la fase de inferencia (cuando se utiliza la red para hacer predicciones sobre nuevos datos), no se tiene un mini-lote. En su lugar, se utilizan estimaciones de la media y la varianza poblacionales, que se calculan durante el entrenamiento.

Hay varias formas de obtener estas estimaciones. Una forma común es utilizar una media móvil exponencial de las medias y varianzas de los mini-lotes durante el entrenamiento.

16.4. Posición de la Capa de Batch Normalization

Normalmente, una capa BN se inserta después de la capa lineal (la multiplicación de matriz $\mathbf{Wx} + \mathbf{b}$), pero antes de aplicar la función de activación. Aunque se puede usar en otras posiciones, esta es la más habitual y recomendada.

16.5. Beneficios de Batch Normalization

- **Acelera el entrenamiento:** Permite utilizar tasas de aprendizaje más altas sin riesgo de divergencia. Al normalizar las activaciones, se reduce el problema del desvanecimiento/explosión del gradiente, y el entrenamiento converge más rápido.
- **Reduce la dependencia de la inicialización de pesos:** El entrenamiento es menos sensible a la inicialización de los pesos.
- **Regulariza (ligeramente):** Al introducir ruido en el cálculo de la media y la varianza (ya que se calculan sobre mini-lotes), Batch Normalization actúa como un regularizador, reduciendo ligeramente el sobreajuste. Sin embargo, no debe considerarse como un sustituto de otras técnicas de regularización como Dropout o L1/L2.
- **Mejora la estabilidad del entrenamiento:** Reduce el 'internal covariate shift', haciendo que el entrenamiento sea más estable.
- **Permite entrenar redes más profundas:** Al mitigar el problema del desvanecimiento/explosión del gradiente, Batch Normalization facilita el entrenamiento de redes neuronales más profundas.

17. Otras Estrategias Importantes en el Entrenamiento de Redes Neuronales

Además de las técnicas ya mencionadas (regularización, optimizadores avanzados, Batch Normalization), existen otras estrategias importantes que pueden mejorar significativamente el entrenamiento y el rendimiento de las redes neuronales profundas.

17.1. Parada Temprana (Early Stopping)

La parada temprana es una técnica sencilla pero poderosa para prevenir el sobreajuste y mejorar la eficiencia del entrenamiento. La idea básica es monitorizar el rendimiento del modelo en un conjunto de *validación* durante el entrenamiento, y detener el entrenamiento cuando el rendimiento en el conjunto de validación deja de mejorar (o comienza a empeorar).

1. Se divide el conjunto de datos en entrenamiento, validación y prueba.
2. Se entrena la red utilizando el conjunto de entrenamiento, aplicando el algoritmo de optimización (por ejemplo, descenso de gradiente con momento o Adam).
3. Periódicamente (por ejemplo, cada época o cada cierto número de iteraciones), se evalúa el rendimiento del modelo en el conjunto de validación (sin actualizar los pesos). Se calcula la función de pérdida (o alguna otra métrica de rendimiento, como la precisión) en el conjunto de validación.

4. Se guarda el modelo (los pesos y bias) que obtiene el mejor rendimiento en el conjunto de validación.
5. Si el rendimiento en el conjunto de validación no mejora durante un cierto número de evaluaciones consecutivas (este número se llama paciencia, (patience)), se detiene el entrenamiento.
6. Se recupera el modelo guardado con el mejor rendimiento en el conjunto de validación.

- **Previene el sobreajuste:** Al detener el entrenamiento antes de que el modelo comience a memorizar los datos de entrenamiento, se evita el sobreajuste y se mejora la capacidad de generalización.
- **Ahorra tiempo y recursos computacionales:** El entrenamiento se detiene antes de que se alcance la convergencia completa, lo que puede ahorrar una cantidad significativa de tiempo y recursos.
- **Es simple de implementar:** La mayoría de los frameworks de deep learning proporcionan funcionalidades para implementar la parada temprana.
- **Selecciona automáticamente un buen modelo:** No hace falta saber de antemano cuántas iteraciones harán falta.

17.2. Aumento de Datos (Data Augmentation)

El aumento de datos es una técnica que consiste en generar artificialmente nuevas instancias de datos de entrenamiento a partir de los datos existentes, aplicando transformaciones aleatorias que preservan la semántica de los datos. Es particularmente útil cuando se tiene un conjunto de datos de entrenamiento pequeño.

Ejemplos de transformaciones (dependiendo del tipo de datos):

- **Imágenes:**
  - Rotaciones
  - Escalado (zoom in/out)
  - Recortes (cropping)
  - Traslaciones
  - Volteos horizontales o verticales (flipping)
  - Cambios de color, brillo y contraste
  - Adición de ruido
- **Audio:**
  - Cambios de tono (pitch shifting)
  - Estiramiento temporal (time stretching)
  - Adición de ruido de fondo
- **Texto:**
  - Reemplazo de palabras por sinónimos
  - Inserción o eliminación aleatoria de palabras
  - Traducción a otro idioma y retrotraducción

Beneficios del aumento de datos:

- **Aumenta el tamaño efectivo del conjunto de entrenamiento:** Al generar nuevas instancias, se reduce el riesgo de sobreajuste.
- **Mejora la generalización:** El modelo se vuelve más robusto a variaciones en los datos de entrada.
- **Introduce variabilidad:** Ayuda al modelo a aprender invariancias (por ejemplo, un clasificador de imágenes debe ser invariante a la rotación o al escalado de los objetos).

17.3. Inicialización de Pesos

La inicialización de los pesos es un aspecto crucial del entrenamiento de redes neuronales. Una mala inicialización puede llevar a un entrenamiento lento o incluso a la no convergencia.

Problemas con una mala inicialización:

- **Todos los pesos iguales (constantes):** Si todos los pesos se inicializan con el mismo valor, todas las neuronas de una capa aprenderán lo mismo, y la red no podrá romper la simetría. Será como tener una sola neurona en cada capa.
- **Pesos aleatorios demasiado pequeños:** Puede llevar al problema del desvanecimiento del gradiente.
- **Pesos aleatorios demasiado grandes:** Puede llevar al problema de la explosión del gradiente.

Estrategias de inicialización:

- **Inicialización aleatoria simple:** Inicializar los pesos con valores aleatorios muestreados de una distribución uniforme o normal. Sin embargo, es importante ajustar la varianza de la distribución.
- **Inicialización de Glorot (Xavier):** Una estrategia diseñada para mantener la varianza de las activaciones y los gradientes constante a través de las capas. Los pesos se muestrean de una distribución:
  - Uniforme: $w \sim U\left[-\sqrt{\frac{6}{n_{in}+n_{out}}}, \sqrt{\frac{6}{n_{in}+n_{out}}}\right]$
  - Normal: $w \sim N\left(0, \sqrt{\frac{2}{n_{in}+n_{out}}}\right)$

  Donde n_{in} es el número de entradas a la capa y n_{out} es el número de salidas de la capa. Funciona bien con funciones de activación como la sigmoide y la tangente hiperbólica.
- **Inicialización de He:** Similar a la inicialización de Glorot, pero diseñada específicamente para funciones de activación ReLU. Los pesos se muestrean de una distribución:
  - Uniforme: $w \sim U\left[-\sqrt{\frac{6}{n_{in}}}, \sqrt{\frac{6}{n_{in}}}\right]$
  - Normal: $w \sim N\left(0, \sqrt{\frac{2}{n_{in}}}\right)$

  Donde n_{in} es el número de entradas a la capa.

18. Redes Convolucionales (Convolutional Neural Networks - CNNs)

18.1. Introducción

Las redes neuronales convolucionales (CNNs o ConvNets) son un tipo de red neuronal profunda especialmente diseñadas para procesar datos con una estructura de cuadrícula (grid-like topology), como imágenes (2D) o señales de audio (1D). Son la arquitectura dominante en el campo de la visión artificial, aunque también se aplican con éxito en otras áreas, como el procesamiento del lenguaje natural.

18.2. Motivación: Limitaciones de las Redes Totalmente Conectadas para Imágenes

Las redes neuronales totalmente conectadas (FCNs), que estudiamos anteriormente, presentan varios inconvenientes cuando se aplican a imágenes:

1. **Pérdida de información espacial:** Una FCN trata una imagen como un vector unidimensional, aplanando (flattening) la estructura 2D (o 3D, si consideramos los canales de color). Esto destruye la información espacial inherente a la imagen, como la relación entre píxeles adyacentes.

2. **Explosión de parámetros:** Incluso para imágenes pequeñas, una FCN con una sola capa oculta tendría una cantidad enorme de parámetros. Por ejemplo, una imagen de 200x200 píxeles a color (3 canales) tendría 200 * 200 * 3 = 120,000 entradas. Si la capa oculta tuviera 1000 neuronas, la matriz de pesos de la primera capa tendría 120,000 * 1000 = 120 millones de parámetros. Esto hace que el entrenamiento sea computacionalmente costoso y propenso al sobreajuste.

3. **Falta de invarianza traslacional:** Una FCN no es inherentemente invariante a traslaciones. Si un objeto en una imagen se mueve a una posición diferente, la FCN tiene que aprender un patrón completamente nuevo.

Las CNNs abordan estos problemas mediante el uso de convoluciones y pooling.

18.3. Tareas de Visión Artificial

Antes de profundizar en la arquitectura de las CNNs, es útil repasar las principales tareas en visión artificial:

- **Clasificación de imágenes:** Asignar una etiqueta a una imagen completa (por ejemplo, 'gato', 'perro', 'coche').
- **Detección de objetos:** Localizar y clasificar múltiples objetos dentro de una imagen. Se suelen utilizar 'cajas delimitadoras' (bounding boxes) para indicar la ubicación de los objetos.
- **Segmentación de imágenes:**
  - **Segmentación semántica:** Asignar una etiqueta a cada píxel de la imagen, clasificando cada píxel en una categoría semántica (por ejemplo, 'carretera').
  - **Segmentación de instancias:** Similar a la segmentación semántica, pero distingue entre diferentes instancias del mismo objeto (por ejemplo, 'coche1').

18.4. Historia Breve de las CNNs

- **1980s-1990s**: Yann LeCun y sus colegas desarrollan las primeras CNNs exitosas, aplicadas al reconocimiento de dígitos manuscritos (LeNet). Ya se usaban convoluciones, pooling y retropropagación.
- **2012**: AlexNet, una CNN profunda, gana el concurso ImageNet de clasificación de imágenes por un amplio margen, marcando el resurgimiento de las CNNs y el inicio de la era del 'deep learning' en visión artificial. El éxito de AlexNet se debió en parte al uso de GPUs para acelerar el entrenamiento, a la disponibilidad de grandes conjuntos de datos (ImageNet) y al uso de la función de activación ReLU.

Entre 1998 (LeNet) y 2012 (AlexNet) ocurrieron avances fundamentales:

- **Disponibilidad de grandes conjuntos de datos:** ImageNet, COCO, etc.
- **Aumento de la potencia de cálculo:** GPUs y TPUs.
- **Desarrollo de frameworks de deep learning:** TensorFlow, Keras, PyTorch, etc.

18.5. Estructura General de una CNN

Una CNN típica consta de dos partes principales:

1. **Extractor de características (Feature Extractor):** Una serie de capas convolucionales y capas de pooling que aprenden a extraer características relevantes de la imagen de entrada.
2. **Clasificador (Classifier):** Una o más capas totalmente conectadas (o una capa de 'global average pooling') que toman las características extraídas por la parte convolucional y producen la salida final (por ejemplo, las probabilidades de cada clase).

La idea clave es que la parte convolucional aprende jerarquías de características. Las primeras capas convolucionales aprenden características de bajo nivel, como bordes y texturas. Las capas convolucionales posteriores combinan estas características de bajo nivel para formar características de nivel superior, como partes de objetos. Las capas de pooling reducen la resolución espacial de los mapas de características, haciendo que la red sea más robusta a pequeñas variaciones en la posición de los objetos.

En las siguientes secciones, profundizaremos en las capas convolucionales, las capas de pooling y otros componentes de las CNNs.

19. Diseño de Arquitecturas de Redes Convolucionales

Una vez comprendidos los componentes básicos de una CNN (capas convolucionales, capas de pooling, capas totalmente conectadas, funciones de activación, etc.), surge la pregunta de cómo combinar estos componentes para construir una arquitectura efectiva para una tarea específica. No existe una receta única para diseñar una CNN, pero hay principios generales y heurísticas que pueden guiar el proceso.

19.1. Arquitectura General

Una CNN típica para clasificación de imágenes sigue el siguiente patrón general:

```
ENTRADA -> [CONV -> RELU -> POOL?]*N -> [FC -> RELU]*M -> FC
```

Donde:

- 'ENTRADA' es la imagen de entrada (típicamente un tensor 3D: ancho x alto x canales).
- 'CONV' es una capa convolucional.
- 'RELU' es la función de activación ReLU (o una variante, como Leaky ReLU).
- 'POOL' es una capa de pooling (generalmente max pooling).
- 'FC' es una capa totalmente conectada.
- '*N' indica que el bloque '[CONV -¿RELU -¿POOL?]' se repite N veces. El '?' después de 'POOL' indica que la capa de pooling es opcional (no todas las capas convolucionales van seguidas de una capa de pooling).
- '*M' indica que el bloque '[FC -¿RELU]' se repite M veces.

La parte '[CONV -¿RELU -¿POOL?]*N' actúa como un extractor de características. Las primeras capas convolucionales aprenden características de bajo nivel (bordes, texturas), y las capas posteriores combinan estas características para formar representaciones de mayor nivel. Las capas de pooling reducen la resolución espacial de los mapas de características.

La parte '[FC -¿RELU]*M -¿FC' actúa como un clasificador. Toma las características extraídas por la parte convolucional y produce la salida final (por ejemplo, las probabilidades de cada clase). A menudo, la última capa FC no tiene función de activación (o tiene una función de activación lineal), y su salida se pasa a una función softmax para obtener las probabilidades de cada clase.

19.2. Componentes Clave y sus Hiperparámetros

19.2.1. Capa Convolucional (CONV)

Extrae características locales de la entrada aplicando un conjunto de filtros (kernels) convolucionales.
Hiperparámetros:

- 'Número de filtros (K)': Determina la cantidad de mapas de características de salida. Cada filtro aprende un patrón diferente.

- 'Tamaño del filtro (F)': La dimensión espacial del filtro (típicamente 3x3 o 5x5). Filtros más pequeños capturan detalles más finos, mientras que filtros más grandes capturan patrones más amplios. Los filtros casi siempre tienen un tamaño impar.
- 'Stride (S)': El número de píxeles que el filtro se desplaza en cada paso. Un stride de 1 significa que el filtro se mueve un píxel a la vez. Un stride mayor reduce la resolución espacial de la salida.
- 'Padding (P)': La cantidad de píxeles que se añaden al borde de la entrada. El padding 'same' añade suficientes píxeles para que la salida tenga la misma dimensión espacial que la entrada (cuando el stride es 1). El padding 'valid' no añade ningún píxel.
- 'Función de activación': Normalmente ReLU (o una variante) se aplica después de la convolución.
- Dilatación (dilation) (Opcional): Especifica un espaciado entre los puntos del kernel; un valor mayor a 1.

Parámetros: Los pesos de los filtros y los bias (un bias por cada filtro).
Cálculo de la dimensión de salida:** Si la entrada tiene dimensión $W_1 \times H_1 \times D$, el filtro tiene tamaño $F \times F$, el stride es S, y el padding es P, la salida tendrá dimensión $W_2 \times H_2 \times K$, donde:

$$W_2 = \frac{W_1 - F + 2P}{S} + 1$$
$$H_2 = \frac{H_1 - F + 2P}{S} + 1$$

19.2.2. Capa de Pooling (POOL)

Reducir la resolución espacial de los mapas de características, disminuyendo la cantidad de parámetros y la carga computacional, y haciendo que la red sea más robusta a pequeñas variaciones en la posición de las características.
Hiperparámetros:

- 'Tamaño del filtro (F)': La dimensión espacial de la ventana de pooling (típicamente 2x2).
- 'Stride (S)': El número de píxeles que la ventana se desplaza en cada paso (típicamente 2).
- Tipos de pooling: 'Max pooling': Toma el valor máximo dentro de la ventana. 'Average pooling': Toma el valor promedio dentro de la ventana.

Parámetros: Las capas de pooling no tienen parámetros aprendibles.
Cálculo de dimensiones: Con un filtro de $F \times F$ y un stride de S, si la entrada tiene dimensión $W_1 \times H_1 \times D$, la salida de la capa de pooling será:

$$W_2 = \frac{W_1 - F}{S} + 1$$
$$H_2 = \frac{H_1 - F}{S} + 1$$

La profundidad (D) no cambia.

19.2.3. Capa Totalmente Conectada (FC)

Combina las características extraídas por las capas convolucionales y de pooling para realizar la clasificación (o regresión). Hiperpárametros: 'Número de neuronas (K)': La dimensión del vector de salida de la capa. Parámetros: Los pesos de la matriz de pesos y los bias.

19.2.4. Capa de Aplanamiento (Flatten)

Transforma la salida multidimensional de las capas convolucionales/pooling a un vector unidimensional para conectarlo a las capas FC. No tiene parámetros ni hiperparámetros.

19.2.5. Capa de Agrupamiento Global (Global Pooling)

Similar a la capa de Pooling, pero el filtro tiene el tamaño de todo el mapa de características. Reduce cada mapa a un único valor. Se usa a menudo como alternativa al Flatten antes de la capa de clasificación.

19.3. Recomendaciones Generales para el Diseño

- **Usar capas convolucionales con filtros pequeños (3x3 o 5x5):** Apilar varias capas convolucionales con filtros pequeños es más eficiente (en términos de número de parámetros) y permite aprender representaciones más complejas que usar una sola capa convolucional con un filtro grande.
- **Usar padding 'same':** Esto facilita el diseño de la red, ya que la dimensión espacial de la salida de cada capa convolucional es la misma que la de la entrada (si el stride es 1).
- **Usar stride 1 en las capas convolucionales (generalmente):** Un stride mayor reduce la resolución espacial demasiado rápido.
- **Usar max pooling con filtro 2x2 y stride 2:** Esta es una configuración muy común que reduce la resolución espacial a la mitad.
- **Usar ReLU (o variantes)** como función de activación en las capas ocultas.**
- **Usar Batch Normalization:** Casi siempre es beneficioso.
- **Usar Dropout para regularizar.**
- **Inicializar los pesos con una estrategia adecuada (Glorot/Xavier o He).**
- **Comenzar con arquitecturas conocidas:** En lugar de diseñar una arquitectura desde cero, es recomendable comenzar con una arquitectura conocida que haya demostrado buen rendimiento en tareas similares (por ejemplo, VGG, ResNet, Inception) y adaptarla a tu problema específico. Esto se conoce como *transfer learning*.
- **Agrupar varias capas convolucionales:** Es común agrupar 2 o 3 capas convolucionales antes de aplicar una capa de pooling.

Una arquitectura básica podría tener la siguiente estructura: 'INPUT -¿[[CONV -¿RELU]*N -¿POOL?]*M -¿[FC -¿RELU]*K -¿FC' Donde '0¡=N¡=3', 'M¿=0', y '0¡=K¡=3'

No existe una única solución correcta. El diseño de una CNN a menudo implica experimentación y ajuste iterativo.

20. Principales Arquitecturas de Redes Convolucionales

A lo largo de los años, se han propuesto diversas arquitecturas de redes convolucionales (CNNs) que han ido mejorando el rendimiento en tareas de visión artificial, especialmente en clasificación de imágenes. Muchas de estas arquitecturas se han desarrollado y evaluado en el contexto del ImageNet Large Scale Visual Recognition Challenge (ILSVRC), una competición anual que se celebró entre 2010 y 2017.

20.1. ImageNet y el ILSVRC

ImageNet es una gran base de datos de imágenes etiquetadas, organizada según la jerarquía de WordNet. Contiene millones de imágenes y miles de categorías. El ILSVRC fue una competición que utilizaba un subconjunto de ImageNet para evaluar algoritmos de clasificación y detección de objetos a gran escala. El ILSVRC impulsó gran parte del progreso en visión artificial en la década de 2010.

20.2. Arquitecturas Clave

20.2.1. LeNet-5 (1998)

- **Autores:** Yann LeCun, Léon Bottou, Yoshua Bengio y Patrick Haffner.
- **Contexto:** Diseñada para el reconocimiento de dígitos manuscritos (MNIST).
- **Arquitectura:** Relativamente simple, con dos capas convolucionales seguidas de capas de subsampling (pooling), y luego capas totalmente conectadas.
- **Contribuciones:**
  - Demostró la viabilidad de las CNNs para el reconocimiento de patrones.
  - Utilizó convoluciones, subsampling (pooling) y retropropagación.
  - Función de activación sigmoide.
- **Limitaciones:** No era lo suficientemente profunda para tareas más complejas.

20.2.2. AlexNet (2012)

- **Autores:** Alex Krizhevsky, Ilya Sutskever y Geoffrey Hinton.
- **Contexto:** Ganadora del ILSVRC 2012, marcando el resurgimiento de las redes neuronales profundas.
- **Arquitectura:** Mucho más profunda que LeNet-5, con cinco capas convolucionales y tres capas totalmente conectadas.
- **Contribuciones:**
  - Uso de la función de activación ReLU, que acelera el entrenamiento y mitiga el problema del desvanecimiento del gradiente.
  - Uso de Dropout para regularización.
  - Uso de GPUs para acelerar el entrenamiento.
  - Aumento de datos (data augmentation).
  - Profundidad: 8 capas (5 convolucionales + 3 FC).

20.2.3. ZFNet (2013)

- **Autores:** Matthew Zeiler y Rob Fergus.
- **Contexto:** Ganadora del ILSVRC 2013. Esencialmente, una versión mejorada y visualizable de AlexNet.
- **Contribuciones:**
  - Ajuste de los hiperparámetros de AlexNet (tamaño de los filtros, stride, número de filtros) para mejorar el rendimiento.
  - Visualización de las características aprendidas por las capas convolucionales, lo que permitió una mejor comprensión del funcionamiento interno de la red.
  - Profundidad: 8 capas.
  - Filtros más pequeños en las primeras capas y mayor número de filtros.

20.2.4. VGGNet (2014)

- **Autores:** Karen Simonyan y Andrew Zisserman (Visual Geometry Group, Universidad de Oxford).
- **Contexto:** Participación destacada en el ILSVRC 2014.
- **Arquitectura:** Caracterizada por su simplicidad y uniformidad. Utiliza solo filtros convolucionales de 3x3 con stride 1 y padding 'same', y max pooling de 2x2 con stride 2. Varias versiones con diferente profundidad (VGG-16, VGG-19).
- **Contribuciones:**
  - Demostró que la profundidad es crucial para el rendimiento.
  - Diseño simple y modular, fácil de entender y replicar.
  - Uso exclusivo de filtros 3x3. Muestra que varias capas con filtros pequeños pueden tener el mismo campo receptivo que una con filtros mayores.
  - Profundidad: 16-19 capas.

20.2.5. GoogLeNet (Inception) (2014)

- **Autores:** Christian Szegedy et al. (Google).
- **Contexto:** Ganadora del ILSVRC 2014.
- **Arquitectura:** Introduce el concepto de 'módulo Inception', que permite a la red aprender características a diferentes escalas simultáneamente. Un módulo Inception contiene convoluciones con diferentes tamaños de filtro (1x1, 3x3, 5x5) y max pooling, todas operando en paralelo, y sus salidas se concatenan.
- **Contribuciones:**
  - Mayor eficiencia computacional que VGGNet, a pesar de tener una mayor profundidad (22 capas).
  - Uso de convoluciones 1x1 para reducir la dimensionalidad (y el número de parámetros).
  - Uso de "auxiliary classifiers"(clasificadores auxiliares) conectados a capas intermedias para combatir el desvanecimiento del gradiente.
  - Profundidad: 22 capas.

20.2.6. ResNet (Residual Network) (2015)

- **Autores:** Kaiming He, Xiangyu Zhang, Shaoqing Ren y Jian Sun (Microsoft Research).
- **Contexto:** Ganadora del ILSVRC 2015.
- **Arquitectura:** Introduce el concepto de "conexiones residuales"(residual connections o skip connections). Una conexión residual suma la entrada a un bloque de capas a la salida de ese bloque.
- **Contribuciones:**
  - Permitió entrenar redes mucho más profundas (hasta 152 capas, y más) sin sufrir el problema del desvanecimiento del gradiente. Las conexiones residuales facilitan el flujo del gradiente durante la retropropagación.
  - Mejora significativa del rendimiento en comparación con arquitecturas anteriores.
  - Uso extensivo de Batch Normalization
  - Profundidad: Hasta 152 capas (y más).

20.2.7. DenseNet (Densely Connected Convolutional Network) (2017)

- **Autores:** Gao Huang, Zhuang Liu, Laurens van der Maaten, y Kilian Q. Weinberger.
- **Arquitectura:** Cada capa está conectada a todas las capas posteriores dentro de un 'bloque denso'.
- **Contribuciones:**
  - Flujo de información y gradientes mejorado.
  - Reduce el número de parámetros.
  - Fomenta la reutilización de características.

20.2.8. MobileNet (2017)

- **Autores:** Andrew G. Howard et al. (Google).
- **Arquitectura:** Utiliza 'convoluciones separables en profundidad' (depthwise separable convolutions) para reducir drásticamente el número de parámetros y la carga computacional, haciendo que la red sea adecuada para dispositivos móviles y embebidos.
- **Contribuciones:**
  - Eficiencia: Modelos mucho más pequeños y rápidos que otras CNNs, con una precisión razonable.
  - Convoluciones separables: Factoriza una convolución estándar en una convolución en profundidad (depthwise convolution) y una convolución puntual (pointwise convolution).

20.2.9. EfficientNet (2019)

- **Autores:** Mingxing Tan y Quoc V. Le (Google).
- **Arquitectura:** Propone un método sistemático para escalar las dimensiones de una CNN (ancho, profundidad, resolución) de forma equilibrada, utilizando un 'coeficiente compuesto'.
- **Contribuciones:**
  - Eficiencia: Logra un rendimiento superior a otras CNNs con un menor número de parámetros y operaciones.
  - Escalado compuesto: Muestra que escalar todas las dimensiones de la red (ancho, profundidad, resolución) de forma equilibrada es más efectivo que escalar solo una o dos.

20.2.10. Vision Transformers (ViT) (2020)

- **Autores:** Alexey Dosovitskiy et al. (Google)
- **Arquitectura:** Aplica la arquitectura Transformer, originalmente desarrollada para procesamiento del lenguaje natural, a la visión por computador. Divide la imagen en 'parches' (patches) y los trata como 'tokens' (similares a las palabras en un texto). Utiliza el mecanismo de auto-atención (self-attention) para modelar las relaciones entre los parches.
- **Contribuciones:**
  - Demuestra que los Transformers pueden competir con (y a veces superar a) las CNNs en tareas de visión, incluso sin convoluciones.
  - Abre nuevas vías de investigación en visión artificial, combinando ideas del procesamiento del lenguaje natural y la visión.

Esta es solo una selección de algunas de las arquitecturas de CNNs más influyentes. La investigación en este campo es muy activa, y continuamente se proponen nuevas arquitecturas y mejoras.

21. Entrenamiento y Visualización de Redes Convolucionales

21.1. Entrenamiento de una CNN: Repaso y Consideraciones

El objetivo del entrenamiento de una CNN es ajustar los parámetros del modelo (los pesos de los filtros convolucionales, los pesos de las conexiones en las capas totalmente conectadas, y los bias) para que la red realice la tarea deseada (clasificación, detección, segmentación, etc.) con la mayor precisión posible.

El proceso de entrenamiento implica:

1. **Definir la arquitectura de la red:** Número de capas, tipo de capas, número de filtros, tamaño de los filtros, funciones de activación, etc.
2. **Definir la función de pérdida:** Una función que mide la diferencia entre las predicciones de la red y los valores reales.
3. **Elegir un optimizador:** Un algoritmo que ajusta los parámetros de la red para minimizar la función de pérdida (por ejemplo, descenso de gradiente estocástico con momento, Adam, etc.).
4. **Preparar los datos:** Dividir el conjunto de datos en entrenamiento, validación y prueba. Preprocesar los datos (normalización, aumento de datos, etc.).
5. **Entrenar la red:** Presentar repetidamente los datos de entrenamiento a la red, calcular la pérdida, calcular los gradientes (usando retropropagación) y actualizar los parámetros utilizando el optimizador.
6. **Evaluar el rendimiento:** Evaluar periódicamente el rendimiento de la red en el conjunto de validación para monitorizar el progreso del entrenamiento, detectar el sobreajuste y ajustar los hiperparámetros.
7. **Parada temprana**: Detener el entrenamiento cuando el rendimiento en el conjunto de validación deja de mejorar.
8. **Evaluar el modelo final:** Evaluar el rendimiento del modelo entrenado en el conjunto de prueba para obtener una estimación imparcial de su capacidad de generalización.

21.2. Hiperparámetros vs. Parámetros

Es importante distinguir entre parámetros e hiperparámetros:

- **Parámetros:** Son los valores que la red aprende durante el entrenamiento (los pesos y los bias).
- **Hiperparámetros:** Son valores que se establecen antes del entrenamiento y que controlan el proceso de aprendizaje (tasa de aprendizaje, tamaño del mini-lote, número de épocas, número de capas, número de filtros, tipo de función de activación, etc.). La elección de los hiperparámetros se realiza a menudo mediante búsqueda en cuadrícula (grid search) o búsqueda aleatoria (random search), evaluando el rendimiento en el conjunto de validación.

21.3. Estrategias para Evitar el Sobreajuste

Ya hemos discutido varias estrategias para evitar el sobreajuste, incluyendo:

- Regularización L1 y L2.
- Dropout.
- Batch Normalization.
- Parada temprana.
- Aumento de datos.

21.4. Aumento de Datos (Data Augmentation)

El aumento de datos es una técnica especialmente útil en visión artificial. Consiste en aplicar transformaciones aleatorias a las imágenes del conjunto de entrenamiento para generar nuevas instancias de entrenamiento. Algunas transformaciones comunes incluyen:

- Recortes aleatorios (random cropping).
- Volteos horizontales o verticales (flipping).
- Rotaciones.
- Cambios de escala (zooming).
- Cambios de color, brillo y contraste (color shifting).
- Adición de ruido.

21.5. Tipos de Entrenamiento

- **Entrenamiento desde cero (from scratch):** Los pesos de la red se inicializan aleatoriamente y se entrenan utilizando solo los datos del problema específico. Esto requiere un conjunto de datos grande.
- **Aprendizaje por transferencia (transfer learning):** Se parte de una red pre-entrenada en un conjunto de datos grande (por ejemplo, ImageNet) y se ajustan los pesos a la nueva tarea. Esto es útil cuando se tiene un conjunto de datos pequeño, o cuando la nueva tarea es similar a la tarea para la que se entrenó la red original. Hay dos estrategias principales:
  1. **Usar la CNN pre-entrenada como un extractor de características fijo:** Se congelan los pesos de las capas convolucionales (se impide que se actualicen durante el entrenamiento) y solo se entrena un nuevo clasificador (capas totalmente conectadas) encima de ellas.
  2. **Ajuste fino (fine-tuning):** Se permite que los pesos de algunas o todas las capas de la red pre-entrenada se actualicen durante el entrenamiento. Esto permite que la red se adapte mejor a la nueva tarea. Normalmente se utiliza una tasa de aprendizaje más baja que en el entrenamiento desde cero.

21.6. Conjuntos de Datos Comunes para Visión Artificial

- **CIFAR-10/CIFAR-100:** Conjuntos de datos de imágenes pequeñas (32x32) con 10 o 100 clases.
- **MNIST/Fashion-MNIST:** Conjuntos de datos de dígitos manuscritos/prendas de ropa (28x28) con 10 clases.
- **ImageNet:** Un conjunto de datos muy grande con millones de imágenes y miles de clases. Se utiliza a menudo para pre-entrenar redes que luego se ajustan a otras tareas.
- **COCO:** Un conjunto de datos para detección de objetos, segmentación de instancias y subtitulado de imágenes (image captioning).

21.7. Visualización de Redes Convolucionales

Visualizar lo que una CNN ha aprendido, es decir, entender cómo funciona internamente, es un desafío, porque las redes profundas son 'cajas negras'. Visualizar los filtros y los mapas de activación puede proporcionar cierta información.

Visualización de filtros: Los filtros de las primeras capas convolucionales suelen aprender a detectar características de bajo nivel, como bordes, esquinas y texturas. Visualizar estos filtros puede ser útil para comprobar que el entrenamiento está funcionando correctamente (los filtros deben tener una estructura clara, sin ruido).

Visualización de mapas de características: Se puede visualizar la salida de cada capa convolucional (los mapas de características) para una imagen de entrada determinada. Esto permite ver cómo la red transforma la imagen a

medida que la información fluye a través de las capas. Las primeras capas tienden a mostrar detalles de bajo nivel, y las capas posteriores activan conceptos más complejos.

Mapas de activación de clase (Class Activation Mapping CAM): Las técnicas CAM, como Grad-CAM, permiten visualizar qué regiones de una imagen son más importantes para la decisión de la red. Generan un 'mapa de calor' que resalta las áreas de la imagen que más contribuyen a la clasificación de la imagen en una clase específica.

CAM original: requiere que la red tenga una capa de 'global average pooling' antes de la capa de salida.

Grad-CAM: Es una generalización de CAM que puede aplicarse a cualquier CNN, sin necesidad de modificar su arquitectura.

22. Redes Convolucionales para Detección de Objetos

Hasta ahora, nos hemos centrado principalmente en la clasificación de imágenes, donde el objetivo es asignar una etiqueta a una imagen completa. La detección de objetos es una tarea más compleja que implica no solo clasificar objetos dentro de una imagen, sino también localizarlos, típicamente dibujando un cuadro delimitador (bounding box) alrededor de cada objeto.

22.1. Diferencias entre Clasificación y Detección

- **Clasificación de imágenes:**
  - La imagen contiene un objeto principal (o un concepto principal).
  - El objetivo es identificar la categoría de ese objeto.
- **Detección de objetos:**
  - La imagen puede contener múltiples objetos de diferentes categorías.
  - El objetivo es identificar la categoría de cada objeto y su ubicación precisa dentro de la imagen (coordenadas del cuadro delimitador).

22.2. Componentes de un Sistema de Detección de Objetos

Un sistema de detección de objetos típico consta de los siguientes componentes:

1. **Propuesta de regiones (Region Proposal):** Identifica regiones de la imagen que podrían contener objetos (regiones de interés, o ROIs). Estas regiones son candidatas a ser analizadas en las etapas posteriores.
2. **Predicciones de la red:** Una red neuronal (típicamente una CNN) procesa cada región propuesta y produce dos tipos de predicciones:
   - **Clase:** Una distribución de probabilidad sobre las posibles categorías de objetos.
   - **Cuadro delimitador (bounding box):** Las coordenadas (x, y, ancho, alto) o (x1,y1,x2,y2) del cuadro delimitador que rodea al objeto.
3. **Supresión no máxima (Non-Maximum Suppression, NMS):** Dado que el detector puede proponer múltiples cuadros delimitadores para el mismo objeto, la supresión no máxima se utiliza para eliminar las detecciones redundantes y quedarse con la mejor detección para cada objeto.
4. **Métricas de evaluación:** Métricas para cuantificar el rendimiento del sistema.

22.3. Propuesta de Regiones

Las regiones propuestas son áreas rectangulares en la imagen que tienen una alta probabilidad de contener un objeto. Cada región propuesta se asocia con una puntuación de confianza (confidence score) que indica la probabilidad de que la región contenga un objeto (de cualquier clase).

Se utiliza un umbral de confianza para filtrar las regiones propuestas. Solo las regiones con una puntuación superior al umbral se pasan a las siguientes etapas del sistema de detección. La elección del umbral es un compromiso entre la precisión (precision) y la exhaustividad (recall).

22.4. Predicciones de la Red

Como se ha comentado, para cada región se predice tanto la clase como la bounding box.

22.5. Supresión No Máxima (Non-Maximum Suppression, NMS)

El propósito de la NMS es eliminar las detecciones duplicadas del mismo objeto. El algoritmo funciona de la siguiente manera:

1. Ordenar las detecciones por puntuación de confianza (de mayor a menor).
2. Seleccionar la detección con la puntuación más alta como una detección válida.
3. Para cada una de las detecciones restantes:
   - Calcular la Intersección sobre Unión (IoU) entre la detección actual y la detección válida seleccionada. La IoU es una medida del solapamiento entre dos cuadros delimitadores.
   - Si la IoU es mayor que un umbral predefinido (típicamente 0.5), descartar la detección actual (ya que se considera que es una detección duplicada del mismo objeto).
4. Repetir los pasos 2 y 3 hasta que se hayan procesado todas las detecciones.

22.6. Métricas de Evaluación

- **Fotogramas por segundo (Frames Per Second, FPS):** Mide la velocidad del detector (cuántas imágenes puede procesar por segundo).
- **Precisión Media (Mean Average Precision, mAP):** Es la métrica de evaluación más común para la detección de objetos. Se calcula de la siguiente manera:
  1. Para cada clase, se calcula la Precisión Promedio (Average Precision, AP). La AP se calcula como el área bajo la curva de precisión-recall (precision-recall curve).
  2. El mAP es el promedio de las APs sobre todas las clases.
- **Intersección sobre Unión (Intersection over Union, IoU):** Mide el solapamiento entre dos cuadros delimitadores. Se calcula como el área de la intersección de los dos cuadros dividida por el área de la unión de los dos cuadros.

$$\text{IoU} = \frac{\text{Área de Intersección}}{\text{Área de Unión}} = \frac{B_{gt} \cap B_p}{B_{gt} \cup B_p}$$

  Donde B_{gt} es el bounding box *ground truth* (el real) y B_p es el bounding box predicho.

Un valor de IoU de 1 indica un solapamiento perfecto, mientras que un valor de 0 indica que no hay solapamiento. Un valor de IoU superior a un cierto umbral (por ejemplo, 0.5) se considera una detección correcta (verdadero positivo).

22.7. Conjuntos de Datos para Detección de Objetos

Algunos conjuntos de datos populares para la detección de objetos incluyen:

- **PASCAL VOC:** Un conjunto de datos clásico con 20 clases de objetos.
- **MS COCO:** Un conjunto de datos a gran escala con 80 clases de objetos.
- **ImageNet:** También tiene datos para detección de objetos (un subconjunto de las imágenes de clasificación).
- **Open Images:** Un conjunto de datos muy grande con miles de clases y millones de imágenes.
- **nuScenes:** Conjunto de datos para conducción autónoma.

22.8. Métodos de Detección de Objetos

Los métodos de detección de objetos se pueden clasificar en dos grandes categorías:

1. **Métodos de dos etapas (Two-Stage Detectors):** Primero, se genera un conjunto de regiones propuestas, y luego, cada región propuesta se clasifica y se refina su cuadro delimitador. Ejemplos:
   - **R-CNN (Regions with CNN features):** Propone regiones utilizando un algoritmo de búsqueda selectiva (Selective Search), extrae características de cada región utilizando una CNN, y luego clasifica cada región utilizando un clasificador SVM (Support Vector Machine).
   - **Fast R-CNN:** Mejora R-CNN al procesar la imagen completa una sola vez con la CNN y luego extraer características de cada región propuesta a partir del mapa de características convolucional. Utiliza una capa de 'ROI pooling' para adaptar las características de cada región a un tamaño fijo.
   - **Faster R-CNN:** Introduce una Red de Propuesta de Regiones (Region Proposal Network, RPN), que es una CNN que genera regiones propuestas directamente a partir de los mapas de características convolucionales. Esto elimina la necesidad de un algoritmo de búsqueda selectiva externo y acelera significativamente el proceso de detección.
2. **Métodos de una etapa (One-Stage Detectors):** Realizan la clasificación y la localización del cuadro delimitador simultáneamente, sin una etapa separada de propuesta de regiones. Son generalmente más rápidos que los métodos de dos etapas, pero pueden ser menos precisos. Ejemplos:
   - **YOLO (You Only Look Once):** Divide la imagen en una cuadrícula y predice cuadros delimitadores y probabilidades de clase para cada celda de la cuadrícula.
   - **SSD (Single Shot MultiBox Detector):** Utiliza una serie de capas convolucionales con diferentes resoluciones para detectar objetos a diferentes escalas.

22.9. Familia R-CNN: Métodos de Dos Etapas

- **R-CNN**
  - Extrae regiones con Selective Search.
  - Extrae características con una CNN (normalmente pre-entrenada en ImageNet).
  - Clasifica cada región con un SVM y refina la bounding box con un regresor lineal.
  - Desventajas: Lento (entrenamiento en múltiples etapas, inferencia lenta).
- **Fast R-CNN**
  - Se pasa la imagen entera por la CNN una vez.
  - Se extraen las características correspondientes a cada propuesta de región del mapa de características resultante.
  - Se usa una capa *ROI Pooling* para adaptar las características a un tamaño fijo.
  - Se usa una capa softmax en lugar de SVMs. Se entrena todo *end-to-end* con una función de pérdida multi-tarea (clasificación y regresión).
  - Más rápido que R-CNN, pero sigue dependiendo de Selective Search.
- **Faster R-CNN**
  - Introduce una Red de Propuesta de Regiones (RPN), una CNN que predice regiones directamente de los mapas de características.
  - La RPN comparte los mapas de características con la red de detección, lo que reduce el coste computacional.
  - Entrenamiento end-to-end.

22.10. YOLO y SSD: Métodos de Una Etapa

YOLO (You Only Look Once)

- Divide la imagen en una cuadrícula SxS.
- Cada celda de la cuadrícula predice B bounding boxes, cada una con una puntuación de confianza y una distribución de probabilidad sobre las clases.
- Muy rápido, pero menos preciso que Faster R-CNN en objetos pequeños. Varias versiones (YOLOv1, YOLOv2, YOLOv3, YOLOv4, YOLOv5, YOLOv8, etc.). YOLOv3: Introduce predicciones a múltiples escalas para mejorar la detección de objetos pequeños.

SSD (Single Shot Detector)

- Utiliza múltiples capas convolucionales con diferentes resoluciones para detectar objetos a diferentes escalas.
- Cada capa convolucional predice bounding boxes y probabilidades de clase para un conjunto de 'cajas ancla' (anchor boxes) predefinidas.
- Similar a YOLO en velocidad, pero generalmente más preciso.

Tanto YOLO como SSD son significativamente más rápidos que los métodos de la familia R-CNN, lo que los hace adecuados para aplicaciones en tiempo real.

23. Redes Convolucionales para Segmentación Semántica

La segmentación semántica es una tarea de visión artificial que consiste en asignar a cada píxel de una imagen una etiqueta de clase semántica. A diferencia de la clasificación de imágenes, que asigna una única etiqueta a toda la imagen, la segmentación semántica produce una máscara de segmentación, donde cada píxel está clasificado. A diferencia de la detección de objetos, no se buscan instancias individuales, sino regiones con una misma etiqueta semántica.

23.1. Conceptos Clave

- **Segmentación vs. Clasificación vs. Detección:**
  - *Clasificación:* Asigna una etiqueta a toda la imagen.
  - *Detección de objetos:* Localiza y clasifica objetos individuales dentro de la imagen (mediante cuadros delimitadores).
  - *Segmentación semántica:* Asigna una etiqueta de clase a cada píxel de la imagen, sin distinguir entre instancias individuales del mismo objeto.
  - *Segmentación de instancias*: Asigna una etiqueta a cada pixel, *distinguiendo* instancias.
  - *Segmentación Panóptica*: Combinación de segmentación semántica y de instancias. Se etiqueta el fondo (stuff) y cada instancia.
- **Things vs. Stuff**
  - *Things (Cosas)*: Objectos contables, con una forma definida.
  - *Stuff (Material)*: Regiones amorfas, sin una forma definida.
- **Predicciones de la red:** La salida de una red de segmentación semántica es un mapa de segmentación (segmentation map) con las mismas dimensiones espaciales que la imagen de entrada. Cada píxel en el mapa de segmentación contiene la etiqueta de clase predicha para ese píxel. Si hay N clases, la salida tendrá N canales.
- **Entrenamiento:** Para entrenar una red de segmentación semántica, se necesita un conjunto de datos de imágenes con máscaras de segmentación etiquetadas a nivel de píxel. Cada píxel en la máscara de segmentación tiene una etiqueta que indica la clase a la que pertenece.

23.2. Arquitecturas para Segmentación Semántica

23.2.1. Redes Completamente Convolucionales (Fully Convolutional Networks - FCNs)

Una de las primeras arquitecturas exitosas para segmentación semántica fue la red completamente convolucional (FCN). Una FCN se caracteriza por:

- **Solo capas convolucionales:** No utiliza capas totalmente conectadas (excepto, posiblemente, una capa final 1x1 para generar las predicciones por píxel). Esto permite que la red acepte imágenes de entrada de cualquier tamaño.
- **Downsampling y upsampling:** La red típicamente tiene una fase de 'downsampling' (reducción de la resolución espacial) mediante capas convolucionales con stride ¿1 o capas de pooling, seguida de una fase de 'upsampling' (aumento de la resolución espacial) para generar una máscara de segmentación del mismo tamaño que la imagen de entrada.
- **Entrenamiento extremo a extremo (end-to-end):** La red se entrena completa, desde la entrada (imagen) hasta la salida (máscara de segmentación), utilizando retropropagación.

23.2.2. Operaciones de Upsampling

Para aumentar la resolución espacial de los mapas de características, se utilizan operaciones de upsampling. Algunas de las operaciones de upsampling más comunes son:

- **Unpooling (desagrupamiento):** Es la operación inversa al pooling. Existen diferentes variantes:
  - *Nearest neighbor unpooling:* Simplemente replica cada valor en un bloque de píxeles.
  - *Bed of nails unpooling:* Coloca el valor original en una posición fija dentro del bloque de salida y rellena el resto con ceros.
  - *Max unpooling:* Recuerda las posiciones de los valores máximos durante la operación de max pooling (en la fase de downsampling) y utiliza estas posiciones para colocar los valores en la fase de upsampling. Requiere almacenar información adicional durante el max pooling.
- **Convolución transpuesta (transposed convolution), también llamada deconvolución (aunque este término es controvertido) o convolución fraccionalmente estratificada (fractionally strided convolution):** Es una operación que realiza una convolución 'inversa', aumentando la resolución espacial de la entrada. A diferencia del unpooling, la convolución transpuesta tiene parámetros aprendibles (los pesos del filtro).

23.2.3. Arquitecturas Codificador-Decodificador

Muchas arquitecturas de segmentación semántica siguen un patrón de codificador-decodificador:

- **Codificador (Encoder):** Extrae características de la imagen de entrada, reduciendo progresivamente la resolución espacial (downsampling). Típicamente, es una CNN similar a las utilizadas para clasificación de imágenes (por ejemplo, VGG, ResNet), pero sin las capas totalmente conectadas finales.
- **Decodificador (Decoder):** Aumenta progresivamente la resolución espacial de los mapas de características (upsampling) y genera la máscara de segmentación final. Utiliza operaciones de upsampling (unpooling o convolución transpuesta).

Algunas arquitecturas populares de tipo codificador-decodificador incluyen:

- **FCN:** Utiliza convoluciones transpuestas para el upsampling.
- **SegNet:** Utiliza max unpooling, recordando las posiciones de los máximos durante el max pooling en el codificador.

- **U-Net:** Introduce conexiones de salto (skip connections) que conectan las salidas de las capas del codificador con las entradas de las capas correspondientes del decodificador. Esto permite que el decodificador recupere información de bajo nivel que se pierde durante el downsampling.
- **DeepLab:** Utiliza convoluciones dilatadas (atrous convolutions) para aumentar el campo receptivo de los filtros sin reducir la resolución espacial. También utiliza un módulo llamado *Atrous Spatial Pyramid Pooling (ASPP)* que aplica convoluciones dilatadas con diferentes tasas de dilatación en paralelo. Las versiones más recientes (DeepLabv3, DeepLabv3+) utilizan una arquitectura codificador-decodificador.

23.3. Métricas de Evaluación

- **Pixel Accuracy:** El porcentaje de píxeles correctamente clasificados. Puede ser engañosa si las clases están desbalanceadas.
- **Intersection over Union (IoU), también llamado índice de Jaccard:** Mide el solapamiento entre la máscara de segmentación predicha y la máscara de segmentación real (ground truth). Se calcula para cada clase y luego se promedia.

$$\text{IoU} = \frac{\text{Área de Intersección}}{\text{Área de Unión}}$$

- **Coeficiente Dice (Dice coefficient), también llamado F1-score:** Similar al IoU, pero con una ponderación diferente.

$$\text{Dice} = \frac{2 \cdot \text{Área de Intersección}}{\text{Área Total}} = \frac{2TP}{2TP + FP + FN}$$

Donde TP son verdaderos positivos, FP falsos positivos y FN falsos negativos.

23.4. Funciones de Pérdida

Las funciones de pérdida para la segmentación semántica se calculan a nivel de píxel. Algunas funciones de pérdida comunes son:

- **Cross-entropy loss (pérdida de entropía cruzada):** Se aplica la entropía cruzada a cada píxel individualmente y luego se promedia sobre todos los píxeles.
- **Dice loss:** Se basa en el coeficiente Dice. Es una alternativa a la cross-entropy loss, especialmente útil cuando las clases están desbalanceadas.
- **IoU loss:** Basada en el IoU.

23.5. Conjuntos de Datos

Algunos conjuntos de datos populares para la segmentación semántica incluyen:

- **PASCAL VOC**
- **MS COCO**
- **Cityscapes** (imágenes de escenas urbanas)
- **ADE20K** (escenas de interior y exterior)

24. Segmentación de Instancias y Segmentación Panóptica

Hemos visto la segmentación semántica, que asigna una etiqueta de clase a cada píxel de una imagen, sin diferenciar entre instancias individuales de la misma clase. Ahora, exploraremos dos extensiones de la segmentación semántica: la segmentación de instancias y la segmentación panóptica.

24.1. Segmentación de Instancias

La segmentación de instancias va un paso más allá de la segmentación semántica. No solo clasifica cada píxel en una categoría, sino que también distingue entre diferentes instancias de la misma categoría. Por ejemplo, en una imagen con varias personas, la segmentación semántica etiquetaría todos los píxeles correspondientes a personas como 'persona', mientras que la segmentación de instancias asignaría una etiqueta única a cada persona individual.

- Clasificación a nivel de píxel.
- Distingue entre diferentes instancias de la misma clase.
- Asigna un identificador único a cada instancia.
- Útil para tareas como el seguimiento de objetos, la manipulación de imágenes y el conteo de objetos.

24.1.1. Mask R-CNN

Una de las arquitecturas más populares para la segmentación de instancias es Mask R-CNN.
Mask R-CNN es una extensión de Faster R-CNN (un detector de objetos de dos etapas).

1. **Backbone network (red troncal):** Una CNN (por ejemplo, ResNet) extrae características de la imagen de entrada.
2. **Region Proposal Network (RPN):** Propone regiones de interés (ROIs) que podrían contener objetos.
3. **ROIAlign:** Para cada ROI, extrae un mapa de características de tamaño fijo del mapa de características generado por la red troncal. A diferencia del ROI pooling utilizado en Faster R-CNN, ROIAlign utiliza interpolación bilineal para evitar la pérdida de información debida a la cuantificación de las coordenadas de la ROI. Esto es crucial para la precisión de la segmentación a nivel de píxel.
4. **Ramas paralelas:**
   - *Rama de clasificación:* Predice la clase del objeto en la ROI.
   - *Rama de regresión de cuadro delimitador:* Refina las coordenadas del cuadro delimitador de la ROI.
   - *Rama de máscara:* Predice una máscara de segmentación binaria para cada ROI. Esta máscara indica qué píxeles dentro de la ROI pertenecen al objeto. La máscara es de baja resolución (ej. 28x28) pero al ser binaria (solo dos valores posibles para cada píxel), la cantidad de parámetros es manejable.

- Es una extensión de Faster R-CNN, añadiendo una rama para la predicción de máscaras.
- Utiliza ROIAlign para una extracción precisa de características de las ROIs.
- Predice máscaras binarias para cada clase, sin competencia entre clases. La clase predicha por la rama de clasificación determina qué máscara se utiliza.
- La función de pérdida es una combinación de las pérdidas de clasificación, regresión de cuadro delimitador y máscara.
- Obtuvo excelentes resultados en el desafío COCO de segmentación de instancias.

24.2. Segmentación Panóptica

La segmentación panóptica combina la segmentación semántica y la segmentación de instancias en una única tarea. El objetivo es asignar a cada píxel de la imagen una etiqueta de clase semántica y, si el píxel pertenece a una instancia de objeto ("thing"), un identificador de instancia.
La segmentación panóptica asigna a cada píxel i de una imagen un par (l_i, z_i), donde:

- $l_i \in \{1, \ldots, K\}$ es la etiqueta de clase semántica (K es el número total de clases).
- $z_i \in \mathbb{N}$ es un identificador de instancia.

Para los píxeles que pertenecen a clases de 'stuff' (fondo, como 'cielo', 'carretera'), z_i es irrelevante. Para los píxeles que pertenecen a clases de things (objetos contables), z_i identifica la instancia específica del objeto.

Diferencias con la segmentación semántica y de instancias:

- La segmentación semántica solo asigna l_i a cada píxel.
- La segmentación de instancias solo se preocupa por los píxeles que pertenecen a things y asigna un identificador de instancia, pero no una etiqueta de clase semántica para los píxeles de stuff.
- La segmentación panóptica proporciona una descripción completa de la imagen, combinando la información de la segmentación semántica y de instancias.

Enfoques para la segmentación panóptica:

- Un enfoque básico consiste en combinar las salidas de un modelo de segmentación semántica y un modelo de segmentación de instancias. Sin embargo, este enfoque puede ser ineficiente y difícil de optimizar.
- Arquitecturas más avanzadas, como Panoptic FPN (Feature Pyramid Network), utilizan una única red con dos ramas: una rama para la segmentación semántica y otra para la segmentación de instancias. Estas ramas comparten características extraídas por una red troncal común (backbone).

Panoptic Feature Pyramid Network (PFPN): Un ejemplo de arquitectura para la segmentación panóptica. Combina una FPN (para extraer características a múltiples escalas) con una rama para segmentación semántica (similar a una FCN) y una rama para segmentación de instancias (similar a Mask R-CNN).

24.3. Métricas de Evaluación

- **Segmentación semántica:** IoU (Intersection over Union) promediado sobre todas las clases.
- **Segmentación de instancias:** mAP (mean Average Precision), calculado a diferentes umbrales de IoU (por ejemplo, mAP@0.5, mAP@0.75).
- **Segmentación panóptica:**
  - **Panoptic Quality (PQ):** Es la métrica principal para la segmentación panóptica. Combina la calidad de la segmentación (qué tan bien se ajustan las máscaras predichas a las máscaras reales) y la calidad del reconocimiento (qué tan bien se clasifican los objetos).

$$\text{PQ} = \frac{\sum_{(p,g)\in TP} \text{IoU}(p,g)}{|TP| + \frac{1}{2}|FP| + \frac{1}{2}|FN|}$$

Donde:

- TP son los verdaderos positivos (predicciones correctas).
- FP son los falsos positivos (predicciones incorrectas).
- FN son los falsos negativos (objetos reales no detectados).
- $\text{IoU}(p,g)$ es la Intersección sobre Unión entre la máscara predicha p y la máscara real g.

El PQ se calcula por separado para 'things' y 'stuff', y luego se puede promediar para obtener un PQ global.

24.4. Conjuntos de Datos

Los conjuntos de datos para la segmentación de instancias y panóptica, además de las anotaciones de segmentación semántica, deben incluir información adicional:

- **Identificadores de instancia:** Para cada objeto ('thing'), un identificador único.
- **Máscaras de instancia:** Para cada objeto, una máscara binaria que indica qué píxeles pertenecen a esa instancia.

Conjuntos como COCO, Cityscapes y ADE20K tienen versiones con anotaciones para la segmentación de instancias y/o panóptica.

24.5. Modelos Avanzados: OneFormer

OneFormer es un ejemplo reciente de modelo que unifica las tareas de segmentación semántica, de instancias y panóptica en una sola arquitectura, utilizando un transformer.

25. Redes Neuronales Recurrentes (RNNs): Fundamentos y Arquitecturas

25.1. Introducción

Hasta ahora, hemos estudiado redes neuronales feedforward (como las redes multicapa y las CNNs), donde la información fluye en una sola dirección, desde la entrada hasta la salida. Las redes neuronales recurrentes (RNNs) introducen un concepto fundamentalmente diferente: conexiones recurrentes. Estas conexiones permiten que la información persista a lo largo del tiempo, creando una especie de memoria interna en la red.

25.2. ¿Qué son las RNNs?

Las RNNs son redes neuronales profundas que contienen bucles o ciclos en su arquitectura. Estos bucles permiten que la información de pasos anteriores influya en el procesamiento de los pasos actuales. En otras palabras, la salida de la red en un instante de tiempo t no solo depende de la entrada en ese instante, sino también del estado interno de la red, que ha sido influenciado por las entradas anteriores.

25.3. ¿Para qué sirven las RNNs?

Las RNNs son especialmente adecuadas para procesar datos secuenciales, donde el orden de los elementos es importante. Algunos ejemplos de datos secuenciales y tareas donde las RNNs son útiles:

- **Texto:** Análisis de sentimiento, traducción automática, generación de texto, etiquetado de partes de la oración (POS tagging), reconocimiento de entidades nombradas (NER).
- **Audio:** Reconocimiento del habla, generación de música, clasificación de audio.
- **Video:** Clasificación de video, descripción de video (video captioning), seguimiento de objetos.
- **Series temporales:** Predicción del precio de acciones, predicción meteorológica, análisis de señales biomédicas.
- **ADN/ARN/Proteínas:** Análisis de secuencias biológicas.
- **Cualquier dato donde el orden importe.**

25.4. Formulación Matemática de una RNN Simple

Una RNN simple (a veces llamada RNN de Elman) tiene la siguiente formulación:

$$\mathbf{h}_t = g_1(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{b}_h)$$
$$\mathbf{y}_t = g_2(\mathbf{V}\mathbf{h}_t + \mathbf{b}_y)$$

Donde:

- $\mathbf{x}_t$ es el vector de entrada en el instante de tiempo t.
- $\mathbf{h}_t$ es el estado oculto (hidden state) en el instante de tiempo t. El estado oculto representa la "memoria"de la red.
- $\mathbf{y}_t$ es el vector de salida en el instante de tiempo t.

- $\mathbf{W}$ es la matriz de pesos de las conexiones recurrentes (del estado oculto anterior al estado oculto actual).
- $\mathbf{U}$ es la matriz de pesos de las conexiones de la entrada al estado oculto.
- $\mathbf{V}$ es la matriz de pesos de las conexiones del estado oculto a la salida.
- $\mathbf{b}_h$ es el vector de bias para el estado oculto.
- $\mathbf{b}_y$ es el vector de bias para la salida.
- g_1 es la función de activación para el estado oculto (típicamente, tanh o ReLU).
- g_2 es la función de activación para la salida (depende de la tarea; por ejemplo, softmax para clasificación, sigmoide para clasificación binaria, o ninguna función para regresión).
- La misma matriz de pesos ($\mathbf{W}$, $\mathbf{U}$, $\mathbf{V}$) y los mismos vectores de bias ($\mathbf{b}_h$, $\mathbf{b}_y$) se utilizan en todos los pasos de tiempo. Esto reduce drásticamente el número de parámetros en comparación con una red feedforward que trataría cada paso de tiempo como una entrada independiente.
- El estado oculto $\mathbf{h}_t$ se calcula en función del estado oculto anterior $\mathbf{h}_{t-1}$ y de la entrada actual $\mathbf{x}_t$. Esto permite que la red recuerde información de pasos anteriores.
- La salida $\mathbf{y}_t$ se calcula en función del estado oculto actual $\mathbf{h}_t$.

25.5. Despliegue en el Tiempo (Unfolding)

Para entender mejor cómo funciona una RNN, es útil desplegarla en el tiempo. Esto significa representar la red como una secuencia de copias de sí misma, una para cada paso de tiempo. Cada copia recibe la entrada correspondiente a ese paso de tiempo y el estado oculto de la copia anterior.

25.6. Entrenamiento de una RNN Backpropagation Through Time (BPTT)

El entrenamiento de una RNN se realiza utilizando una variante del algoritmo de retropropagación (backpropagation) llamada Backpropagation Through Time (BPTT).

Pasos de BPTT:

1. **Forward pass:** Se presenta una secuencia de entrada a la red y se calcula la salida para cada paso de tiempo, 'desplegando' la red en el tiempo. Se almacenan los estados ocultos y las salidas en cada paso.
2. **Cálculo de la pérdida:** Se calcula la pérdida total de la secuencia, que es la suma de las pérdidas en cada paso de tiempo.
3. **Backward pass:** Se calcula el gradiente de la pérdida con respecto a los parámetros de la red (W, U, V, bh, by), propagando el error hacia atrás en el tiempo, desde el último paso hasta el primero. La regla de la cadena se aplica repetidamente.
4. **Actualización de parámetros:** Se actualizan los parámetros utilizando un algoritmo de optimización (por ejemplo, descenso de gradiente estocástico).

Dado que las mismas matrices de pesos se utilizan en todos los pasos de tiempo, el gradiente total con respecto a cada matriz de pesos es la suma de los gradientes calculados en cada paso de tiempo.

25.7. Desafíos del Entrenamiento de RNNs: Desvanecimiento y Explosión del Gradiente

Al igual que otras redes profundas, las RNNs pueden sufrir los problemas de desvanecimiento y explosión de gradientes, y se acentúan por la recurrencia.

- Si la secuencia de entrada tiene longitud n, una RNN se comporta de igual forma que una red feed-forward con n capas.

- Si la función de activación tiene valores menores que 1, el gradiente se desvanece.
- Si la función de activación es mayor que 1, el gradiente explota.

25.8. Tipos de Arquitecturas RNN

Existen diferentes formas de conectar la entrada, el estado oculto y la salida en una RNN, lo que da lugar a diferentes arquitecturas. Algunos ejemplos comunes:

- **Muchos a uno (Many-to-one):** Una secuencia de entrada produce una única salida. Ejemplo: análisis de sentimiento (una secuencia de palabras se clasifica como positiva o negativa).
- **Uno a muchos (One-to-many):** Una única entrada produce una secuencia de salida. Ejemplo: generación de música (una nota inicial genera una secuencia de notas).
- **Muchos a muchos (Many-to-many):** Una secuencia de entrada produce una secuencia de salida. Ejemplos:
  - *Sincronizado (Synced):* La salida en cada paso de tiempo corresponde a la entrada en ese mismo paso. Ejemplo: etiquetado de partes de la oración (POS tagging).
  - *Asíncrono (Unsynced):* La secuencia de salida puede tener una longitud diferente a la secuencia de entrada. Ejemplo: traducción automática.

26. Redes Recurrentes: GRU, LSTM y Bidireccionales

En la sección anterior, introdujimos las redes neuronales recurrentes (RNNs) básicas y sus fundamentos. Aunque las RNNs simples son poderosas en teoría, en la práctica sufren del problema del desvanecimiento del gradiente, lo que dificulta el aprendizaje de dependencias a largo plazo en las secuencias. Para abordar este problema, se han desarrollado arquitecturas más sofisticadas, como las GRUs y las LSTMs. Además, veremos las redes bidireccionales.

26.1. Gated Recurrent Unit (GRU)

La Gated Recurrent Unit (GRU) es una variante de la RNN que introduce compuertas (gates) para controlar el flujo de información a través de la red. Estas compuertas permiten a la red decidir qué información del pasado debe conservarse y qué información debe olvidarse. Las GRUs son más simples que las LSTMs (que veremos más adelante), pero a menudo ofrecen un rendimiento comparable.

26.1.1. Componentes de una GRU

Una GRU tiene dos compuertas:

- **Reset gate ($\mathbf{r}_t$):** Determina cuánta información del estado oculto anterior ($\mathbf{h}_{t-1}$) debe olvidarse.
- **Update gate ($\mathbf{z}_t$):** Determina cuánta información del estado oculto anterior debe conservarse y cuánta información nueva debe añadirse. (En las imágenes de las diapositivas proporcionadas, la compuerta de actualización se representa como Γ_u y la compuerta reset como Γ_r).

Además de las compuertas, una GRU tiene un estado oculto ($\mathbf{h}_t$) que se actualiza en cada paso de tiempo.

26.1.2. Ecuaciones de una GRU

Las ecuaciones que definen el funcionamiento de una GRU son las siguientes:

$$
\begin{aligned}
\mathbf{z}_t &= \sigma(\mathbf{W}_z \mathbf{x}_t + \mathbf{U}_z \mathbf{h}_{t-1} + \mathbf{b}_z) \\
\mathbf{r}_t &= \sigma(\mathbf{W}_r \mathbf{x}_t + \mathbf{U}_r \mathbf{h}_{t-1} + \mathbf{b}_r) \\
\tilde{\mathbf{h}}_t &= \tanh(\mathbf{W} \mathbf{x}_t + \mathbf{U}(\mathbf{r}_t \odot \mathbf{h}_{t-1}) + \mathbf{b}_h) \\
\mathbf{h}_t &= (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t
\end{aligned}
$$

Donde:

- $\mathbf{x}_t$ es el vector de entrada en el instante de tiempo t.
- $\mathbf{h}_t$ es el estado oculto en el instante de tiempo t.
- $\mathbf{z}_t$ es el vector de la compuerta de actualización (update gate).
- $\mathbf{r}_t$ es el vector de la compuerta de reset (reset gate).
- $\tilde{\mathbf{h}}_t$ es el vector de estado oculto candidato.
- $\mathbf{W}_z$, $\mathbf{W}_r$, $\mathbf{W}$, $\mathbf{U}_z$, $\mathbf{U}_r$, $\mathbf{U}$ son matrices de pesos.
- $\mathbf{b}_z$, $\mathbf{b}_r$, $\mathbf{b}_h$ son vectores de bias.
- σ es la función sigmoide.
- tanh es la función tangente hiperbólica.
- $\odot$ denota la multiplicación elemento a elemento (producto de Hadamard).

Explicación de las ecuaciones

1. **Compuerta de actualización ($\mathbf{z}_t$):** Calcula un valor entre 0 y 1 para cada neurona, utilizando una función sigmoide. Este valor determina cuánto del estado oculto anterior se conservará y cuánto del nuevo estado oculto candidato se incorporará.
2. **Compuerta de reset ($\mathbf{r}_t$):** Calcula un valor entre 0 y 1 para cada neurona. Este valor determina cuánto del estado oculto anterior se olvidará.
3. **Estado oculto candidato ($\tilde{\mathbf{h}}_t$):** Calcula un nuevo estado oculto candidato, utilizando una función tangente hiperbólica. La compuerta de reset modula la influencia del estado oculto anterior en el cálculo del estado oculto candidato.
4. **Estado oculto ($\mathbf{h}_t$):** Calcula el nuevo estado oculto como una combinación lineal del estado oculto anterior ($\mathbf{h}_{t-1}$) y el estado oculto candidato ($\tilde{\mathbf{h}}_t$). La compuerta de actualización controla la ponderación de cada uno.

26.2. Long Short-Term Memory (LSTM)

La Long Short-Term Memory (LSTM) es otra variante de la RNN, aún más potente que la GRU. Las LSTMs fueron diseñadas específicamente para abordar el problema del desvanecimiento del gradiente y aprender dependencias a largo plazo.

26.2.1. Componentes de una LSTM

Una LSTM tiene tres compuertas:

- **Forget gate ($\mathbf{f}_t$):** Decide qué información se va a olvidar del estado de la celda (cell state).
- **Input gate ($\mathbf{i}_t$):** Decide qué información nueva se va a almacenar en el estado de la celda.
- **Output gate ($\mathbf{o}_t$):** Decide qué parte del estado de la celda se va a utilizar para calcular la salida de la LSTM.

Además de las compuertas, una LSTM tiene dos estados:

- **Estado de la celda ($\mathbf{c}_t$):** Es el principal mecanismo de memoria de la LSTM. La información fluye a través del estado de la celda a lo largo del tiempo, con modificaciones controladas por las compuertas.
- **Estado oculto ($\mathbf{h}_t$):** Es la salida de la LSTM en cada paso de tiempo. Se calcula a partir del estado de la celda y la compuerta de salida.

En las imágenes proporcionadas, la compuerta de olvido se representa como Γ_f, la de entrada como Γ_u y la de salida como Γ_o

26.2.2. Ecuaciones de una LSTM

Las ecuaciones que definen el funcionamiento de una LSTM son las siguientes:

$$
\begin{aligned}
\mathbf{f}_t &= \sigma(\mathbf{W}_f \mathbf{x}_t + \mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{b}_f) \\
\mathbf{i}_t &= \sigma(\mathbf{W}_i \mathbf{x}_t + \mathbf{U}_i \mathbf{h}_{t-1} + \mathbf{b}_i) \\
\mathbf{o}_t &= \sigma(\mathbf{W}_o \mathbf{x}_t + \mathbf{U}_o \mathbf{h}_{t-1} + \mathbf{b}_o) \\
\tilde{\mathbf{c}}_t &= \tanh(\mathbf{W}_c \mathbf{x}_t + \mathbf{U}_c \mathbf{h}_{t-1} + \mathbf{b}_c) \\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t \\
\mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t)
\end{aligned}
$$

Donde:

- $\mathbf{f}_t$, $\mathbf{i}_t$ y $\mathbf{o}_t$ son los vectores para las compuertas de olvido, entrada y salida

Y siendo el resto de variables análogas a las de la GRU.

1. **Compuerta de olvido ($\mathbf{f}_t$):** Se calcula usando una función sigmoide, por lo que sus valores están entre 0 y 1.
2. **Compuerta de entrada ($\mathbf{i}_t$):** Se calcula usando una función sigmoide.
3. **Compuerta de salida ($\mathbf{o}_t$):** Se calcula usando una función sigmoide.
4. **Estado de celda candidato ($\tilde{\mathbf{c}}_t$):** Se calcula un nuevo estado de celda candidato, utilizando una función tangente hiperbólica.
5. **Estado de celda ($\mathbf{c}_t$):** El nuevo estado de la celda se calcula como una combinación del estado de la celda anterior ($\mathbf{c}_{t-1}$) y el estado de celda candidato ($\tilde{\mathbf{c}}_t$). La compuerta de olvido controla cuánto del estado anterior se olvida, y la compuerta de entrada controla cuánto del nuevo estado candidato se añade.
6. **Estado oculto ($\mathbf{h}_t$):** La salida de la LSTM se calcula aplicando la tangente hiperbólica al estado de la celda y multiplicando el resultado por la compuerta de salida.

26.3. RNNs Bidireccionales (Bidirectional RNNs - BRNNs)

Las RNNs que hemos visto hasta ahora procesan la secuencia de entrada en una sola dirección (normalmente, de izquierda a derecha). Las RNNs bidireccionales procesan la secuencia en ambas direcciones: de izquierda a derecha y de derecha a izquierda.

Una BRNN consta de dos RNNs:

- Una RNN forward que procesa la secuencia en el orden original.
- Una RNN backward que procesa la secuencia en el orden inverso.

En cada paso de tiempo, la salida de la BRNN se calcula combinando las salidas de la RNN forward y la RNN backward (normalmente, concatenándolas o sumándolas).

Permiten que la red tenga en cuenta tanto el contexto pasado como el contexto futuro de cada elemento de la secuencia. Esto puede ser muy útil en tareas donde el contexto futuro es importante, como el análisis de sentimiento o el etiquetado de partes de la oración. Por ejemplo, para saber si "banco"se refiere a una institución financiera o a un asiento, es necesario mirar el contexto tanto anterior como posterior.

27. Transformers y Mecanismos de Atención

Los Transformers son una arquitectura de red neuronal relativamente reciente (introducida en 2017 en el artículo "Attention is All You Need") que ha revolucionado el campo del procesamiento del lenguaje natural (PLN) y se está aplicando cada vez más en otras áreas, como la visión artificial. A diferencia de las RNNs, los Transformers no se basan en la recurrencia, sino en un mecanismo llamado atención (attention).

27.1. Motivación: Limitaciones de las RNNs en Secuencias Largas

Las RNNs, aunque potentes para modelar secuencias, tienen algunas limitaciones:

- **Procesamiento secuencial:** Las RNNs procesan las secuencias de entrada de forma secuencial, un elemento a la vez. Esto dificulta la paralelización del entrenamiento y puede hacer que el procesamiento de secuencias largas sea lento.
- **Memoria a corto plazo:** Aunque las LSTMs y GRUs mejoran la capacidad de las RNNs para recordar información a largo plazo, todavía pueden tener dificultades para capturar dependencias entre elementos muy distantes en la secuencia (problema del desvanecimiento del gradiente).
- **Cuello de botella de información:** En tareas de secuencia a secuencia (como la traducción automática), la RNN típicamente codifica toda la secuencia de entrada en un único vector de estado oculto, que luego se utiliza para generar la secuencia de salida. Este vector puede convertirse en un cuello de botella de información, especialmente para secuencias largas.
- **No toda la entrada es relevante:** En muchas tareas, no todos los elementos de la entrada son igual de relevantes para predecir la salida. Una RNN no tiene forma de .enfocarse.en la parte relevante de la secuencia.

Los Transformers abordan estas limitaciones mediante el uso de mecanismos de atención.

27.2. Mecanismos de Atención (Attention Mechanisms)

La idea central de los mecanismos de atención es permitir que la red se enfoque en diferentes partes de la secuencia de entrada al generar cada elemento de la secuencia de salida. En lugar de tratar de comprimir toda la información de la entrada en un único vector de estado oculto, la atención permite a la red acceder directamente a todos los elementos de la entrada y ponderar su importancia en función de la tarea.

Definición general: Un mecanismo de atención toma como entrada un conjunto de vectores (que pueden representar palabras, características de una imagen, etc.) y produce una salida ponderada de estos vectores. Los pesos (llamados pesos de atención) indican la importancia de cada vector de entrada para la salida actual.

27.3. Self-Attention (Auto-Atención)

El tipo de atención utilizado en los Transformers se llama *self-attention* (o *intra-attention*). En self-attention, la red calcula los pesos de atención entre los diferentes elementos de la *misma* secuencia. Esto permite a la red modelar las relaciones entre los diferentes elementos de la secuencia, independientemente de su distancia.

27.3.1. Componentes de Self-Attention

El mecanismo de self-attention opera sobre tres conjuntos de vectores:

- **Queries (Q):** Representan la información que estamos 'buscando'. En el contexto de una secuencia de palabras, cada query podría representar una palabra para la cual queremos calcular la atención sobre las otras palabras.
- **Keys (K):** Representan la información que estamos 'comparando' con las queries. En el contexto de una secuencia de palabras, cada key podría representar una palabra que estamos comparando con la palabra representada por la query.
- **Values (V):** Representan la información que vamos a 'extraer' en función de la atención calculada. En el contexto de una secuencia de palabras, cada value podría representar la información asociada a una palabra.

En la forma más básica de self-attention (sin las mejoras que se utilizan en los Transformers), las queries, keys y values son simplemente los vectores de entrada originales (por ejemplo, los embeddings de las palabras). Sin embargo, es mucho mas habitual que sean proyecciones de las entradas.

27.3.2. Cálculo de la Atención

Los pasos para calcular la self-attention son los siguientes:

1. **Calcular los scores de atención:** Se calcula un score de atención entre cada query y cada key. El score indica la compatibilidad o similitud entre la query y la key. Una forma común de calcular los scores es utilizando el producto escalar (dot product):

$$\text{score}(\mathbf{q}_i, \mathbf{k}_j) = \mathbf{q}_i^T \mathbf{k}_j$$

   Donde $\mathbf{q}_i$ es la query i y $\mathbf{k}_j$ es la key j.

2. **Normalizar los scores (softmax):** Los scores se normalizan utilizando una función softmax para obtener los pesos de atención. La softmax asegura que los pesos sean positivos y sumen 1:

$$\alpha_{ij} = \text{softmax}(\text{score}(\mathbf{q}_i, \mathbf{k}_j)) = \frac{\exp(\text{score}(\mathbf{q}_i, \mathbf{k}_j))}{\sum_{j'} \exp(\text{score}(\mathbf{q}_i, \mathbf{k}_{j'}))}$$

   Donde α_{ij} es el peso de atención entre la query i y la key j.

3. **Calcular la salida ponderada:** La salida de la capa de self-attention para cada query se calcula como una suma ponderada de los values, utilizando los pesos de atención:

$$\mathbf{y}_i = \sum_j \alpha_{ij} \mathbf{v}_j$$

   Donde $\mathbf{y}_i$ es la salida para la query i y $\mathbf{v}_j$ es el value j.

27.4. Mejoras al Self-Attention Básico (Utilizadas en los Transformers)

Los Transformers introducen varias mejoras al mecanismo básico de self-attention:

1. **Queries, Keys y Values aprendibles:** En lugar de usar directamente los vectores de entrada como queries, keys y values, se aprenden transformaciones lineales de los vectores de entrada para obtener las queries, keys y values:

$$\begin{aligned} \mathbf{q}_i &= \mathbf{W}_q \mathbf{x}_i \\ \mathbf{k}_i &= \mathbf{W}_k \mathbf{x}_i \\ \mathbf{v}_i &= \mathbf{W}_v \mathbf{x}_i \end{aligned}$$

   Donde $\mathbf{W}_q$, $\mathbf{W}_k$ y $\mathbf{W}_v$ son matrices de pesos entrenables.

2. **Escalado del producto escalar:** Para evitar que los productos escalares se vuelvan demasiado grandes (lo que puede causar problemas con la softmax), se divide el producto escalar por la raíz cuadrada de la dimensión de las keys ($\sqrt{d_k}$, donde d_k es la dimensión de las keys):

$$\text{score}(\mathbf{q}_i, \mathbf{k}_j) = \frac{\mathbf{q}_i^T \mathbf{k}_j}{\sqrt{d_k}}$$

3. **Multi-Head Attention:** En lugar de realizar una única operación de self-attention, se realizan múltiples operaciones de self-attention en paralelo, con diferentes matrices de pesos aprendibles ($\mathbf{W}_q$, $\mathbf{W}_k$, $\mathbf{W}_v$). Esto permite a la red atender a diferentes aspectos de la información en paralelo. Las salidas de las diferentes 'cabezas' de atención se concatenan y se proyectan linealmente a la dimensión deseada.

27.5. Cross-Attention

La cross-attention es muy similar a la self-attention. La diferencia fundamental es que se aplica sobre *dos secuencias distintas*. Es decir, las queries se obtienen a partir de una secuencia, mientras que las keys y los values se obtienen de otra.

27.6. Resumen de Self-Attention (con mejoras)

La ecuación completa para el self-attention con las mejoras mencionadas (sin multi-head) es:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

Donde:

- $\mathbf{Q}$ es la matriz de queries (cada fila es una query).
- $\mathbf{K}$ es la matriz de keys (cada fila es una key).
- $\mathbf{V}$ es la matriz de values (cada fila es un value).
- d_k es la dimensión de las keys.

En la siguiente sección, veremos cómo se utiliza el mecanismo de self-attention dentro de la arquitectura Transformer.

28. Transformers: Arquitectura y Aplicaciones

Después de haber introducido el concepto de atención y self-attention, estamos listos para abordar la arquitectura Transformer, que ha revolucionado el procesamiento del lenguaje natural y se está extendiendo a otras áreas.

28.1. Arquitectura del Transformer

El Transformer original, presentado en el artículo 'Attention is All You Need', es una arquitectura de red neuronal diseñada para tareas de secuencia a secuencia (seq2seq), como la traducción automática. A diferencia de las RNNs, los Transformers no utilizan recurrencia. En su lugar, se basan completamente en mecanismos de atención y capas feedforward.

28.1.1. Encoder-Decoder

El Transformer original sigue una arquitectura de codificador-decodificador (encoder-decoder):

- **Encoder (Codificador):** Toma la secuencia de entrada y la transforma en una representación contextualizada de alta dimensión.
- **Decoder (Decodificador):** Toma la representación generada por el encoder y genera la secuencia de salida, un elemento a la vez.

Tanto el encoder como el decoder están compuestos por una pila de N módulos idénticos (en el artículo original, N=6).

28.1.2. Componentes del Encoder

Cada módulo del encoder consta de dos subcapas principales:

1. **Multi-Head Self-Attention:** Aplica el mecanismo de self-attention (con las mejoras descritas anteriormente: queries, keys, values, escalado y multi-head) a la secuencia de entrada. Esto permite que la red atienda a diferentes partes de la secuencia al codificar cada elemento.
2. **Feedforward Network (FFN):** Una red feedforward completamente conectada que se aplica a cada posición de la secuencia de forma independiente. Típicamente, consta de dos capas lineales con una activación ReLU entre ellas.

Además, cada subcapa (self-attention y FFN) tiene una conexión residual (residual connection) alrededor de ella, seguida de una normalización de capa (layer normalization). Es decir, la salida de cada subcapa es:

$$\text{LayerNorm}(x + \text{Sublayer}(x))$$

Donde 'Sublayer(x)' es la función implementada por la subcapa (self-attention o FFN).

28.1.3. Componentes del Decoder

Cada módulo del decoder es similar al encoder, pero tiene tres subcapas principales:

1. **Masked Multi-Head Self-Attention:** Similar al self-attention del encoder, pero con una modificación: se aplica una máscara para evitar que la red atienda a posiciones futuras en la secuencia de salida. Esto es crucial para la generación secuencial, ya que la salida en cada paso de tiempo solo debe depender de las salidas anteriores, no de las futuras.
2. **Multi-Head Cross-Attention:** Aplica atención *entre* la salida del decoder y la salida del encoder. Esto permite que el decoder se 'enfoque' en las partes relevantes de la secuencia de entrada al generar cada elemento de la secuencia de salida. Las queries provienen de la capa anterior del decoder, y las keys y values provienen de la salida del encoder.
3. **Feedforward Network (FFN):** Similar al FFN del encoder.

Al igual que en el encoder, cada subcapa tiene una conexión residual alrededor de ella, seguida de una normalización de capa.

28.1.4. Positional Encoding

Dado que el Transformer no tiene recurrencia ni convoluciones, no tiene una forma inherente de incorporar información sobre la posición de los elementos en la secuencia. Para solucionar esto, se añade un posicional encoding (codificación posicional) a los embeddings de entrada, tanto en el encoder como en el decoder.

El positional encoding es un vector que codifica la posición de un elemento en la secuencia. En el artículo original, se utilizan funciones seno y coseno de diferentes frecuencias:

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{\text{model}}})$$

Donde:

- pos es la posición del elemento en la secuencia.
- i es la dimensión dentro del vector de positional encoding.
- d_{model} es la dimensión de los embeddings.

Estos vectores de positional encoding se suman a los embeddings de entrada. La idea es que estas funciones sinusoidales permiten a la red aprender a atender a la posición relativa de los elementos. También se pueden usar positional encodings aprendidos en lugar de los sinusoidales.

28.2. Ventajas de los Transformers sobre las RNNs

- **Paralelización:** A diferencia de las RNNs, que procesan la secuencia de forma secuencial, los Transformers pueden procesar todos los elementos de la secuencia en paralelo. Esto permite un entrenamiento mucho más rápido, especialmente en GPUs.
- **Memoria a largo plazo:** El mecanismo de self-attention permite a la red acceder directamente a cualquier elemento de la secuencia, independientemente de su distancia. Esto mitiga el problema de la memoria a corto plazo de las RNNs.
- **Interpretación:** Los pesos de atención pueden proporcionar información sobre qué partes de la entrada son más relevantes para cada salida.

28.3. Aplicaciones de los Transformers

Los Transformers se han convertido en la arquitectura dominante en el procesamiento del lenguaje natural, superando a las RNNs en una amplia variedad de tareas, incluyendo:

- Traducción automática.
- Modelado del lenguaje (predecir la siguiente palabra en una secuencia).
- Clasificación de texto.
- Respuesta a preguntas (question answering).
- Generación de texto.
- Resumen de texto.

Algunos modelos de Transformer famosos incluyen:

- **BERT (Bidirectional Encoder Representations from Transformers):** Un modelo pre-entrenado en una gran cantidad de texto que puede ser ajustado (fine-tuned) para una variedad de tareas de PLN.
- **GPT (Generative Pre-trained Transformer):** Una familia de modelos de lenguaje generativos (GPT-2, GPT-3, GPT-4) que pueden generar texto coherente y de alta calidad.
- **T5 (Text-to-Text Transfer Transformer):** Un modelo que enmarca todas las tareas de PLN como un problema de texto a texto.

Además, los Transformers se están aplicando cada vez más en otras áreas, como:

- **Visión artificial:** Vision Transformer (ViT), que aplica la arquitectura Transformer a imágenes.
- **Procesamiento de audio:** Aplicaciones en reconocimiento de voz y generación de música.
- **Tareas multimodales:** Combinación de texto, imágenes y/o audio. Un ejemplo notable es CLIP.

29. Aprendizaje No Supervisado y Autoencoders

Hasta ahora, la mayor parte de lo que hemos visto se ha centrado en el aprendizaje supervisado, donde tenemos un conjunto de datos etiquetado (es decir, conocemos la salida 'correcta' o 'deseada' para cada entrada). Sin embargo, en muchos casos, no disponemos de etiquetas. El aprendizaje no supervisado se ocupa de encontrar patrones y estructuras en datos no etiquetados.

29.1. Aprendizaje No Supervisado

En el aprendizaje no supervisado, solo tenemos un conjunto de datos de entrada $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\}$, sin etiquetas asociadas. El objetivo es aprender algo sobre la estructura subyacente de los datos. Algunas tareas comunes de aprendizaje no supervisado incluyen:

- **Agrupamiento (Clustering):** Encontrar grupos de datos similares.
- **Reducción de dimensionalidad:** Encontrar una representación de los datos en un espacio de menor dimensión, preservando la información importante.
- **Estimación de densidad:** Aprender la distribución de probabilidad subyacente de los datos.
- **Generación de datos:** Aprender a generar nuevas muestras de datos similares a los datos de entrenamiento.
- **Detección de anomalías:** Identificar datos que son 'atípicos' o diferentes del resto.

29.2. Autoencoders

Los autoencoders son un tipo de red neuronal que se utiliza para el aprendizaje no supervisado. La idea básica de un autoencoder es aprender una representación codificada (o latente) de los datos de entrada, y luego reconstruir la entrada original a partir de esta representación codificada.

29.2.1. Arquitectura de un Autoencoder

Un autoencoder consta de dos partes principales:

- **Encoder (Codificador):** Una función f que mapea la entrada $\mathbf{x}$ a una representación codificada $\mathbf{z}$:

$$\mathbf{z} = f(\mathbf{x})$$

  La representación codificada $\mathbf{z}$ se llama a menudo código (code) o representación latente (latent representation). Típicamente, la dimensión de $\mathbf{z}$ es menor que la dimensión de $\mathbf{x}$, lo que obliga a la red a aprender una representación comprimida de los datos.

- **Decoder (Decodificador):** Una función g que mapea la representación codificada $\mathbf{z}$ de vuelta al espacio de entrada original, produciendo una reconstrucción $\hat{\mathbf{x}}$:

$$\hat{\mathbf{x}} = g(\mathbf{z})$$

El objetivo del entrenamiento es aprender las funciones f y g de tal manera que la reconstrucción $\hat{\mathbf{x}}$ sea lo más parecida posible a la entrada original $\mathbf{x}$. Es decir, el autoencoder trata de aprender la función identidad.

29.2.2. Función de Pérdida

La función de pérdida de un autoencoder mide la diferencia entre la entrada original $\mathbf{x}$ y la reconstrucción $\hat{\mathbf{x}}$. Una función de pérdida común es el error cuadrático medio (MSE):

$$\mathcal{L}(\mathbf{x}, \hat{\mathbf{x}}) = ||\mathbf{x} - \hat{\mathbf{x}}||^2 = ||\mathbf{x} - g(f(\mathbf{x}))||^2$$

El objetivo del entrenamiento es minimizar esta pérdida sobre todos los datos de entrenamiento:

$$\min_{f,g} \frac{1}{N} \sum_{i=1}^{N} ||\mathbf{x}_i - g(f(\mathbf{x}_i))||^2$$

29.2.3. Deep Autoencoders

En la práctica, tanto el encoder como el decoder suelen ser redes neuronales profundas (con múltiples capas). Esto permite al autoencoder aprender representaciones no lineales y más complejas de los datos.

29.2.4. Ejemplo: Autoencoder para MNIST

Si usamos digitos de MNIST (imágenes de 28x28 = 784 píxeles) y contruimos un autoencoder con una capa oculta de, por ejemplo, 32 neuronas, la red aprenderá una representación de los dígitos en un espacio de 32 dimensiones. El encoder mapeará cada imagen de 784 píxeles a un vector de 32 números, y el decoder intentará reconstruir la imagen original a partir de este vector de 32 números.

29.3. Aplicaciones de los Autoencoders

- **Reducción de dimensionalidad:** La representación codificada $\mathbf{z}$ puede utilizarse como una representación de baja dimensión de los datos de entrada.
- **Eliminación de ruido (denoising):** Se puede entrenar un autoencoder para reconstruir una imagen limpia a partir de una versión ruidosa de la misma.
- **Detección de anomalías:** Las anomalías (datos atípicos) tendrán un error de reconstrucción alto, ya que el autoencoder no ha sido entrenado para representarlas.
- **Pre-entrenamiento de redes profundas:** Se pueden utilizar autoencoders para pre-entrenar las capas de una red profunda antes de entrenarla para una tarea supervisada (esto era más común antes de que se desarrollaran técnicas de entrenamiento más efectivas, como Batch Normalization).
- **Generación de datos (con Variational Autoencoders):** Los Variational Autoencoders (VAEs) son una variante de los autoencoders que permiten generar nuevas muestras de datos similares a los datos de entrenamiento.

29.4. Interpolación en el Espacio Latente

Una propiedad interesante de los autoencoders es que se puede interpolar entre dos puntos en el espacio latente y generar muestras intermedias. Esto es una consecuencia directa de usar una función de activación en las capas del autoencoder. Si tenemos que f es el encoder y g el decoder, y tenemos dos datos de entrada x_1 y x_2, entonces:

1. Se codifican en el espacio latente con f, obteniendo los vectores latentes $\mathbf{z}_1 = f(\mathbf{x}_1)$ y $\mathbf{z}_2 = f(\mathbf{x}_2)$.
2. Se interpolan puntos en el espacio latente, entre $\mathbf{z}_1$ y $\mathbf{z}_2$. Por ejemplo, usando una interpolación lineal: $\mathbf{z}_\alpha = \alpha\mathbf{z}_1 + (1-\alpha)\mathbf{z}_2$, donde α varía entre 0 y 1.
3. Se decodifican los puntos interpolados $\mathbf{z}_\alpha$ de vuelta al espacio original, usando el decoder g: $\hat{\mathbf{x}}_\alpha = g(\mathbf{z}_\alpha)$.

Esto permite generar una secuencia de muestras que se transforman gradualmente de $\mathbf{x}_1$ a $\mathbf{x}_2$.

30. Redes Generativas Adversarias (Generative Adversarial Networks - GANs)

Las Redes Generativas Adversarias (GANs) son un tipo de modelo generativo, es decir, un modelo que aprende a generar nuevos datos que se asemejan a los datos de entrenamiento. A diferencia de los autoencoders, que aprenden una representación latente y luego reconstruyen la entrada, las GANs aprenden a generar datos completamente nuevos a partir de una distribución de probabilidad aprendida.

30.1. Introducción y Motivación

Los modelos generativos se enmarcan dentro del aprendizaje no supervisado. El objetivo principal de los modelos generativos es, a partir de un conjunto de datos de entrenamiento $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\}$, aprender la distribución de probabilidad subyacente $p(\mathbf{x})$ de los datos, o una aproximación $p_\theta(\mathbf{x})$. Una vez aprendida esta distribución, se puede generar nuevos datos muestreando de ella.

30.2. Concepto de GAN

Las GANs, introducidas por Goodfellow et al. (2014), se basan en una idea ingeniosa: entrenar dos redes neuronales que compiten entre sí en un juego de suma cero. Estas redes son:

- **Generador (Generator, G):** Toma como entrada un vector de ruido aleatorio ($\mathbf{z}$, típicamente muestreado de una distribución normal o uniforme) y produce una muestra de datos ($\mathbf{x}'$). El objetivo del generador es aprender a generar datos que sean indistinguibles de los datos reales.
- **Discriminador (Discriminator, D):** Toma como entrada una muestra de datos (ya sea una muestra real del conjunto de entrenamiento o una muestra generada por el generador) y produce una probabilidad de que la muestra sea real (en lugar de falsa/generada).

El generador y el discriminador se entrenan simultáneamente en un proceso de 'competencia':

- El generador intenta engañar al discriminador, produciendo muestras cada vez más realistas.
- El discriminador intenta distinguir entre las muestras reales y las muestras generadas, mejorando su capacidad de detección.

Este proceso se puede visualizar como un juego entre un falsificador (generador) y un policía (discriminador). El falsificador intenta crear billetes falsos que sean indistinguibles de los reales, mientras que el policía intenta detectar los billetes falsos. Con el tiempo, ambos mejoran en sus respectivas tareas.

30.3. Arquitectura y Funcionamiento

Matemáticamente, podemos expresar el funcionamiento de una GAN de la siguiente manera:

- El generador es una función $G(\mathbf{z}; \boldsymbol{\theta}_g)$ que mapea un vector de ruido $\mathbf{z}$ (normalmente de baja dimensión) a un punto en el espacio de los datos $\mathbf{x}'$. $\boldsymbol{\theta}_g$ representa los parámetros del generador.
- El discriminador es una función $D(\mathbf{x}; \boldsymbol{\theta}_d)$ que toma una muestra $\mathbf{x}$ (real o generada) y produce un escalar entre 0 y 1, que representa la probabilidad de que la muestra sea real. $\boldsymbol{\theta}_d$ representa los parámetros del discriminador.

30.4. Entrenamiento de una GAN

El entrenamiento de una GAN implica un proceso de optimización en dos etapas, alternando entre la optimización del discriminador y la optimización del generador:

1. **Optimización del discriminador (D):**
   - Se congelan los parámetros del generador ($\boldsymbol{\theta}_g$).
   - Se toman muestras del conjunto de datos real ($\mathbf{x}$) y se generan muestras falsas ($\mathbf{x}' = G(\mathbf{z})$) a partir del generador, muestreando el ruido $\mathbf{z}$.
   - Se entrena el discriminador para maximizar la probabilidad de clasificar correctamente las muestras reales y las muestras falsas. Esto se puede lograr minimizando la siguiente función de pérdida (binary cross-entropy):
   $$\mathcal{L}_D = -\mathbb{E}_{\mathbf{x} \sim p_{data}(\mathbf{x})}[\log D(\mathbf{x})] - \mathbb{E}_{\mathbf{z} \sim p_z(\mathbf{z})}[\log(1 - D(G(\mathbf{z})))]$$
2. **Optimización del generador (G):**
   - Se congelan los parámetros del discriminador ($\boldsymbol{\theta}_d$).
   - Se generan muestras falsas ($\mathbf{x}' = G(\mathbf{z})$) a partir del generador.
   - Se entrena el generador para minimizar la probabilidad de que el discriminador clasifique correctamente las muestras generadas como falsas. Esto se puede lograr minimizando la siguiente función de pérdida:
   $$\mathcal{L}_G = -\mathbb{E}_{\mathbf{z} \sim p_z(\mathbf{z})}[\log(D(G(\mathbf{z})))]$$
   O, equivalentemente, maximizando $\mathbb{E}_{\mathbf{z} \sim p_z(\mathbf{z})}[\log(D(G(\mathbf{z})))]$

Este proceso se repite iterativamente hasta que se alcanza un equilibrio (en teoría, el generador produce muestras indistinguibles de los datos reales, y el discriminador tiene una precisión del 50

30.5. Problemas del Entrenamiento de GANs

El entrenamiento de GANs puede ser notoriamente difícil y presenta varios desafíos:

- **Inestabilidad del entrenamiento:** El proceso de entrenamiento puede ser inestable, con oscilaciones y falta de convergencia.
- **Desvanecimiento del gradiente:** Si el discriminador se vuelve demasiado bueno demasiado rápido, el generador puede dejar de aprender (el gradiente se desvanece).
- **Colapso del modo (mode collapse):** El generador puede aprender a producir solo un pequeño subconjunto de las posibles muestras, ignorando la mayor parte de la distribución de datos real. Es decir, el generador çolapsa.[a] un modo de la distribución real.
- **Dificultad de evaluación:** No existe una métrica de evaluación única y universalmente aceptada para las GANs. Evaluar la calidad de las muestras generadas es subjetivo y difícil.

Se han propuesto numerosas variantes de GANs y técnicas de entrenamiento para abordar estos problemas.

30.6. Aplicaciones de las GANs

Las GANs tienen una amplia variedad de aplicaciones, incluyendo:

- **Generación de imágenes:** Generar imágenes realistas de rostros, paisajes, objetos, etc.
- **Transferencia de estilo:** Transferir el estilo de una imagen a otra.
- **Super-resolución:** Aumentar la resolución de una imagen.
- **Inpainting:** Rellenar regiones faltantes en una imagen.
- **Traducción de imagen a imagen (image-to-image translation):** Transformar una imagen de un dominio a otro (por ejemplo, de una foto a una pintura, de un boceto a una foto, de día a noche).
- **Generación de texto a imagen (text-to-image generation):** Generar una imagen a partir de una descripción textual.
- **Edición de imágenes.**
- **Generación de datos para aumentar conjuntos de datos de entrenamiento.**

30.7. Tipos de GANs (solo nombrar, sin detalle)

Se mencionan sin entrar en detalle:

- DCGAN (Deep Convolutional GAN)
- Conditional GAN (cGAN)
- CycleGAN
- StyleGAN
- BigGAN