گزارش پروژه پیاده‌سازی MLP از پایه

شبکه‌های عصبی از جمله روش‌های پرکاربرد و موثر در حل مسائل مختلف هستند که به صورت مستقیم تحت تأثیر الگوهای عصبی انسان قرار می‌گیرند. یکی از انواع شبکه‌های عصبی، شبکه‌های عصبی چند لایه (MLP) هستند که از چندین لایه از نورون‌ها به عنوان واحدهای پردازش استفاده می‌کنند تا الگوهای پیچیده را به خوبی توصیف کنند و پیش‌بینی‌های دقیق‌تری را ارائه دهند.

معرفی کلاس‌ها و توابع اصلی

در این پروژه، از چندین کلاس و تابع اصلی استفاده شده است که در ادامه به بررسی دقیق‌تر کدها و نحوه عملکرد آنها می‌پردازیم:

کلاس **Tensor**:

کلاس Tensor یک ساختار داده چند بعدی است که از مقادیر عددی وابسته به یک عملیات مشتق‌پذیر (مانند ضرب، جمع، توان و ...) و مقادیر گرادیان برای محاسبات مشتقات استفاده می‌کند. در این کلاس، ما عملیات ریاضی مختلف را برای این ساختار داده پیاده‌سازی کرده‌ایم تا بتوانیم به صورت ساده و کارآمد عملیات مختلف را روی داده‌های ما انجام دهیم و گرادیان‌ها را محاسبه کنیم.

__init__:

value: مقدار عددی مربوط به تنسور.

label: برچسب مربوط به تنسور (مورد استفاده در صورت نیاز).

children: لیستی از فرزندان تنسور در گراف محاسباتی (تنسورهایی که به عنوان ورودی در عملیات محاسباتی مورد استفاده قرار می‌گیرند).

operator: نوع عملیات مشتق‌پذیری که بر روی تنسور انجام شده است.

grad: مقدار گرادیان مربوط به تنسور (پیش فرض صفر است).

_backward: تابعی که مشتق‌های مورد نیاز را محاسبه می‌کند (پیش فرض آن یک تابع خالی است).

__repr__:

این متد برای نمایش مقدار و گرادیان تنسور استفاده می‌شود.

__mul__، __add__، __sub__، __rsub__، __pow__:

این متدها عملیات ضرب، جمع، تفریق و توان بر روی تنسورها را پیاده‌سازی می‌کنند.

برای هر عملیات، مقدار جدید تنسور و مشتقات مربوط به آن عملیات محاسبه می‌شوند و در تنسورهای والدین ذخیره می‌شوند.

backward:

این متد گرادیان‌های مربوط به عملیات‌های مشتق‌پذیر را محاسبه می‌کند.

برای این کار، از روش بازگشتی بر روی گراف محاسباتی استفاده می‌شود تا گرادیان‌ها به ترتیب محاسبه شوند.

این کلاس به عنوان یک بستر اصلی برای پیاده‌سازی مدل‌های عصبی و سایر الگوریتم‌های مرتبط با یادگیری عمیق استفاده می‌شود. از آن برای محاسبات مشتقات و بهبود عملکرد مدل‌ها با استفاده از الگوریتم‌های بهینه‌سازی مختلف نیز می‌توان استفاده کرد.

```
class Tensor:
    def __init__(self, value, label='', children=(),
operator=None):
        self.value = value
        self.children = set(children)
        self.operator = operator
        self.grad = 0  # Gradient of the tensor
        self._backward = lambda: None  # Lambda that does
nothing by default
        self.label = label

    def __repr__(self) -> str:
        return f"Tensor(value={self.value}, grad={self.grad})"

    def __mul__(self, other):
        other = other if isinstance(other, Tensor) else
Tensor(other)
```

```
        out = Tensor(self.value * other.value, children=(self,
other), operator='*')

        def backward():
            self.grad += other.value * out.grad
            other.grad += self.value * out.grad
        out._backward = backward

        return out

    def __add__(self, other):
        other = other if isinstance(other, Tensor) else
Tensor(other)
        out = Tensor(self.value + other.value, children=(self,
other), operator='+')

        def backward():
            # For addition, the gradient of input wrt output is
1
            self.grad += 1 * out.grad
            other.grad += 1 * out.grad
        out._backward = backward

        return out

    def __sub__(self, other):
        other = other if isinstance(other, Tensor) else
Tensor(other)
        out = Tensor(self.value - other.value, children=(self,
other), operator='-')

        def backward():
            self.grad += 1 * out.grad
            other.grad -= 1 * out.grad
        out._backward = backward

        return out

    def __rsub__(self, other):

        return -self + other

    def __pow__(self, other):
        out = Tensor(self.value ** other, children=(self,),
operator='**')
```

```
        def backward():
            self.grad += other * (self.value ** (other - 1)) *
out.grad
        out._backward = backward

        return out

    def __radd__(self, other):
        # Support adding tensors on the right side of numbers
        return self + other

    def __rmul__(self, other):
        # Support multiplying tensors on the right side of
numbers
        return self * other

    def backward(self, grad=1):

        self.grad = grad  # Initialize the gradient

        topo_order = []

        def toposort(tensor):
            if tensor not in topo_order:
                for child in tensor.children:
                    toposort(child)
                topo_order.append(tensor)

        toposort(self)

        # Reverse topo_order for correct backward pass execution
        for tensor in reversed(topo_order):
            tensor._backward()
```

کلاس Neuron:

کلاس Neuron یک نورون در یک شبکه عصبی را پیاده‌سازی می‌کند که قادر است وزن‌ها و بایاس را به صورت خودکار مقداردهی اولیه کند و همچنین دارای توابعی برای محاسبه خروجی نورون است.

__init__:

input_size: تعداد ورودی‌های نورون.

weights: لیستی از تنسورهای وزن ورودی برای هر نورون.

bias: تنسوری که بایاس نورون را نشان می‌دهد.

forward:

x: لیستی از ورودی‌های نورون.

محاسبه مجموع ضرب وزن‌های ورودی در مقادیر ورودی به صورت مجزا و سپس جمع کردن آن‌ها.

اعمال تابع فعال‌سازی تانژانت هیپربولیک (از کلاس F) بر روی مجموع وزن‌ها با بایاس.

__call__:

تابعی برای اجرای محاسبات forward و برگرداندن خروجی نورون، به گونه‌ای که نورون قابل فراخوانی مستقیم باشد.

parameters:

بازگرداندن لیستی شامل تمام وزن‌ها و بایاس به عنوان پارامترهای قابل آموزش نورون.

این کلاس برای پیاده‌سازی لایه‌های مختلف در شبکه‌های عصبی و همچنین برای ایجاد مدل‌های یادگیری عمیق مورد استفاده قرار می‌گیرد و از آن برای ایجاد توانایی یادگیری و تطبیق مدل با داده‌های ورودی مختلف استفاده می‌شود.

```
class Neuron:

    def __init__(self, input_size):
        self.weights = [Tensor(random.uniform(-1,1)) for i in
range(input_size)]  # Initialize weights to Tensor(1)
        self.bias = Tensor(random.uniform(-1,1))  # Initialize
bias to Tensor(5)

    def forward(self, x):
        res = sum([w_i * x_i for w_i, x_i in zip(self.weights,
x)])  # Compute weighted sum
        return F.tanh(res + self.bias)  # Add bias

    def __call__(self, x):
        return self.forward(x)  # Make instance callable

    def parameters(self):
        # Return all the weights and bias as a list
        return self.weights + [self.bias]
```

کلاس F:

کلاس F حاوی توابع فعال‌سازی است که در شبکه‌های عصبی استفاده می‌شوند. در اینجا، تابع فعال‌سازی تانژانت هیپربولیک (tanh) پیاده‌سازی شده است.

tanh:

x: یک تنسور که مقدار آن به عنوان ورودی به تابع تانژانت هیپربولیک داده می‌شود.

محاسبه مقدار تابع تانژانت هیپربولیک بر اساس فرمول مربوطه.

ایجاد یک تنسور جدید که مقدار آن برابر با مقدار تابع تانژانت هیپربولیک محاسبه شده است.

تعیین تابع محاسبه مشتق (backward) که مشتق ورودی نسبت به مقدار تانژانت هیپربولیک را محاسبه می‌کند و به مشتق ورودی اضافه می‌کند.

این کلاس برای اعمال توابع فعال‌سازی مختلف به خروجی لایه‌های شبکه عصبی و همچنین محاسبه مشتقات مربوط به این توابع در فرایند بهینه‌سازی استفاده می‌شود.

```
class F:
    @staticmethod
    def tanh(x: Tensor) -> Tensor:
        # Compute tanh using the provided formula
        output_value = (math.exp(x.value) - math.exp(-x.value))
/ (math.exp(x.value) + math.exp(-x.value))
        out = Tensor(output_value, children=(x,),
operator='tanh')

        def backward():
            x.grad += (1 - out.value ** 2) * out.grad

        out._backward = backward

        return out
```

کلاس Layer:

کلاس Layer به عنوان یک لایه در شبکه عصبی عمل می‌کند و شامل یک تعداد مشخص از نورون‌ها است.

input_size: اندازه ورودی لایه، یعنی تعداد ویژگی‌های ورودی به هر نورون.

output_size: اندازه خروجی لایه، یعنی تعداد نورون‌های موجود در لایه.

neurons: لیستی از نورون‌ها که به تعداد output_size ساخته شده‌اند و هر کدام از آن‌ها یک نمونه از کلاس Neuron هستند.

forward(x):

x: ورودی به لایه، یک تنسور یا بردار ویژگی‌ها.

محاسبه خروجی لایه با فراخوانی هر نورون با ورودی x و ذخیره نتایج در یک لیست.

اگر تنها یک خروجی وجود داشته باشد، آن را به عنوان خروجی برمی‌گرداند؛ در غیر این صورت، لیستی از خروجی‌ها را برمی‌گرداند.

__call__(x):

به منظور استفاده از لایه به عنوان یک تابع قابل فراخوانی (callable) تعریف شده است.

ورودی x را به عنوان ورودی به تابع forward ارسال می‌کند.

parameters():

بازگرداندن لیستی شامل تمام وزن‌ها و بایاس‌های موجود در نورون‌های لایه.

برای هر نورون در لایه، وزن‌ها و بایاس‌های آن را به لیست params اضافه می‌کند.

در نهایت، لیست params را باز می‌گرداند.

این کلاس برای ایجاد و مدیریت لایه‌های مختلف در یک شبکه عصبی استفاده می‌شود و وظیفه‌اش اجرای عملیات فوروارد و بازگشتی برای محاسبه خطا و به‌روزرسانی وزن‌ها در فرآیند آموزش است.

```python
class Layer:

  def __init__(self, input_size, output_size):
    self.neurons = [Neuron(input_size) for _ in
range(output_size)]

  def forward(self, x):
    out = [neuron(x) for neuron in self.neurons]
    return out[0] if len(out)==1 else out

  def __call__(self, x):
    return self.forward(x)

  def parameters(self):
    params = []
    for neuron in self.neurons:
      params+=neuron.parameters()
    return params
```

کلاس MLP:

کلاس MLP به عنوان یک شبکه عصبی چندلایه (Multi-Layer Perceptron) عمل می‌کند و شامل یک تعداد لایه‌های مختلف است.

input_size: اندازه ورودی شبکه، یعنی تعداد ویژگی‌های ورودی به شبکه.

layer_sizes: لیستی از اندازه‌های لایه‌ها، یعنی تعداد نورون‌ها در هر لایه.

layers: لیستی از لایه‌های شبکه که به ترتیب از ورودی تا خروجی ساخته می‌شوند.

__init__(input_size, layer_sizes):

ایجاد لایه‌های شبکه با تعداد و اندازه مشخص شده.

layers_total: لیستی که شامل تعداد ورودی و اندازه‌های لایه‌ها می‌شود.

برای هر لایه، یک نمونه از کلاس Layer با استفاده از اندازه ورودی و خروجی مربوطه ساخته می‌شود.

forward(x):

x: ورودی به شبکه، یک تنسور یا بردار ویژگی‌ها.

اعمال فرایند فوروارد به ترتیب برای هر لایه از شبکه با فراخوانی تابع forward هر لایه.

نتیجه خروجی شبکه را برمی‌گرداند.

__call__(x):

استفاده از شبکه به عنوان یک تابع قابل فراخوانی (callable).

ورودی x را به عنوان ورودی به تابع forward ارسال می‌کند.

parameters():

بازگرداندن لیستی شامل تمام وزن‌ها و بایاس‌های موجود در تمام لایه‌های شبکه.

برای هر لایه، لیست پارامترهای آن را از تابع parameters آن لایه استخراج کرده و به لیست params اضافه می‌کند.

در نهایت، لیست params را باز می‌گرداند.

کلاس MLP برای ایجاد و مدیریت یک شبکه عصبی چندلایه استفاده می‌شود و تمامی عملیات مربوط به فوروارد و بازگشتی در آن پیاده‌سازی شده است.

```
class MLP:
    def __init__(self, input_size, layer_sizes):
        layers_total = [input_size] + layer_sizes
        self.layers = [Layer(layers_total[i], layers_total[i+1])
for i in range(len(layer_sizes))]

    def forward(self, x):
        for layer in self.layers:
            x = layer(x) # Use the layers as callable to perform
forward pass
        return x

    def __call__(self, x):
        return self.forward(x)

    def parameters(self):
        # Retrieve parameters from all layers
        params = []
        for layer in self.layers:
            params += layer.parameters()
        return params
```

تشکیل دیتاست فرضی:

این بخش به منظور مقداردهی اولیه به داده‌ها و برچسب‌ها و ایجاد یک نمونه از شبکه عصبی چندلایه مورد استفاده قرار می‌گیرد.

مقداردهی اولیه داده‌ها و برچسب‌ها:

X: ماتریس ویژگی‌ها یا داده‌های ورودی که به شبکه تغذیه می‌شود.

Y: برچسب‌های متناظر با هر نمونه در داده‌های ورودی.

تعداد ویژگی‌ها در هر نمونه برابر با تعداد ستون‌ها در ماتریس X است.

تعداد نورون‌ها در لایه خروجی برابر با ۱ است زیرا در اینجا از یک مدل طبقه‌بندی دو کلاسه استفاده شده است.

ایجاد نمونه از شبکه عصبی چندلایه:

input_size: تعداد ویژگی‌های ورودی به شبکه.

layer_sizes: لیستی از اندازه‌های لایه‌ها، که مشخص می‌کند تعداد نورون‌ها در هر لایه چقدر است.

model: یک نمونه از کلاس MLP با استفاده از تعداد ویژگی‌های ورودی و لایه‌های مشخص شده ساخته می‌شود.

به طور خلاصه، این بخش برای مقداردهی اولیه به داده‌ها و برچسب‌ها و ایجاد یک نمونه از شبکه عصبی چندلایه استفاده می‌شود.

```
X = [[-1,2,5,4],
     [4,2,6,3],
     [2,3,4,7],
     [7,2,4,1]]
Y = [1,-1,-1,1]
input_size = len(X) # Number of features in the input layer
layer_sizes = [4,2,1] # Number of neurons in each hidden and output layer
model = MLP(input_size, layer_sizes)
```

کلاس Optimizer :

این کلاس برای به‌روزرسانی پارامترهای مدل با استفاده از الگوریتم بهینه‌سازی Gradient Descent استفاده می‌شود.

متدها:

__init__(self, parameters, lr):

متغیرها:

parameters: یک لیست از پارامترهای مدل که باید به‌روزرسانی شوند.

lr: نرخ یادگیری برای استفاده در فرآیند به‌روزرسانی.

این متد در هنگام ایجاد یک نمونه از کلاس Optimizer فراخوانی می‌شود و پارامترهای لازم برای عملیات به‌روزرسانی پارامترها را مقداردهی اولیه می‌کند.

zero_grad(self):

این متد مقدار گرادیان تمام پارامترهای مدل را صفر می‌کند. این مرحله معمولاً قبل از شروع محاسبه گرادیان‌ها در هر دوره آموزش انجام می‌شود.

step(self):

این متد مقدار پارامترهای مدل را با توجه به گرادیان و نرخ یادگیری به‌روزرسانی می‌کند. برای هر پارامتر، مقدار جدید آن برابر با مقدار فعلی منهای محصول نرخ یادگیری و گرادیان آن است.

به طور خلاصه، کلاس Optimizer برای مدیریت بهینه‌سازی و به‌روزرسانی پارامترهای مدل در فرآیند آموزش استفاده می‌شود.

```
class Optimizer:
    def __init__(self, parameters, lr):
        self.parameters = list(parameters)
        self.lr = lr

    def zero_grad(self):
        for param in self.parameters:
            if param.grad is not None:
                param.grad = 0

    def step(self):
        for param in self.parameters:
            if param.grad is not None:
                param.value -= self.lr * param.grad
```

ساخت یک شی از کلاس بهینه سازی:

این بخش از کد برای ایجاد یک نمونه از کلاس Optimizer با استفاده از پارامترهای مدل و نرخ یادگیری مشخص شده است که در فرآیند آموزش مدل استفاده می‌شود.

```
optim = Optimizer(model.parameters(),0.2)
```

اجرای بهینه سازی گرادیان کاهشی:

```
n_epochs = 200

for epoch in range(n_epochs):

  for x,y in zip(X,Y):

    y_hat = model(x)
    loss = (y_hat-y)**2*len(X)**-1

    optim.zero_grad()

    loss.backward()

    optim.step()

  print(f"epoch: {epoch}: ",f"loss: {loss}")
```

n_epochs = 200: تعداد دوره‌های آموزش مدل را مشخص می‌کند.

for _ in range(n_epochs):: این حلقه برای اجرای تعداد دوره‌های مشخص شده از آموزش استفاده می‌شود.

for x, y in zip(X, Y):: با استفاده از تابع zip، هر داده و برچسب متناظر آن به صورت ترکیبی مورد استفاده قرار می‌گیرد.

y_hat = model(x): این بخش از کد، خروجی مدل برای داده ورودی مورد نظر را محاسبه می‌کند.

loss = (y_hat - y) ** 2 * len(X) ** -1: مقدار تابع هزینه را با استفاده از فرمول MSE که میانگین مجذور ارور ها است برای خروجی مدل و برچسب واقعی محاسبه می‌کند.

optim.zero_grad(): این بخش از کد، گرادیان‌های پارامترهای مدل را صفر می‌کند تا در دوره جدید محاسبات گرادیان از اول شروع شود.

loss.backward(): با فراخوانی این متد، گرادیان‌های تابع هزینه نسبت به پارامترهای مدل محاسبه می‌شوند.

optim.step(): در این بخش از کد، پارامترهای مدل با استفاده از گرادیان‌های محاسبه شده و نرخ یادگیری به روزرسانی می‌شوند.

print(loss): مقدار تابع هزینه در پایان هر دوره آموزش چاپ می‌شود.

این بخش از کد به صورت کامل فرآیند آموزش مدل با استفاده از روش بهینه‌سازی گرادیان نزولی را ارائه می‌دهد.

نتایج:

در این پروسه آموزش، مدل با استفاده از روش بهینه‌سازی گرادیان نزولی آموزش داده شد. در پایان 200 دوره آموزش، مقادیر تابع هزینه به ترتیب به شرح زیر بودند:

```
epoch: 0:  loss: Tensor(value=0.24672508022134787, grad=1)
epoch: 1:  loss: Tensor(value=0.24130503528839853, grad=1)
epoch: 2:  loss: Tensor(value=0.23534651493384845, grad=1)
epoch: 3:  loss: Tensor(value=0.22881733881999, grad=1)
epoch: 4:  loss: Tensor(value=0.22169231042266804, grad=1)
epoch: 5:  loss: Tensor(value=0.2139568436463607, grad=1)
epoch: 6:  loss: Tensor(value=0.20561115892982898, grad=1)
epoch: 7:  loss: Tensor(value=0.19667476051819915, grad=1)
epoch: 8:  loss: Tensor(value=0.18719068132979996, grad=1)
epoch: 9:  loss: Tensor(value=0.17722874344974557, grad=1)
epoch: 10:  loss: Tensor(value=0.1668869072354576, grad=1)
epoch: 11:  loss: Tensor(value=0.15628978145570208, grad=1)
epoch: 12:  loss: Tensor(value=0.14558365245958135, grad=1)
epoch: 13:  loss: Tensor(value=0.13492800698245452, grad=1)
epoch: 14:  loss: Tensor(value=0.12448437389877168, grad=1)
epoch: 15:  loss: Tensor(value=0.11440413223506624, grad=1)
epoch: 16:  loss: Tensor(value=0.10481738176712067, grad=1)
epoch: 17:  loss: Tensor(value=0.09582480409378766, grad=1)
epoch: 18:  loss: Tensor(value=0.08749367842488477, grad=1)
epoch: 19:  loss: Tensor(value=0.07985816036009395, grad=1)
epoch: 20:  loss: Tensor(value=0.0729229982848054, grad=1)
epoch: 21:  loss: Tensor(value=0.0666693487042278, grad=1)
epoch: 22:  loss: Tensor(value=0.061061317505155366, grad=1)
epoch: 23:  loss: Tensor(value=0.05605215809317711, grad=1)
epoch: 24:  loss: Tensor(value=0.05158948956206531, grad=1)
epoch: 25:  loss: Tensor(value=0.04761929168152635, grad=1)
epoch: 26:  loss: Tensor(value=0.04408870809533377, grad=1)
epoch: 27:  loss: Tensor(value=0.04094783673875419, grad=1)
epoch: 28:  loss: Tensor(value=0.038150736897377996, grad=1)
epoch: 29:  loss: Tensor(value=0.03565587383851222, grad=1)
epoch: 30:  loss: Tensor(value=0.033426185946905634, grad=1)
epoch: 31:  loss: Tensor(value=0.031428916031423206, grad=1)
epoch: 32:  loss: Tensor(value=0.02963530853614692, grad=1)
epoch: 33:  loss: Tensor(value=0.02802024193796847, grad=1)
epoch: 34:  loss: Tensor(value=0.026561841221166567, grad=1)
epoch: 35:  loss: Tensor(value=0.025241097958092127, grad=1)
epoch: 36:  loss: Tensor(value=0.02404151367823382, grad=1)
epoch: 37:  loss: Tensor(value=0.02294877441459453, grad=1)
epoch: 38:  loss: Tensor(value=0.021950459373321587, grad=1)
epoch: 39:  loss: Tensor(value=0.021035783668870853, grad=1)
epoch: 40:  loss: Tensor(value=0.02019537334761268, grad=1)
epoch: 41:  loss: Tensor(value=0.019421070032848993, grad=1)
epoch: 42:  loss: Tensor(value=0.018705762155603968, grad=1)
epoch: 43:  loss: Tensor(value=0.018043239683607187, grad=1)
epoch: 44:  loss: Tensor(value=0.017428069391940144, grad=1)
epoch: 45:  loss: Tensor(value=0.01685548794708333, grad=1)
epoch: 46:  loss: Tensor(value=0.0163213103476668, grad=1)
```

```
epoch: 47:  loss: Tensor(value=0.015821851547159038, grad=1)
epoch: 48:  loss: Tensor(value=0.015353859356751792, grad=1)
epoch: 49:  loss: Tensor(value=0.014914456980314846, grad=1)
epoch: 50:  loss: Tensor(value=0.0145010937625618, grad=1)
epoch: 51:  loss: Tensor(value=0.014111502934950613, grad=1)
epoch: 52:  loss: Tensor(value=0.013743665321843873, grad=1)
epoch: 53:  loss: Tensor(value=0.013395778123699444, grad=1)
epoch: 54:  loss: Tensor(value=0.01306622802673654, grad=1)
epoch: 55:  loss: Tensor(value=0.012753568002013486, grad=1)
epoch: 56:  loss: Tensor(value=0.012456497253531615, grad=1)
epoch: 57:  loss: Tensor(value=0.012173843857088175, grad=1)
epoch: 58:  loss: Tensor(value=0.011904549701180347, grad=1)
epoch: 59:  loss: Tensor(value=0.011647657400137396, grad=1)
epoch: 60:  loss: Tensor(value=0.011402298899426932, grad=1)
epoch: 61:  loss: Tensor(value=0.011167685535134047, grad=1)
epoch: 62:  loss: Tensor(value=0.010943099345137841, grad=1)
epoch: 63:  loss: Tensor(value=0.010727885459530823, grad=1)
epoch: 64:  loss: Tensor(value=0.010521445423204883, grad=1)
epoch: 65:  loss: Tensor(value=0.010323231324996523, grad=1)
epoch: 66:  loss: Tensor(value=0.010132740625961619, grad=1)
epoch: 67:  loss: Tensor(value=0.00994951159475528, grad=1)
epoch: 68:  loss: Tensor(value=0.009773119271164121, grad=1)
epoch: 69:  loss: Tensor(value=0.009603171889942488, grad=1)
epoch: 70:  loss: Tensor(value=0.009439307706550463, grad=1)
epoch: 71:  loss: Tensor(value=0.009281192174437835, grad=1)
epoch: 72:  loss: Tensor(value=0.009128515430382546, grad=1)
epoch: 73:  loss: Tensor(value=0.008980990050256066, grad=1)
epoch: 74:  loss: Tensor(value=0.008838349042607429, grad=1)
epoch: 75:  loss: Tensor(value=0.008700344051755982, grad=1)
epoch: 76:  loss: Tensor(value=0.00856674374577649, grad=1)
epoch: 77:  loss: Tensor(value=0.00843733236793161, grad=1)
epoch: 78:  loss: Tensor(value=0.008311908432841353, grad=1)
epoch: 79:  loss: Tensor(value=0.008190283551035004, grad=1)
epoch: 80:  loss: Tensor(value=0.008072281367567418, grad=1)
epoch: 81:  loss: Tensor(value=0.007957736602144718, grad=1)
epoch: 82:  loss: Tensor(value=0.0078464941797298, grad=1)
epoch: 83:  loss: Tensor(value=0.007738408441925322, grad=1)
epoch: 84:  loss: Tensor(value=0.007633342430583893, grad=1)
epoch: 85:  loss: Tensor(value=0.007531167236098853, grad=1)
epoch: 86:  loss: Tensor(value=0.007431761403705408, grad=1)
epoch: 87:  loss: Tensor(value=0.007335010391885734, grad=1)
epoch: 88:  loss: Tensor(value=0.007240806077641877, grad=1)
epoch: 89:  loss: Tensor(value=0.007149046303986503 6, grad=1)
epoch: 90:  loss: Tensor(value=0.007059634465515792, grad=1)
epoch: 91:  loss: Tensor(value=0.006972479128381764, grad=1)
epoch: 92:  loss: Tensor(value=0.006887493681379261, grad=1)
epoch: 93:  loss: Tensor(value=0.006804596015213902, grad=1)
epoch: 94:  loss: Tensor(value=0.006723708227327024 5, grad=1)
epoch: 95:  loss: Tensor(value=0.00664475634992781, grad=1)
epoch: 96:  loss: Tensor(value=0.006567670099125257, grad=1)
epoch: 97:  loss: Tensor(value=0.006492382643267523, grad=1)
epoch: 98:  loss: Tensor(value=0.006418830388786983, grad=1)
epoch: 99:  loss: Tensor(value=0.00634695278201938, grad=1)
epoch: 100:  loss: Tensor(value=0.006276692125615975, grad=1)
```

```
epoch: 101:  loss: Tensor(value=0.0062079934083023435, grad=1)
epoch: 102:  loss: Tensor(value=0.006140804146857593, grad=1)
epoch: 103:  loss: Tensor(value=0.006075074239295288, grad=1)
epoch: 104:  loss: Tensor(value=0.006010755828322786, grad=1)
epoch: 105:  loss: Tensor(value=0.005947803174242452, grad=1)
epoch: 106:  loss: Tensor(value=0.0058861725365344855, grad=1)
epoch: 107:  loss: Tensor(value=0.005825822063431255, grad=1)
epoch: 108:  loss: Tensor(value=0.005766711688854382, grad=1)
epoch: 109:  loss: Tensor(value=0.005708803036142639, grad=1)
epoch: 110:  loss: Tensor(value=0.005652059328048447, grad=1)
epoch: 111:  loss: Tensor(value=0.005596445302526982, grad=1)
epoch: 112:  loss: Tensor(value=0.005541927133882502, grad=1)
epoch: 113:  loss: Tensor(value=0.005488472358873931, grad=1)
epoch: 114:  loss: Tensor(value=0.005436049807415305, grad=1)
epoch: 115:  loss: Tensor(value=0.005384629537536986, grad=1)
epoch: 116:  loss: Tensor(value=0.005334182774301636, grad=1)
epoch: 117:  loss: Tensor(value=0.005284681852393387, grad=1)
epoch: 118:  loss: Tensor(value=0.005236100162122012, grad=1)
epoch: 119:  loss: Tensor(value=0.005188412098604252, grad=1)
epoch: 120:  loss: Tensor(value=0.0051415930139034685, grad=1)
epoch: 121:  loss: Tensor(value=0.005095619171926013, grad=1)
epoch: 122:  loss: Tensor(value=0.0050504677058883585, grad=1)
epoch: 123:  loss: Tensor(value=0.0050061165781833256, grad=1)
epoch: 124:  loss: Tensor(value=0.004962544542486771, grad=1)
epoch: 125:  loss: Tensor(value=0.004919731107958425, grad=1)
epoch: 126:  loss: Tensor(value=0.004877656505400999, grad=1)
epoch: 127:  loss: Tensor(value=0.004836301655251997, grad=1)
epoch: 128:  loss: Tensor(value=0.004795648137292055, grad=1)
epoch: 129:  loss: Tensor(value=0.00475567816196178, grad=1)
epoch: 130:  loss: Tensor(value=0.004716374543186862, grad=1)
epoch: 131:  loss: Tensor(value=0.00467772067261859, grad=1)
epoch: 132:  loss: Tensor(value=0.004639700495203166, grad=1)
epoch: 133:  loss: Tensor(value=0.004602298485999575, grad=1)
epoch: 134:  loss: Tensor(value=0.004565499628171101, grad=1)
epoch: 135:  loss: Tensor(value=0.00452928939208067, grad=1)
epoch: 136:  loss: Tensor(value=0.004493653715425355, grad=1)
epoch: 137:  loss: Tensor(value=0.0044585789843492, grad=1)
epoch: 138:  loss: Tensor(value=0.0044240520154779405, grad=1)
epoch: 139:  loss: Tensor(value=0.004390060038822798, grad=1)
epoch: 140:  loss: Tensor(value=0.0043565906815042745, grad=1)
epoch: 141:  loss: Tensor(value=0.004323631952249216, grad=1)
epoch: 142:  loss: Tensor(value=0.004291172226618875, grad=1)
epoch: 143:  loss: Tensor(value=0.004259200232926798, grad=1)
epoch: 144:  loss: Tensor(value=0.004227705038809126, grad=1)
epoch: 145:  loss: Tensor(value=0.0041966760384119, grad=1)
epoch: 146:  loss: Tensor(value=0.004166102940161936, grad=1)
epoch: 147:  loss: Tensor(value=0.004135975755090094, grad=1)
epoch: 148:  loss: Tensor(value=0.004106284785677991, grad=1)
epoch: 149:  loss: Tensor(value=0.004077020615200122, grad=1)
epoch: 150:  loss: Tensor(value=0.0040481740975361735, grad=1)
epoch: 151:  loss: Tensor(value=0.004019736347428662, grad=1)
epoch: 152:  loss: Tensor(value=0.003991698731163493, grad=1)
epoch: 153:  loss: Tensor(value=0.003964052857651662, grad=1)
epoch: 154:  loss: Tensor(value=0.003936790569892002, grad=1)
```

```
epoch: 155:  loss: Tensor(value=0.0039099903936795784, grad=1)
epoch: 156:  loss: Tensor(value=0.0038833852453553147, grad=1)
epoch: 157:  loss: Tensor(value=0.0038572269931393794, grad=1)
epoch: 158:  loss: Tensor(value=0.0038314218810998756, grad=1)
epoch: 159:  loss: Tensor(value=0.0038059628066674083, grad=1)
epoch: 160:  loss: Tensor(value=0.0037808428857168678, grad=1)
epoch: 161:  loss: Tensor(value=0.0037560553034119355, grad=1)
epoch: 162:  loss: Tensor(value=0.0037315935936612366, grad=1)
epoch: 163:  loss: Tensor(value=0.0037074513477540144, grad=1)
epoch: 164:  loss: Tensor(value=0.0036836223514906774, grad=1)
epoch: 165:  loss: Tensor(value=0.0036601005512387824, grad=1)
epoch: 166:  loss: Tensor(value=0.0036368800487481786, grad=1)
epoch: 167:  loss: Tensor(value=0.0036139550961675466, grad=1)
epoch: 168:  loss: Tensor(value=0.0035913200912530754, grad=1)
epoch: 169:  loss: Tensor(value=0.0035689695727607034, grad=1)
epoch: 170:  loss: Tensor(value=0.0035468982160137188, grad=1)
epoch: 171:  loss: Tensor(value=0.0035251008286376795, grad=1)
epoch: 172:  loss: Tensor(value=0.003503572346455594, grad=1)
epoch: 173:  loss: Tensor(value=0.0034823078295360466, grad=1)
epoch: 174:  loss: Tensor(value=0.0034613024583877324, grad=1)
epoch: 175:  loss: Tensor(value=0.0034405515302940224, grad=1)
epoch: 176:  loss: Tensor(value=0.0034200504557815574, grad=1)
epoch: 177:  loss: Tensor(value=0.0033997947552170444, grad=1)
epoch: 178:  loss: Tensor(value=0.0033797800555270003, grad=1)
epoch: 179:  loss: Tensor(value=0.0033600020870349994, grad=1)
epoch: 180:  loss: Tensor(value=0.0033404566804117286, grad=1)
epoch: 181:  loss: Tensor(value=0.0033211397637330464, grad=1)
epoch: 182:  loss: Tensor(value=0.0033020473596415043, grad=1)
epoch: 183:  loss: Tensor(value=0.0032831755826072544, grad=1)
epoch: 184:  loss: Tensor(value=0.0032645206362841082, grad=1)
epoch: 185:  loss: Tensor(value=0.003246078810956993, grad=1)
epoch: 186:  loss: Tensor(value=0.0032278464810769985, grad=1)
epoch: 187:  loss: Tensor(value=0.0032098201028807087, grad=1)
epoch: 188:  loss: Tensor(value=0.0031919962120901687, grad=1)
epoch: 189:  loss: Tensor(value=0.003174371421690535, grad=1)
epoch: 190:  loss: Tensor(value=0.0031569424197822517, grad=1)
epoch: 191:  loss: Tensor(value=0.0031397059675047643, grad=1)
epoch: 192:  loss: Tensor(value=0.003122658897029171, grad=1)
epoch: 193:  loss: Tensor(value=0.003105798109616961, grad=1)
epoch: 194:  loss: Tensor(value=0.0030891205737422874, grad=1)
epoch: 195:  loss: Tensor(value=0.0030726233232756467, grad=1)
epoch: 196:  loss: Tensor(value=0.0030563034557261444, grad=1)
epoch: 197:  loss: Tensor(value=0.003040158130540614, grad=1)
epoch: 198:  loss: Tensor(value=0.003024184567456984, grad=1)
epoch: 199:  loss: Tensor(value=0.0030083800449104323, grad=1)
```

این مقادیر نشان دهنده کاهش تدریجی مقدار تابع هزینه در طول فرآیند آموزش مدل است و نشان می‌دهند که مدل با استفاده از روش گرادیان نزولی بهبود یافته و در هر دوره، خطا کاهش یافته است. این امر نشان از عملکرد موثر مدل در پیش‌بینی برچسب‌های جدید دارد.