DECONVOLUCIÓN CIEGA DE IMÁGENES CON TÉCNICAS DE APRENDIZAJE PROFUNDO

1.- Idea general de aprendizaje automático

El Aprendizaje Profundo (Deep Learning) constituye un campo interdisciplinario que aborda principios matemáticos, estadísticos y de ciencias de la computación, con el propósito de construir de modelos con la capacidad de adquirir representaciones complejas a partir de conjuntos de datos; Este enfoque se orienta hacia la emulación del funcionamiento cerebral humano, capacitando a los sistema computacionales para asimilar y procesar información proveniente de diversas fuentes, lo que les permite detectar patrones , realizar predicciones y mejorar su desempeño a través de la experiencia adquirida mediante los inputs recibidos.

1.1 Aprender de los datos

El caso de uso más sencillo para un modelo entrenado a partir de datos es cuando se tiene acceso a una señal x, por ejemplo, consideremos el escenario donde se tiene una imagen de una matrícula de un vehículo (x), a partir de la cual se quiere predecir una cantidad y, como la cadena de caracteres escritos en la matrícula.

En situaciones donde los datos (x) son de alta dimensionalidad y provienen de entornos no controlados, encontrar una relación analítica precisa entre la entrada (x) y la salida (y) puede ser extremadamente complejo.

Lo que sí se puede hacer es recopilar un gran conjunto de entrenamiento ($\mathcal{D}$) que contiene pares de datos de entrada y salida (x_n,y_n), e idear un modelo paramétrico f, con parámetros entrenables (w) que modulan su comportamiento El objetivo del entrenamiento es encontrar valores de parámetros (w“) que minimice una función de pérdida ($\mathcal{L}$(w)), que cuantifica la discrepancia entre las predicciones del modelo y las salidas reales en el conjunto del entrenamiento.

1.2 Underfitting y Overfitting

Una consideración clave es encontrar un equilibrio entre la capacidad del modelo (su flexibilidad y habilidad para ajustarse a datos diversos) y la cantidad y calidad de los datos de entrenamiento.

Underfitting: Cuando la capacidad es insuficiente, el modelo no puede ajustarse a los datos (es decir, el modelo es demasiado simple para capturar la estructura subyacente de los datos).

Overfitting: Cuando la cantidad de datos es insuficiente, el modelo suele aprender características específicas de los ejemplos de entrenamiento, lo que se traduce en un excelente rendimiento durante el entrenamiento, a costa de un peor ajuste a la estructura global de los datos y un rendimiento deficiente a las nuevas entradas (es decir, el modelo

se ajusta demasiado a los datos de entrenamiento, perdiendo capacidad de generalización).

El arte del aprendizaje automático radica en diseñar modelos que no sean demasiado flexibles pero capaces de ajustarse a los datos sin overfitting.

1.3 Categorías de modelos

Podemos categorizar los modelos de aprendizaje en tres grandes grupos_

- Regresión: Tiene como objetivo predecir valores continuos basados en variables independientes. Por ejemplo, posición geométrica de un objeto dada una señal de entrada x; U otro ejemplo: predecir el precio de una casa basándose en características como el área, el número de habitaciones, la ubicación, etc.
- Clasificación: El objetivo es asignar una etiqueta/categoría a una instancia de datos basándose en sus características. Por ejemplo, para identificar el tipo de animal en una imagen.
- Modelado de densidad: Tiene como objetivo modelar la función de densidad de probabilidad de los datos μX para describir la distribución de los datos (es decir, el objetivo es entender la estructura subyacente de los datos y modelar cómo se distribuyen en el espacio de características). Esto puede ser útil en la generación de imágenes, donde se desea crear nuevas imágenes que se parezcan a las de un conjunto de datos de entrenamiento.

Tanto la regresión como la clasificación se refieren generalmente al aprendizaje supervisado, ya que el valor a predecir, que se requiere como objetivo durante el entrenamiento, debe ser proporcionado, mientras la modelización de densidad se considera aprendizaje no supervisado, ya que basta con tomar datos existentes sin necesidad de producir una verdad de referencia asociada.

Estas tres categorías no son mutuamente excluyentes, existen multitud de problemas los cuales se abordan utilizando más de una de estas categorías.

2.- Regresión Lineal

En el campo del aprendizaje automático la regresión lineal emerge como una de las herramientas fundamentales para modelar y comprender relaciones entre variables. En este contexto, el objetivo principal es encontrar una línea recta que se ajuste de manera óptima a un conjunto de datos dispersos en un plano, lo que implica la predicción de un valor numérico a partir de variables independientes (aquí es donde radican los problemas de la regresión).

Consideremos un ejemplo práctico: supongamos que se desea predecir el precio de viviendas en función de su superficie y su antigüedad. Para desarrollar un modelo de predicción de precios de la vivienda, necesitamos disponer de datos que incluyan el precio de venta, la superficie y la antigüedad de las casas. En la terminología del aprendizaje automático, el conjunto de datos se denomina conjunto de datos de entrenamiento o conjunto de entrenamiento, y cada fila (que contiene los datos

correspondientes a una venta) se denomina ejemplo (o punto de datos, instancia, muestra). La variable que intentamos predecir (el precio) se denomina etiqueta (u objetivo). Las variables (superficie y antigüedad) en las que se basan las predicciones se denominan características (o covariables).

2.1 Conceptos Básicos

La regresión lineal es a la vez la más sencilla y la más popular de las herramientas estándar para abordar los problemas de regresión. Este método, que tiene sus raíces en el siglo XIX (en los trabajos pioneros de Gauss y Legendre), se basa en varios supuestos fundamentales que deben cumplirse para su aplicación efectiva.

En primer lugar, se supone que la relación entre las características (x) y el objetivo (y) es aproximadamente lineal, lo que implica que la media condicional de (y) dado (x) E[Y|X=x] puede expresarse como una suma ponderada de las características (x). Esto significa que el valor esperado de la variable dependiente varia linealmente con respecto a las variables independientes. Esta configuración presupone el "ruido de observación"; esto se refiere a que el valor objetivo pueda desviarse de su valor esperado. Este ruido puede deberse a diversas razones, como errores de medición, fluctuaciones aleatorias en el sistema que estamos estudiando, etc. Para abordar este ruido se impone el supuesto de que dicho ruido se comporta bien, siguiendo una distribución gaussiana o normal.

En lo que se refiere a la notición, 'n' se utiliza para indicar el número total de ejemplos en el conjunto de datos. Los superíndices se utilizan para enumerar las muestras y los objetivos ($x^{(i)}$), y subíndices para indexar las coordenadas ($x_j^{(i)}$). Por ejemplo, si estamos analizando el desempeño académico de estudiantes y cada fila de nuestro conjunto de datos representa a un estudiante, entonces $x^{(i)}$ sería una fila particular que corresponde a un estudiante específico. Ahora, $x_j^{(i)}$ indica el valor de una característica específica para esa observación en particular. Aquí, 'j' representa el índice de la característica. Por ejemplo, si estamos analizando el desempeño académico de los estudiantes y tenemos características como horas de estudio, puntaje en matemáticas, puntaje en ciencias, etc., entonces $x_j^{(i)}$ sería el valor de la característica 'j' (por ejemplo, horas de estudio, puntaje en matemáticas, etc.) para la observación 'i' (por ejemplo, el estudiante 'i').

2.2 Modelo matemático

El supuesto de linealidad significa que se asume que la relación entre las características (variable independiente) y el objetivo (variable dependiente) es lineal. Esto es, que el valor esperado del objetivo puede expresarse como una suma ponderada de las características. El modelo en su forma compacta es el siguiente:

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

En la ecuación x es el vector de características, w se denomina vector de pesos/ponderaciones, y b se denomina sesgo. La ponderación determina la influencia/peso de cada característica en la predicción, mientras que el sesgo determina el valor de estimación cuando todas las características son 0.

Para tratar las características de todo nuestro conjunto de datos de n ejemplos de manera eficiente se utiliza la matriz de diseño, donde X contiene una fila para cada ejemplo y una columna para cada característica, haciendo que las predicciones ^y se expresen mediante un producto matriz-vector:

$$y\hat{}=\mathbf{X}\cdot\mathbf{w}+b$$

Dadas las características de un conjunto de datos de entrenamiento 'X' y las correspondientes etiquetas (conocidas) 'y', el objetivo de la regresión lineal es encontrar el vector de pesos 'w' y el término de sesgo 'b' de forma que, dadas las características de un nuevo ejemplo de datos muestreado a partir de la misma distribución que 'X' la etiqueta del nuevo ejemplo se prediga (en principio) con el menor error, es decir, que, en promedio, los valores del vector de pesos 'w' y el sesgo 'b' hagan que las predicciones de nuestro modelo se ajusten lo más posible a los valores reales observados en los datos.

Antes de buscar los mejores parámetros w y b necesitaremos:

(i) Una medida de calidad.
(ii) Un procedimiento para actualizar el modelo.

2.3 Función de pérdida

Para ajustar nuestro modelo a los datos, se necesita definir una medida de adecuación/ajuste; esta es la función de pérdida, la cual cuantifica la diferencia entre los valores reales y las predicciones del objetivo. En el contexto de la regresión lineal, la función de pérdida más común es el error cuadrático medio (MSE):

$$l^{(i)}(\mathbf{w},b)=\frac{1}{2}\left(\hat{y}^{(i)}-y^{(i)}\right)^2$$

- $y\hat{}^{(i)}$ es la predicción del modelo para el ejemplo i.
- $y^{(i)}$ es el valor real correspondiente al ejemplo i.

La pérdida suele ser un número no negativo en el que los valores más pequeños son mejores y las predicciones perfectas incurren en una pérdida de 0.

Es importante destacar que la naturaleza cuadrática del MSE puede hacer que el modelo evite grandes errores, pero sea excesivamente sensible a los datos anómalos (es decir, el minimizar esta función puede llevar a un ajuste excesivo a puntos atípicos en el conjunto de datos). Para medir la calidad de un modelo en todo el conjunto de datos de n ejemplos, basta con promediar las pérdidas en el conjunto de entrenamiento:

$$L(\mathbf{w},b)=\frac{1}{n}\sum_{i=1}^{n}l^{(i)}(\mathbf{w},b)=\frac{1}{n}\sum_{i=1}^{n}\frac{1}{2}\left(\mathbf{w}^\top\mathbf{x}^{(i)}+b-y^{(i)}\right)^2.$$

2.4 Solución analítica

La regresión lineal ofrece una manera sencilla y elegante de calcular los valores óptimos de los parámetros del modelo que minimicen la función de pérdida a través de la siguiente. Primero:

$$L(w)=||y-Xw||^2$$

podemos subsumir el sesgo b en el parámetro w añadiendo una columna a la matriz de diseño formada por toda por 1s. Entonces nuestro problema de predicción es minimizar ||y - Xw||^2. Mientras la matriz de diseño X tenga rango completo (ninguna característica depende linealmente de las demás), sólo habrá un punto crítico en la superficie de pérdida, que corresponde al mínimo de la pérdida en todo el dominio. Si se toma la derivada de la pérdida con respecto a w e igualándola a cero se obtiene:

$$\partial_{\mathbf{w}}\|\mathbf{y}-\mathbf{X}\mathbf{w}\|^2 = 2\mathbf{X}^\top(\mathbf{X}\mathbf{w}-\mathbf{y}) = 0 \text{ and hence } \mathbf{X}^\top\mathbf{y} = \mathbf{X}^\top\mathbf{X}\mathbf{w}.$$

Resolviendo para w, nos proporciona la solución óptima para el problema de optimización:

$$\mathbf{w}^* = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}$$

Nótese que esta solución solo será única cuando la matriz X^тX es invertible.

2.5 Descenso de Gradiente Estocástico por Mini-Lotes

Esta técnica es crucial para optimizar modelos que no pueden ser resueltos de manera analítica y pueden resultar difíciles de optimizar. El descenso de gradiente es el algoritmo principal para reducir iterativamente el error mediante la actualización de los parámetros en la dirección que disminuye incrementalmente la función de pérdida.

La aplicación más básica del descenso de gradiente es calcular la derivada de la función de pérdida, que es un promedio de las pérdidas calculadas en cada ejemplo del conjunto de datos. Sin embargo, en la práctica realizar una actualización completa en cada paso puede ser extremadamente lento, especialmente si hay mucha redundancia en los datos de entrenamiento.

El descenso de gradiente estocástico (SGD) es una solución eficaz (incluso para grandes conjuntos de datos) que consiste en considerar un único ejemplo a la vez para actualizar los parámetros. Aunque eficiente, el SGD tiene desventajas tanto computacionales como estadísticas, como el tiempo necesario para procesar una muestra a la vez y la dificultad para aplicar ciertas técnicas como la normalización por lotes.

Para abordar estos problemas se propone una estrategia intermedia, el descenso de gradiente estocástico mini-batch, donde en lugar de tomar un lote completo o una sola muestra a la vez, tomamos un mini-lote de observaciones.

El funcionamiento de este modelo es el siguiente:

(i) Inicializa los valores de los parámetros del modelo de forma aleatoria.
(ii) Muestrea iterativamente minilotes aleatorios de los datos, actualizando los parámetros en la dirección del gradiente negativo de la pérdida promedio en el minibatch (el gradiente indica la dirección y la magnitud en la que la pérdida cambia más rápidamente con respecto a los parámetros del modelo).

Después del entrenamiento, los parámetros del modelo se registran para su evaluación en un conjunto de datos de validación separado.

Aunque el algoritmo converge lentamente hacia los minimizadores de pérdida, normalmente no los encontrará exactamente en un número finito de pasos, sin embargo, en la práctica, el objetivo es encontrar cualquier conjunto de parámetros que conduzca a predicciones precisas sobre datos nunca antes vistos.

2.6 Vectorización para velocidad

Durante el proceso de entrenamiento de modelos de regresión lineal, la eficiencia computacional es crucial para manejar grandes conjuntos de datos de manera efectiva. Cuando entrenamos nuestros modelos, normalmente queremos procesar mini-lotes enteros de ejemplos simultáneamente. Para hacerlo de forma eficiente, es necesario vectorizar los cálculos.

La vectorización es una técnica fundamental en la programación computacional que consiste en realizar operaciones sobre vectores o matrices enteras en lugar de elementos individuales. En el contexto de Python y el aprendizaje automático, la vectorización se logra utilizando bibliotecas especializadas de álgebra lineal, como 'PyTorch', que están optimizadas para ejecutar operaciones en matrices de manera eficiente.

Para ilustrar la importancia de la vectorización, consideremos dos métodos para sumar vectores de alta dimensionalidad:

1er método) Iteramos sobre cada elemento de los vectores utilizando un bucle for en Python y sumamos los elementos correspondientes uno a uno.

```
n = 10000
a = torch.ones(n)
b = torch.ones(n)
c = torch.zeros(n)
t = time.time()

for i in range(n):
c[i] = a[i] + b[i]

f'{time.time() - t:.5f} sec'
```

```
'0.17802 sec'
```

2do método) aprovechamos las capacidades de vectorización de la biblioteca PyTorch para sumar los vectores en una sola operación '+'.

```
t = time.time()
d = a + b
f'{time.time() - t:.5f} sec'
```

```
'0.00036 sec'
```

Al comparar los tiempos de ejecución de ambos métodos, observamos una diferencia significativa en la eficiencia a favor del método donde utilizamos operaciones vectorizadas. Este aumento en la velocidad de ejecución puede ser del orden de magnitud, lo que demuestra la importancia de la vectorización para optimizar el rendimiento de los algoritmos de regresión lineal. Además, también reduce la complejidad del código y aumenta su portabilidad al trasladar la carga computacional a las bibliotecas de álgebra lineal.

2.7 La distribución normal y la pérdida al cuadrado

La distribución normal o gaussiana es una de las distribuciones de probabilidad más importantes en la teoría estadística. Esta viene determinada por dos parámetros: la media μ y la varianza σ2 (desviación típica σ); y viene dada por la fórmula:

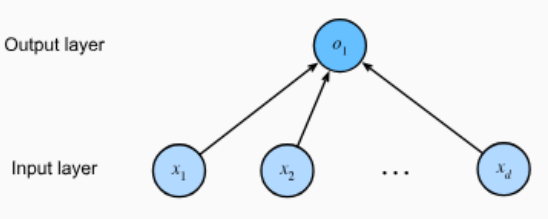
$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right).$$

La regresión lineal busca modelar la relación entre una variable dependiente Y y una o más variables independientes X mediante una función lineal. En el contexto de la regresión lineal, se asume que las observaciones están sujetas a errores aleatorios (es decir, las diferencias entre los valores observados y los valores predichos por el modelo) que siguen una distribución normal. Es decir, se supone que los errores de predicción ε tienen una distribución normal con una media de cero (se espera que los errores de predicción se distribuyan alrededor de cero) y una varianza constante σ^2 (la dispersión de los errores es constante en todo el rango de los valores de las variables independientes).

La elección de la pérdida al cuadrado en la regresión lineal se justifica teóricamente mediante el principio de máxima verosimilitud (es decir, buscar aquellos valores de los parámetros del modelo que hacen que nuestros datos sean lo más probables posible). Según este principio, los mejores estimadores de los parámetros del modelo son aquellos que maximizan la probabilidad de observar los datos dados los parámetros del modelo. En el caso de la regresión lineal, maximizar la verosimilitud es equivalente a minimizar la suma de los cuadrados de los errores residuales, ya que se asume que los errores siguen una distribución normal. Por lo tanto, la minimización de la pérdida al cuadrado en la regresión lineal conduce a estimaciones de parámetros que son consistentes con el enfoque de máxima verosimilitud.

2.8 Regresión Lineal como red neuronal

La siguiente figura representa la regresión lineal como una red neuronal. El diagrama destaca el patrón de conectividad, por ejemplo, cómo se conecta cada entrada $x_1,\ldots,x_d$ (es decir las diferentes características o variables independientes) a la salida 'o_1' (la predicción), pero no los valores específicos que toman los pesos o los sesgos.



En resumen, en este ilustración podemos pensar en la regresión lineal como una red neuronal totalmente conectada de una sola capa.

3.2.- Implementación en Pytorch

En pytorch tenemos tres clases principales:

(i). Module: contiene módulos, pérdidas y métodos de optimización.

(ii). DataModule: proporciona cargadores de datos para el entrenamiento y la validación.

(iii). ambas clases se combinan utilizando la clase Trainer, que nos permite entrenar modelos en una variedad de plataformas de hardware.

```
import time
import numpy as np
import torch
from torch import nn
from d2l import torch as d2l
```

3.2.1.- Utilidades

Uno de los contras de las definiciones de clases suelen ser bloques de código bastante largos; Para solucionar esto existe la sentencia '**@add_to_class(*Class*)**' .

Por ejemplo: queremos implementar una clase **A** que va a tener un método **do**. En lugar de que en A este todo el código de la clase, podemos implementar en **A** los datos miembros de esta y su inicializador. Luego podemos declarar el método **do** fuera del alcance de la clase, pero añadiendo el decorador **@add_to_class(A)**' para que el método se incorpore como método de la clase A:

```python
def add_to_class(Class):  #@save
    """Register functions as methods in created class."""
    def wrapper(obj):
        setattr(Class, obj.__name__, obj)
    return wrapper

class A:
    def __init__(self):
        self.b = 1

a = A()

@add_to_class(A)
def do(self):
    print('Class attribute "b" is', self.b)

a.do()
```

```
Class attribute "b" is 1
```

La segunda es una clase de utilidad que guarda todos los argumentos que se le pasan al método __init__ de una clase como atributos de clase. Esto nos permite llamar al constructor con diferentes argumentos sin tener que añadir código adicional.

```python
class HyperParameters:  #@save
    """The base class of hyperparameters."""
    def save_hyperparameters(self, ignore=[]):
        raise NotImplemented
```

Para usarlo, definimos nuestra clase que hereda de HyperParameters y llama a save_hyperparameters en el método __init__.

```python
# Call the fully implemented HyperParameters class saved in d2l
class B(d2l.HyperParameters):
    def __init__(self, a, b, c):
        self.save_hyperparameters(ignore=['c'])
        print('self.a =', self.a, 'self.b =', self.b)
        print('There is no self.c =', not hasattr(self, 'c'))

b = B(a=1, b=2, c=3)
```

```
self.a = 1 self.b = 2
There is no self.c = True
```

3.2.2.- Module

La clase **Module** es la clase base de todos los modelos (y de las redes neuronales). Necesitaremos tres métodos:

(i). **__init__**: almacena los parámetros aprendibles.

(ii). **training_step**: acepta un lote de datos para devolver el valor de pérdida.

(iii). **configure_optimizers**: devuelve el método de optimización, o una lista de ellos, que se utiliza para actualizar los parámetros aprendibles.

```
class Module(nn.Module, d2l.HyperParameters):  #@save
    """The base class of models."""
    def __init__(self, plot_train_per_epoch=2, plot_valid_per_epoch=1):
        super().__init__()
        self.save_hyperparameters()
        self.board = ProgressBoard()

    def loss(self, y_hat, y):
        raise NotImplementedError

    def forward(self, X):
        assert hasattr(self, 'net'), 'Neural network is defined'
        return self.net(X)

    def plot(self, key, value, train):
        """Plot a point in animation."""
        assert hasattr(self, 'trainer'), 'Trainer is not inited'
        self.board.xlabel = 'epoch'
        if train:
            x = self.trainer.train_batch_idx / \
                self.trainer.num_train_batches
            n = self.trainer.num_train_batches / \
                self.plot_train_per_epoch
        else:
            x = self.trainer.epoch + 1
            n = self.trainer.num_val_batches / \
                self.plot_valid_per_epoch
        self.board.draw(x, value.to(d2l.cpu()).detach().numpy(),
                        ('train_' if train else 'val_') + key,
                        every_n=int(n))

    def training_step(self, batch):
        l = self.loss(self(*batch[:-1]), batch[-1])
        self.plot('loss', l, train=True)
        return l

    def validation_step(self, batch):
        l = self.loss(self(*batch[:-1]), batch[-1])
        self.plot('loss', l, train=False)
```

```
    def configure_optimizers(self):
        raise NotImplementedError
```

Opcionalmente podemos definir **validation_step** para informar de las medidas de evaluación.

3.2.3.- Data

La clase **DataModule** es la clase base para los datos. Utiliza el método **__init__** para preparar los datos. El método **train_dataloader** devuelve el cargador de datos para el conjunto de datos de entrenamiento. Un cargador de datos es un generador (Python) que produce un lote de datos cada vez que se utiliza. Este lote se introduce en el método **training_step** de Module para calcular la pérdida. *(Existe un **val_dataloader** opcional para devolver el cargador del conjunto de datos de validación. Se comporta de la misma manera, excepto que produce lotes de datos para el método validation_step de Module.)*

```
class DataModule(d2l.HyperParameters):  #@save
    """The base class of data."""
    def __init__(self, root='../data', num_workers=4):
        self.save_hyperparameters()

    def get_dataloader(self, train):
        raise NotImplementedError

    def train_dataloader(self):
        return self.get_dataloader(train=True)

    def val_dataloader(self):
        return self.get_dataloader(train=False)
```

3.2.4.- Trainning

La clase **Trainer** entrena los parámetros aprendibles de la clase Módulo con los datos especificados en DataModule. El método clave es **fit**, que acepta dos argumentos: *model*, una instancia de Module, y *data*, una instancia de DataModule. A continuación, itera sobre todo el conjunto de datos *max_epochs* veces para entrenar el modelo. Como antes, dejaremos la implementación de este método para capítulos posteriores.

```
class Trainer(d2l.HyperParameters):  #@save
    """The base class for training models with data."""
    def __init__(self, max_epochs, num_gpus=0, gradient_clip_val=0):
        self.save_hyperparameters()
        assert num_gpus == 0, 'No GPU support yet'

    def prepare_data(self, data):
        self.train_dataloader = data.train_dataloader()
        self.val_dataloader = data.val_dataloader()
        self.num_train_batches = len(self.train_dataloader)
```

```
        self.num_val_batches = (len(self.val_dataloader)
                                if self.val_dataloader is not None else 0)

    def prepare_model(self, model):
        model.trainer = self
        model.board.xlim = [0, self.max_epochs]
        self.model = model

    def fit(self, model, data):
        self.prepare_data(data)
        self.prepare_model(model)
        self.optim = model.configure_optimizers()
        self.epoch = 0
        self.train_batch_idx = 0
        self.val_batch_idx = 0
        for self.epoch in range(self.max_epochs):
            self.fit_epoch()

    def fit_epoch(self):
        raise NotImplementedError
```

3.4.- Implementación desde 0: Regresión Lineal

Comentado [JT1]: En el código 'regresion lineal - pytorch.ipynb' está el código explicado.

Ahora estamos listos para trabajar con una implementación de regresión lineal completamente funcional. Vamos a implementar el método desde cero, incluyendo:

(i) el modelo;

(ii) la función de pérdida;

(iii) un optimizador de descenso de gradiente estocástico minibatch;

(iv) la función de entrenamiento que une todas estas piezas.

```
%matplotlib inline
import torch
from d2l import torch as d2l
```

3.4.1.- Definiendo el modelo

Antes de que podamos empezar a optimizar los parámetros de nuestro modelo mediante SGD (descenso de gradiente estocástico) de mini-lotes, necesitamos tener algunos parámetros en primer lugar. A continuación, inicializamos los pesos extrayendo números aleatorios de una distribución normal con media 0 y una desviación estándar de 0,01. El número mágico 0,01 suele funcionar bien en la práctica, pero se puede especificar un valor diferente mediante el argumento sigma. Además, establecemos el sesgo a 0.

```python
class LinearRegressionScratch(d2l.Module):  #@save
    """The linear regression model implemented from scratch."""
    def __init__(self, num_inputs, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.w = torch.normal(0, sigma, (num_inputs, 1), requires_grad=True)
        self.b = torch.zeros(1, requires_grad=True)
```

A continuación, tenemos que definir nuestro modelo, relacionando su entrada y sus parámetros con su salida. Tomamos el producto vectorial-matricial de las características de entrada X y las ponderaciones del modelo w, y añadimos el desplazamiento b a cada ejemplo. El producto Xw es un vector y b es un escalar (Cuando sumamos un vector y un escalar, el escalar se añade a cada componente del vector). El método forward resultante se registra en la clase LinearRegressionScratch mediante **add_to_class**.

```python
@d2l.add_to_class(LinearRegressionScratch)  #@save
def forward(self, X):
    return torch.matmul(X, self.w) + self.b
```

3.4.2.- Definiendo la función de pérdida

Dado que la actualización de nuestro modelo requiere tomar el gradiente de nuestra función de pérdida, debemos definir primero la función de pérdida. Utilizaremos la función de pérdida al cuadrado (error cuadrático medio MSE). En la implementación, necesitamos transformar el valor verdadero **y** (etiquetas reales) en la forma del valor predicho **y_hat** (predicciones del modelo). El resultado devuelto por el siguiente método también tendrá la misma forma que **y_hat**. También devolvemos el valor medio de la pérdida entre todos los ejemplos del mini-lote.

```python
@d2l.add_to_class(LinearRegressionScratch)  #@save
def loss(self, y_hat, y):
    l = (y_hat - y) ** 2 / 2
    return l.mean()
```

3.4.3.- Definiendo del algoritmo de optimización

En cada paso, utilizando un mini-lote extraído aleatoriamente de nuestro conjunto de datos, estimamos el gradiente de la pérdida con respecto a los parámetros. A continuación, actualizamos los parámetros en la dirección que pueda reducir la pérdida.

El código siguiente aplica la actualización, dado un conjunto de parámetros, una tasa de aprendizaje **lr**. Dado que nuestra pérdida se calcula como una media sobre el mini-lote, no necesitamos ajustar la tasa de aprendizaje en función del tamaño del lote.

*Actualizamos los parámetros en el método **step**. El método **zero_grad** establece todos los gradientes a 0, que debe ejecutarse antes de un paso de retropropagación.*

```python
class SGD(d2l.HyperParameters):  #@save
    """Minibatch stochastic gradient descent."""
```

```
    def __init__(self, params, lr):
        self.save_hyperparameters()

    def step(self):
        for param in self.params:
            param -= self.lr * param.grad

    def zero_grad(self):
        for param in self.params:
            if param.grad is not None:
                param.grad.zero_()
```

A continuación, definimos el método **configure_optimizers**, que devuelve una instancia de la clase SGD.

```
@d2l.add_to_class(LinearRegressionScratch)  #@save
def configure_optimizers(self):
    return SGD([self.w, self.b], self.lr)
```

3.4.4.- Entrenamiento

Ahora que tenemos todas las piezas en su lugar (parámetros, función de pérdida, modelo y optimizador), estamos listos para implementar el bucle de entrenamiento principal. En cada '***epoch***', iteramos a través de todo el conjunto de datos de entrenamiento, pasando una vez por cada ejemplo (suponiendo que el número de ejemplos es divisible por el tamaño del lote). En cada iteración, tomamos un mini-lote de ejemplos de entrenamiento, y calculamos su pérdida a través del método training_step del modelo. A continuación, calculamos los gradientes con respecto a cada parámetro. Finalmente, llamaremos al algoritmo de optimización para actualizar los parámetros del modelo. En resumen, ejecutaremos el siguiente bucle:

- Inicializar parámetros (w,b)
- Repetir hasta terminar
    - Calcular el gradiente g
    - Actualizar parámetros (w,b) <- (w,b)

En la mayoría de los casos querremos un conjunto de datos de validación para medir la calidad de nuestro modelo. Aquí pasamos el cargador de datos de validación una vez en cada ***epoch*** (época) para medir el rendimiento del modelo. Siguiendo nuestro diseño orientado a objetos, los métodos prepare_batch y fit_epoch están registrados en la clase d2l.Trainer (introducida en la Sección 3.2.4).

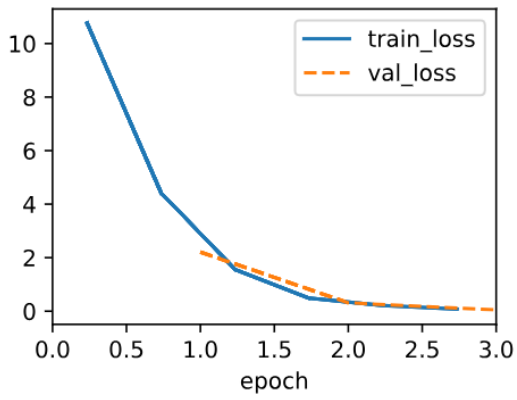
```
@d2l.add_to_class(d2l.Trainer)  #@save
def prepare_batch(self, batch):
    return batch

@d2l.add_to_class(d2l.Trainer)  #@save
def fit_epoch(self):
    self.model.train()
    for batch in self.train_dataloader:
        loss = self.model.training_step(self.prepare_batch(batch))
```

```
        self.optim.zero_grad()
        with torch.no_grad():
            loss.backward()
            if self.gradient_clip_val > 0:  # To be discussed later
                self.clip_gradients(self.gradient_clip_val, self.model)
            self.optim.step()
        self.train_batch_idx += 1
    if self.val_dataloader is None:
        return
    self.model.eval()
    for batch in self.val_dataloader:
        with torch.no_grad():
            self.model.validation_step(self.prepare_batch(batch))
        self.val_batch_idx += 1
```

Estamos casi listos para entrenar el modelo, pero primero necesitamos algunos datos de entrenamiento. Aquí usamos la clase SyntheticRegressionData y le pasamos algunos parámetros de verdad. A continuación, entrenamos nuestro modelo con la tasa de aprendizaje lr=0,03 y establecemos max_epochs=3. Tenga en cuenta que, en general, tanto el número de épocas como la tasa de aprendizaje son hiperparámetros. En general, establecer hiperparámetros es complicado y normalmente querremos utilizar una división en tres, un conjunto para el entrenamiento, un segundo para la selección de hiperparámetros, y el tercero reservado para la evaluación final.

```
model = LinearRegressionScratch(2, lr=0.03)
data = d2l.SyntheticRegressionData(w=torch.tensor([2, -3.4]), b=4.2)
trainer = d2l.Trainer(max_epochs=3)
trainer.fit(model, data)
```



Comentado [JT2]: **train_loss**: Representa la pérdida durante el entrenamiento del modelo en cada época. Esta pérdida se calcula utilizando los datos de entrenamiento y se utiliza para actualizar los parámetros del modelo durante el proceso de entrenamiento. El objetivo es minimizar esta pérdida para que el modelo se ajuste mejor a los datos de entrenamiento.

val_loss: Representa la pérdida durante la validación del modelo en cada época. Esta pérdida se calcula utilizando un conjunto de datos de validación separado, que no se utiliza para entrenar el modelo. La validación se realiza para evaluar el rendimiento del modelo en datos que no ha visto durante el entrenamiento y para detectar posibles problemas de sobreajuste. El objetivo es minimizar también esta pérdida, lo que indica que el modelo generaliza bien a nuevos datos.

3.6.- Generalización (Overfitting y Underfitting)

3.6.1.- Error de entrenamiento y error de generalización

- Error de entrenamiento (Remp): Es una métrica que muestra qué tan bien se ajusta tu modelo a los datos que has utilizado para entrenarlo. (Error de entrenamiento bajo -> modelo capaz de ajustarse a etiquetas arbritarias).
- Error de generalización (R): Este es como el error que esperarías ver si aplicaras tu modelo a un número infinito de nuevos datos que provienen de la misma fuente que tus datos de entrenamiento.
- Diferencia entre ambos: El error de entrenamiento se calcula usando solo los datos de entrenamiento, mientras que el error de generalización se calcula considerando toda la población subyacente.
- Problema central: Nunca podemos calcular el error de generalización exactamente, ya que no conocemos la verdadera distribución de los datos. Además, no podemos tener un flujo infinito de datos para probar nuestro modelo.
- Cómo se estima el error de generalización en la práctica: Utilizamos un conjunto de pruebas independiente, que es una selección aleatoria de datos del conjunto de entrenamiento que no se usaron durante el entrenamiento del modelo. Aplicamos el modelo a este conjunto de pruebas para estimar su rendimiento.
- Importancia de la generalización: La cuestión clave es cuándo podemos esperar que el error de entrenamiento sea una buena aproximación del error de generalización. Esto es fundamental para asegurarnos de que nuestro modelo funcione bien en datos nuevos y no solo en los datos de entrenamiento.

3.6.2.- ¿Underfitting o Overfitting?

Cuando miramos los errores de entrenamiento y validación, hay dos situaciones clave que debemos tener en cuenta:

1.- **Underfitting**: Si los errores de entrenamiento y validación son altos, pero apenas hay diferencia entre ellos, podría significar que nuestro modelo es demasiado simple para capturar el patrón que intentamos modelar. En este caso, podríamos necesitar un modelo más complejo para mejorar la predicción.

2.- **Overfitting**: Cuando el error de entrenamiento es mucho más bajo que el error de validación, indica que el modelo se ha ajustado demasiado a los datos de entrenamiento y no generaliza bien a nuevos datos. Aunque el sobreajuste no siempre es malo, ya que algunos modelos complejos pueden funcionar mejor en los datos de entrenamiento, lo importante es reducir el error de generalización, preocupándonos por la brecha (Remp - R) solo si dificulta este objetivo.

Si el error de entrenamiento es cero, la brecha de generalización es igual al error de generalización, y solo podemos avanzar reduciendo esta brecha.

Comentado [JT3]: Imagina que estás enseñando a un robot a reconocer gatos en imágenes. Tienes un conjunto de datos de 1000 imágenes de gatos, cada una etiquetada como "gato" o "no gato". Este es tu conjunto de datos de entrenamiento.

1. **Error de entrenamiento (Remp)**: Después de entrenar al robot con estas 1000 imágenes, calculas que comete errores en 50 imágenes. Entonces, tu error de entrenamiento sería del 5% (50 errores de clasificación de 1000 imágenes).

2. **Error de generalización (R)**: Ahora, imagina que tienes un flujo infinito de imágenes de gatos que nunca antes has visto, y le pides al robot que las clasifique. El error de generalización sería el porcentaje de estas nuevas imágenes de gatos que el robot clasifica incorrectamente.

Entonces, la diferencia clave es que el error de entrenamiento se basa en el conjunto de datos de entrenamiento específico que tienes, mientras que el error de generalización se refiere a cómo se desempeñaría el robot en un conjunto infinito de datos nuevos de la misma naturaleza que los datos de entrenamiento.

En resumen, el error de entrenamiento te dice cómo le está yendo al robot con los datos con los que lo entrenaste, mientras que el error de generalización te dice qué tan bien podría hacer el robot con datos completamente nuevos que nunca antes ha visto.

Comentado [JT4]: El modelo no logra capturar la estructura subyacente de los datos (es decir, los patrones o las relaciones fundamentales presentes en los datos).

Comentado [JT5]: El modelo se ajusta demasiado a los datos de entrenamiento capturando incluso ruido o las fluctuaciones irrelevantes en los datos.

Comentado [JT6]: Cuantas menos muestras tengamos en el conjunto de datos de entrenamiento mas probabilidad de overfitting

5.1.- Perceptrón Multicapa

El perceptrón multicapa (MLP por sus siglas en inglés) es una arquitectura de redes neuronales profundas que consiste en una sucesión de capas totalmente conectadas separadas por funciones de activación.

5.1.1.- Capas Ocultas

En comparación con modelos lineales simples, los perceptrones multicapa (redes neuronales) introducen capas ocultas que permiten aprender representaciones más complejas de los datos. Mientras que los modelos lineales operan bajo la suposición de linealidad, las capas ocultas permiten capturar relaciones no lineales entre las características de entrada y las salidas deseadas.

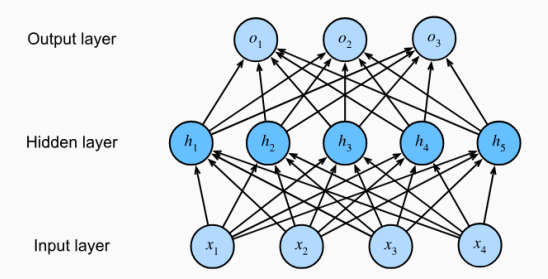
5.1.1.1- Limitaciones de los modelos

Los modelos lineales asumen una relación lineal entre las características de entrada y las salidas. Sin embargo, esta suposición puede ser demasiado restrictiva en muchos casos. Por ejemplo, la monotonicidad (aumento o disminución constante en la salida ante cambios en las características de entrada) puede no ser válida en todas las situaciones.

En problemas de clasificación de imágenes, la no linealidad es aún más evidente. La relación entre los píxeles de una imagen y su categoría (por ejemplo, perro o gato) no es lineal y puede ser altamente compleja. La confianza en un modelo lineal para esta tarea es ingenua, ya que no puede capturar adecuadamente las relaciones espaciales entre los píxeles que son esenciales para la clasificación precisa de imágenes.

5.1.1.2.- Incorporando capas ocultas

Podemos superar las limitaciones de los modelos lineales incorporando una o más capas ocultas. La forma más sencilla de hacerlo es apilar muchas capas completamente conectadas una encima de la otra. Cada capa alimenta a la siguiente capa, con la última capa produciendo las salidas. Las primeras L-1 capas se consideran la representación de los datos, mientras que la última capa se utiliza para realizar la predicción lineal. Esta arquitectura es comúnmente conocida como MLP.



5.1.2.- Funciones de activación

Son operadores diferenciables que deciden si una neurona debe activarse o no al calcular la suma ponderada y agregar un sesgo adicional. Introducen no linealidad en las redes neuronales, lo que es crucial para el aprendizaje profundo.

5.1.2.1.- Función ReLU

- Es una función de activación no lineal muy popular, que retorna cero para valores negativos y el mismo valor para valores positivos.
- ReLU (x) = max(0, x)
- La función ReLU es muy eficiente computacionalmente y permite un entrenamiento más rápido de redes neuronales profundas debido a su simplicidad.
- Sin embargo, puede presentar el problema conocido como "dying ReLU", donde algunas neuronas pueden quedar inactivas y no actualizar sus pesos durante el entrenamiento, lo que puede afectar negativamente al rendimiento de la red.

5.1.2.- Función Sigmoide

- La función sigmoide toma cualquier valor real como entrada y produce una salida en el rango de 0 a 1.
- Sigmoid(x) = 1 / 1 + exp(-x)
- Es una función suave y diferenciable, lo que la hace útil en el entrenamiento de redes neuronales mediante técnicas de optimización como el descenso de gradiente.
- Sin embargo, puede sufrir del problema de desvanecimiento del gradiente cuando se utilizan muchas capas en una red profunda, lo que puede dificultar el entrenamiento.

5.1.2.- Función Tanh

- Similar a la función sigmoide, pero produce una salida en el rango de -1 a 1.
- tanh(x) = 1 – exp(-2x) / 1 + exp(-2x)
- Al igual que la función sigmoide, es suave y diferenciable, lo que la hace útil en el entrenamiento de redes neuronales.
- A menudo se prefiere sobre la función sigmoide en capas ocultas de redes neuronales, ya que produce salidas centradas alrededor de cero, lo que puede ayudar a mantener la salida de la red en un rango manejable.

5.2.- Implementación de perceptrones multicapa

5.2.1.2.- Modelo

La implementación del MLP se realiza utilizando la clase Sequential de PyTorch, lo que nos permite definir de manera secuencial las capas del modelo de forma más compacta. Definimos una capa oculta y una capa de salida en una sola línea de código.

```
class MLP(d2l.Classifier):
    def __init__(self, num_outputs, num_hiddens, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(nn.Flatten(), nn.LazyLinear(num_hiddens),
                                 nn.ReLU(), nn.LazyLinear(num_outputs))
```

Comentado [JT7]: class MLP(d2l.Classifier):: Define una nueva clase llamada MLP que hereda de la clase d2l.Classifier. Esto significa que la clase MLP tendrá todas las propiedades y métodos de d2l.Classifier, además de cualquier modificación o adición que hagamos en la definición de la clase MLP.

def __init__(self, num_outputs, num_hiddens, lr):: Este es el método constructor de la clase MLP. Toma tres parámetros: num_outputs (número de unidades en la capa de salida), num_hiddens (número de unidades en la capa oculta) y lr (tasa de aprendizaje).

super().__init__(): Llama al método __init__() de la clase padre (d2l.Classifier). Esto es útil cuando queremos inicializar propiedades heredadas de la clase padre.

self.save_hyperparameters(): Guarda automáticamente los hiperparámetros (en este caso, num_outputs, num_hiddens y lr) para su uso posterior. Esto es útil para mantener un registro de los hiperparámetros utilizados durante el entrenamiento.

self.net = nn.Sequential(nn.Flatten(), nn.LazyLinear(num_hiddens), nn.ReLU(), nn.LazyLinear(num_outputs)): Define la red neuronal como una secuencia de capas utilizando nn.Sequential(). Las capas incluidas son:

nn.Flatten(): Esta capa se utiliza para aplanar los datos de entrada. Es útil cuando los datos de entrada tienen una forma de tensor multidimensional, como imágenes.
nn.LazyLinear(num_hiddens): Define una capa lineal (totalmente conectada) con num_hiddens unidades. La capa LazyLinear se utiliza para una inicialización diferida de los pesos, lo que puede ser útil en algunos casos de implementación.
nn.ReLU(): Define la función de activación ReLU que se aplicará después de la primera capa lineal.
nn.LazyLinear(num_outputs): Define la capa de salida lineal con num_outputs unidades.

5.2.1.3.- Training

Afortunadamente, el ciclo de entrenamiento para MLP es exactamente el mismo que para la regresión softmax. Definimos el modelo, los datos y el entrenador, luego finalmente invocamos el fit método en el modelo y los datos.

```
model = MLP(num_outputs=10, num_hiddens=256, lr=0.1)
data = d2l.FashionMNIST(batch_size=256)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```

6.1.- Capas y módulos

En el contexto de las redes neuronales, inicialmente nos centramos en modelos simples con una sola neurona que toma entradas, produce una salida y tiene parámetros ajustables. Más tarde, al considerar redes con múltiples salidas, utilizamos capas completas de neuronas, que funcionan de manera similar, pero con entradas y salidas vectorizadas. Incluso al introducir modelos más complejos como los MLP, estos mantienen la misma estructura básica.

Para construir redes neuronales más grandes y complejas, surgió la necesidad de hablar sobre componentes más grandes que una sola capa, pero más pequeños que el modelo completo. Para abordar esto, introducimos el concepto de módulos de red neuronal, que pueden describir desde una sola capa hasta el modelo completo. Los módulos se pueden combinar de forma recursiva para crear artefactos más grandes. Esto simplifica la implementación de redes complejas, ya que los módulos pueden definirse con métodos para la propagación hacia adelante y hacia atrás, y encapsulan los parámetros necesarios.

```
import torch
from torch import nn
from torch.nn import functional as F

net = nn.Sequential(nn.LazyLinear(256), nn.ReLU(), nn.LazyLinear(10))

X = torch.rand(2, 20)
net(X).shape
```

```
torch.Size([2, 10])
```

En el código, implementamos un MLP con una capa oculta de 256 unidades con activación ReLU, seguida de una capa de salida con diez unidades. Esto se logra mediante la clase nn.Sequential, que organiza las capas en orden y las ejecuta secuencialmente. Cada capa está representada por una instancia de la clase Module, con métodos simples para propagar la entrada hacia adelante. La invocación del modelo se realiza a través de net(X), que es una abreviatura de net.__call__(X).

6.1.1.- Un módulo personalizado

Para entender cómo funciona un módulo, es útil implementarlo desde cero. Un módulo debe cumplir varias funciones básicas:

1. Aceptar datos de entrada para la propagación hacia adelante.
2. Generar una salida basada en los datos de entrada.
3. Calcular el gradiente de la salida con respecto a la entrada.
4. Almacenar y proporcionar acceso a los parámetros necesarios.
5. Inicializar los parámetros del modelo según sea necesario.

En el siguiente código, creamos un módulo personalizado para un MLP con una capa oculta de 256 unidades y una capa de salida de 10 dimensiones. La clase MLP hereda de nn.Module y define un constructor y un método de propagación hacia adelante.

```
class MLP(nn.Module):
    def __init__(self):
        # Call the constructor of the parent class nn.Module to perform
        # the necessary initialization
        super().__init__()
        self.hidden = nn.LazyLinear(256)
        self.out = nn.LazyLinear(10)

    # Define the forward propagation of the model, that is, how to return the
    # required model output based on the input X
    def forward(self, X):
        return self.out(F.relu(self.hidden(X)))
```

El método de propagación hacia adelante toma la entrada X, calcula la representación oculta con una función de activación aplicada, y genera los logits (son las salidas no normalizadas de un modelo, es decir, los valores que se generan directamente a partir de la última capa antes de aplicar la función de activación final). Las capas del MLP se instancian en el constructor y se invocan en cada llamada al método forward.

Una ventaja clave de la abstracción del módulo es su versatilidad. Podemos usarla para crear capas individuales, modelos completos o componentes de complejidad intermedia. Esta versatilidad se explora en los próximos capítulos, como cuando trabajamos con redes neuronales convolucionales.

6.1.3.- Ejecutar código en el método de propagación hacia delante

La clase Sequential simplifica la creación de modelos, pero a veces necesitamos más flexibilidad para definir nuestras propias arquitecturas. Podemos ejecutar código Python personalizado en el método de propagación hacia adelante y realizar operaciones matemáticas arbitrarias. Por ejemplo, podríamos querer incorporar parámetros constantes en nuestras capas, como en el caso de la clase FixedHiddenMLP.

```
class FixedHiddenMLP(nn.Module):
    def __init__(self):
        super().__init__()
        # Random weight parameters that will not compute gradients and
        # therefore keep constant during training
        self.rand_weight = torch.rand((20, 20))
        self.linear = nn.LazyLinear(20)

    def forward(self, X):
        X = self.linear(X)
        X = F.relu(X @ self.rand_weight + 1)
        # Reuse the fully connected layer. This is equivalent to sharing
        # parameters with two fully connected layers
        X = self.linear(X)
        # Control flow
        while X.abs().sum() > 1:
            X /= 2
        return X.sum()
```

Esta clase implementa una capa oculta con pesos inicializados aleatoriamente que permanecen constantes durante el entrenamiento. Después de pasar la salida de esta capa por una capa completamente conectada, ejecutamos un bucle while que divide el vector de salida por 2 hasta que cumple una condición específica. Esta operación no es común en las redes neuronales estándar, pero ilustra cómo integrar código personalizado en el flujo de cálculo de la red.

La clase NestMLP anida módulos de manera creativa, utilizando una combinación de capas secuenciales y capas lineales. Por ejemplo, la función forward de NestMLP primero pasa los datos de entrada a través de una secuencia de capas lineales y activaciones ReLU, y luego los pasa por una capa lineal final.

```
class NestMLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(nn.LazyLinear(64), nn.ReLU(),
                                 nn.LazyLinear(32), nn.ReLU())
        self.linear = nn.LazyLinear(16)

    def forward(self, X):
        return self.linear(self.net(X))

chimera = nn.Sequential(NestMLP(), nn.LazyLinear(20), FixedHiddenMLP())
chimera(X)
```

Podemos combinar diferentes módulos de formas diversas, como se muestra en el ejemplo con la variable chimera, que contiene una instancia de NestMLP, seguida de una

capa lineal y una instancia de FixedHiddenMLP. Esto demuestra cómo podemos crear arquitecturas más complejas ensamblando varios módulos de manera creativa.

6.2.- Gestión de parámetros

Una vez configurada la arquitectura y los hiperparámetros de nuestra red neuronal, nos adentramos en el ciclo de entrenamiento. Durante este proceso, nuestro objetivo es minimizar la función de pérdida encontrando valores óptimos para los parámetros del modelo. Posteriormente, estos parámetros serán esenciales para realizar predicciones futuras y pueden ser útiles para otros propósitos, como la reutilización del modelo en diferentes contextos o su análisis científico.

Aunque en la mayoría de los casos confiamos en los frameworks de aprendizaje profundo para manejar los detalles de declaración y manipulación de parámetros, a veces es necesario adentrarse en estos aspectos, especialmente al trabajar con arquitecturas no estándar. En esta sección, nos enfocamos en dos aspectos clave:

6.2.1.- Acceso a parámetros

Para depuración, diagnóstico y visualizaciones, es importante poder acceder a los parámetros de nuestro modelo. Esto se puede lograr de manera sencilla cuando se utiliza una estructura secuencial como en el caso de `torch.nn.Sequential`. Por ejemplo, podemos acceder a los parámetros de una capa específica utilizando la indexación del modelo y examinar sus atributos.

```
net[2].state_dict()
```

```
OrderedDict([('weight',
              tensor([[-0.1649,  0.0605,  0.1694, -0.2524,  0.3526, -0.3414, -
0.2322,  0.0822]])),
             ('bias', tensor([0.0709]))])
```

6.2.2.- Parámetros compartidos

En ocasiones, deseamos compartir parámetros entre diferentes componentes de nuestro modelo. Esto se puede lograr asignando los mismos parámetros a varias capas. Por ejemplo, podemos configurar los parámetros de una capa para que sean los mismos que los de otra capa, lo que resulta útil en ciertos escenarios de diseño de redes.

```
shared = nn.LazyLinear(8)
net = nn.Sequential(
    nn.LazyLinear(8),
    nn.ReLU(),
    shared,
    nn.ReLU(),
    shared,
    nn.ReLU(),
    nn.LazyLinear(1)
)
```