1. Installation

<miniconda설치>

① Miniconda — miniconda documentation에서 miniconda 설치합니다.

② conda를 직접 실행할 수 있도록 셸을 초기화합니다.

```
~/miniconda3/bin/conda init
```

③ 현재 셸을 닫고 다시 열어서 새 환경을 만듭니다.

```
conda create --name d2l python=3.9 -y
```

④ 환경을 활성화합니다.

```
conda activate d2l
```

<딥 러닝 프레임워크와 d2l 패키지 설치하기>

① 딥러닝 프레임워크 설치

-PyTorch를 설치합니다.

```
pip install torch==2.0.0 torchvision==0.15.1
```

- MXNet 설치합니다.

```
# For macOS and Linux users
pip install mxnet-cu112==1.9.1

# For Windows users
pip install mxnet-cu112==1.9.1 -f https://dist.mxnet.io/python
```

- JAX와 Flax을 설치합니다.

```
# GPU
pip install "jax[cuda11_pip]==0.4.13" -f https://storage.googleapis.com/jax-releases/jax_cuda_releases.html fl
```

```
# CPU
pip install "jax[cpu]==0.4.13" flax==0.7.0
```

- Tensorflow 설치합니다.

```
pip install tensorflow==2.12.0 tensorflow-probability==0.20.0
```

② d2l 패키지를 설치합니다.

```
pip install d2l==1.0.3
```

<코드 다운로드 및 실행하기>

① 각 장의 코드 블록을 실행할 수 있도록 노트북을 다운로드합니다.

```
mkdir d2l-en && cd d2l-en
curl https://d2l.ai/d2l-en-1.0.3.zip -o d2l-en.zip
unzip d2l-en.zip && rm d2l-en.zip
```

② jupyter notebook을 실행합니다.

```
jupyter notebook
```

3. Linear Neural Networks for Regression

3.1 Linear Regression

Regression은 수치 값을 예측하고자 할 때 발생합니다.

예를 들어 **각 집에 대한 면적과 나이를 고려해서 집의 가격을 예측한다고** 했을 때,

한 거래에 해당하는 행은 **example(or data point, instance, sample)**이라고 합니다.

예측하려는 대상인 '집의 가격'은 **label(or target)**이라고 부릅니다.

나이와 면적은 **features(or covariates)**라고 합니다.

1-1. Model

Regression중에서 가장 간단하면서도 인기있는 회귀문제인 Linear Regression은 feature와 target간의 관계가 대략적으로 선형이라고 가정합니다.(target(집의 가격)의 기대값이 features(면적과 연식)의 가중 합으로 표현될 수 있다는 것입니다.)

$$\hat{y} = w_1 x_1 + \cdots + w_d x_d + b.$$

여기서 w1, w2,...는 가중치(weights)라고 하고, b는 편향(bias, offset, intercept)이라고 합니다. 그리고 입력이 n개의 특징을 갖는 경우 각각에 인덱스를 할당할 수 있으며, 일반적으로 "hat" 기호는 추정값을 나타냅니다

모델학습의 목표: 동일한 분포에서 샘플링된 새로운 데이터 예제의 특징이 주어졌을 때, 새로운 예제의 레이블이 (평균적으로) 최소의 오차로 예측되도록 하는, **가중치 벡터 w와 편향 b를 찾는 것입니다.**

$$\hat{y} = \mathbf{w}^\top \mathbf{x} + b.$$

첫번째 식을 위와 같이 표현할 수도 있습니다.

1-2.Loss function

손실 함수는 대상의 실제 값과 예측 값 사이의 거리를 측정합니다. 손실함수는 일반적으로 0보다 크거나 같은 숫자이며, 작을수록 좋으며 완벽한 예측은 손실이 0입니다. 회귀 문제의 경우 가장 일반적인 손실 함수는 **제곱 오차(Squared error)입니다**.

예제i에 대한 예측이 $\hat{y}^{(i)}$ 이고 해당하는 실제 레이블이 $y^{(i)}$ 일 때, 제곱 오차는 다음과 같습니다. 손실의 도함수를 취할 때 표기상 편리함을 위해 1/2를 추가로 곱합니다.

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

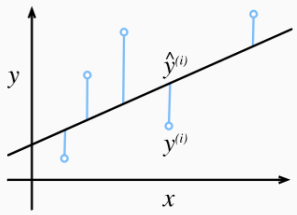


Fig. 3.1.1 Fitting a linear regression model to one-dimensional data.

제곱의 형태이기 때문에, 오차가 커질수록 제곱 오차 손실은 기하급수적으로 증가합니다. 따라서 제곱 오차 손실 함수는 이상치(특이값)에 대해 지나치게 민감할 수 있습니다. 이상치의 큰 오차가 손실을 크게 증가시킬 수 있기 때문입니다. 따라서 전체 데이터셋에 대한 모델의 품질을 측정하기 위해 우리는 훈련 세트에서 손실을 평균화(또는 동등하게 합산)합니다

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

모델을 훈련할 때, 우리는 아래와 같이 손실함수를 최소하시키는 w와 b파라미터를 찾고자 합니다.

$$\mathbf{w}^*, b^* = \operatorname*{argmin}_{\mathbf{w}, b}\ L(\mathbf{w}, b).$$

1-3. Analytic solution

X matrix가 full rank(어떠한 feature도 선형의존적이면 안된다)일 때, $\|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2$을 최소화하시키는 매개변수를 간단한 공식을 적용하여 찾을 수 있습니다.

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

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

1-4. Minibatch Stohastic Gradient Descent

위의 공식으로 해결할 수 없는 경우에도 모델을 효과적으로 훈련시킬 수 있는 방법이 있습니다. 게다가 많은 작업에서는 최적화가 어려운 모델이 더 나은 결과를 가져와서 그 모델을 훈련하는 방법을 찾는 것이 공식으로 찾는 것보다 수고스럽더라도 가치가 있을 때가 많습니다.

거의 모든 딥러닝 모델을 최적화하는 주요 기술은 에러를 반복적으로 줄이기 위해 매개변수를 업

데이트하는 것으로 **Gradient Descent**라고 합니다.

① Batch Gradient Descent

손실 함수를 모든 훈련 데이터에 대해 계산하고 매개변수를 업데이트합니다. 이는 전체 데이터셋에 대한 정보를 사용하므로 더 정확한 업데이트가 가능하지만, 데이터가 많은 경우 한 번의 업데이트에 많은 계산이 필요하며, 중복된 데이터가 많은 경우 업데이트 효과가 제한될 수 있습니다.

② Stochastic Gradient Descent, SGD

SGD는 한 번에 하나의 예제만 고려하고 하나의 관측에 기반하여 업데이트 단계를 수행하는 것입니다. 이 방법은 큰 데이터셋에도 효과적일 수 있지만, 데이터를 메모리에서 캐시로 이동하는 것은 연산에 비해 상당한 시간이 걸리는데 SGD는 한 번에 하나의 샘플을 처리하기 때문에 데이터를 이동하는 비용이 상당히 큽니다. 또한 각 샘플에 대한 업데이트를 수행하므로 행렬-벡터 연산이 아니라 벡터-벡터 연산으로 처리되는데 이는 더 낮은 계산 효율을 가집니다.

③ Mini-Batch Stochastic Gradient Descent

위 단점들을 보완하기 위해 미니배치 확률적 그래디언트 디센트(Mini-Batch Stochastic Gradient Descent)가 사용되는데, 이는 각 업데이트 단계에서 전체 데이터셋이 아니라 작은 미니배치를 사용하여 업데이트를 수행하는 것입니다. 이는 계산 효율성과 업데이트의 안정성을 조화시키는 중간적인 방법입니다.

미니배치 확률적 경사 하강법(minibatch SGD)은 다음과 같이 진행됩니다

(i) 모델 매개변수의 값을 일반적으로 무작위로 초기화합니다.

(ii) 데이터에서 무작위 미니배치를 반복적으로 샘플링하면서 매개변수를 음의 그래디언트 방향으로 업데이트합니다

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \partial_{(\mathbf{w},b)} l^{(i)}(\mathbf{w}, b).$$

학습률은 매개변수를 업데이트할 때의 보폭을 말하는데, 이 값을 미니배치 크기로 나누면 한 번의 업데이트에서 각 샘플이 기여하는 정도를 나타낼 수 있습니다. 이렇게 함으로써 더 안정적으로 수렴하고, 특히 미니배치 크기가 큰 경우에도 효과적으로 학습을 진행할 수 있습니다.

여기서 학습률과 미니매치 크기는 훈련과정에서 업데이트 되지 않는 Hyperparameter라고 합니다.

1-5. prediction

모델이 주어진 경우, 면적 x1과 나이 x2가 주어졌을 때 이전에 보지 못한 집의 판매 가격을 예측 **(prediction)**할 수 있습니다.

3.2 Object Oriented design for implementation

3가지 클래스를 생성해서 실행합니다.

(i) Module은 모델, 손실 및 최적화 방법을 포함합니다.

Module 클래스는 구현할 모든 모델의 기본 클래스입니다. 첫 번째로 __init__메서드는 기본적으로 모델의 하이퍼파라미터를 저장하고, ProgressBoard를 초기화하여 학습 진행 상황을 시각화하는 데 사용될 보드를 만듭니다. training_step 메서드와 validation_step메서드는 각각 학습 및 검증 단계에서 손실을 계산하고, 그 결과를 시각화하는데 사용됩니다. configure_optimizers 메서드는 학습 가능한 매개변수를 업데이트하는 데 사용되는 최적화 방법 또는 그들의 리스트를 반환합니다.

```
class Module(tf.keras.Model, 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()
        self.training = None

    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 call(self, X, *args, **kwargs):
        if kwargs and "training" in kwargs:
            self.training = kwargs['training']
        return self.forward(X, *args)

    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.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
```

(ii) DataModule은 훈련 및 검증을 위한 데이터 로더를 제공합니다. DataModule 클래스는 데이터의 기본 클래스입니다. __init__ 메서드는 데이터를 준비하는 데 사용되고 필요한 경우 데이터를 다운로드하고 전처리하는 것을 포함합니다. train_dataloader는 훈련 데이터셋에 대한 데이터 로더를

반환합니다. 그런 다음 Module클래스의 training_step 메서드에 공급되어 손실을 계산하는 데 사용됩니다. val_dataloader는 검증 데이터셋 로더를 반환합니다.

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

(iii) 두 클래스를 Trainer 클래스를 사용하여 결합하며, 이를 통해 다양한 하드웨어 플랫폼에서 모델을 훈련시킬 수 있습니다. Trainer 클래스는 Module 클래스의 학습 가능한 매개변수를 DataModule에서 지정한 데이터로 훈련합니다. 주요 메서드는 fit으로, 두 인수를 받습니다. model은 Module의 인스턴스이고, data는 DataModule의 인스턴스입니다. 그런 다음 이 메서드는 모델을 훈련하기 위해 전체 데이터셋을 max_epochs번 반복합니다.

```
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
```

3.3 Synthetic Regression Data

1. Generating the Dataset

모델의 성능을 테스트하고 모델이 불확실성을 어떻게 처리하는지 확인하기 위해 노이즈를 추가한 합성 데이터셋을 만들어줍니다. 이때, 표준 정규 분포에서 뽑은 2차원 특징을 가진 1000개의 예제를 생성합니다. 그리고 편의를 위해 노이즈 ϵ가 평균이 0이고 표준 편차가 0.01인 정규 분포에서 추출된 것으로 가정합니다.

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b + \boldsymbol{\epsilon}.$$

```
class SyntheticRegressionData(d2l.DataModule):  #@save
    """Synthetic data for linear regression."""
    def __init__(self, w, b, noise=0.01, num_train=1000, num_val=1000,
                 batch_size=32):
        super().__init__()
        self.save_hyperparameters()
        n = num_train + num_val
        self.X = tf.random.normal((n, w.shape[0]))
        noise = tf.random.normal((n, 1)) * noise
        self.y = tf.matmul(self.X, tf.reshape(w, (-1, 1))) + b + noise
```

2. reading the dataset

기계 학습 모델을 훈련시키려면 데이터 세트를 여러 번 통과하여 한 번에 하나의 미니배치를 가져와야 합니다. 그런 다음 이 데이터는 모델을 업데이트하는 데 사용됩니다. 이 데이터를 생성하과 관리하는 것을 get_dataloader 메서드로 구현합니다.

```
@d2l.add_to_class(SyntheticRegressionData)
def get_dataloader(self, train):
    if train:
        indices = list(range(0, self.num_train))
        # The examples are read in random order
        random.shuffle(indices)
    else:
        indices = list(range(self.num_train, self.num_train+self.num_val))
    for i in range(0, len(indices), self.batch_size):
        j = tf.constant(indices[i : i+self.batch_size])
        yield tf.gather(self.X, j), tf.gather(self.y, j)
```

3.4 Linear Regression Implementation from Scratch

이제 선형 회귀의 완전한 기능을 갖춘 구현을 진행할 준비가 되었습니다. 이 섹션에서는 (i) 모델; (ii) 손실 함수; (iii) 미니배치 확률적 경사 하강 옵티마이저; 그리고 (iv) 이러한 모든 구성 요소를 함께 연결하는 훈련 함수를 scratch에서 구현할 것입니다. 마지막으로 Section 3.3에서 생성한 get_dataloader를 실행하고 결과 데이터셋에 모델을 적용할 것입니다.

1. Defining the Model

가중치를 평균이 0이고 표준 편차가 0.01인 정규 분포에서 무작위로 선택된 숫자로 초기화합니다. 또한 편향을 0으로 설정합니다.

```
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()
        w = tf.random.normal((num_inputs, 1), mean=0, stddev=0.01)
        b = tf.zeros(1)
        self.w = tf.Variable(w, trainable=True)
        self.b = tf.Variable(b, trainable=True)
```

다음으로 입력 특징 X와 모델 가중치 w의 행렬-벡터 곱셈을 간단히 수행하고 각 예제에 오프셋 b를 더하여 출력을 생성하는 forward메서드를 정의합니다.

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

2. Defining the Loss Function

모델을 업데이트하려면 손실 함수의 그래디언트를 계산해야 합니다. 손실함수로는 제곱 손실함수를 사용하며, 미니배치의 모든 예제에 대한 평균 손실 값을 반환합니다.

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

3. Defining the Optimization Algorithm

각 단계에서 데이터셋에서 무작위로 추출한 미니배치를 사용하여 손실에 대한 매개변수의 그래디

언트를 추정합니다. 그런 다음 손실을 감소시킬 수 있는 방향으로 매개변수를 업데이트합니다.

apply_gradients 메서드를 통해 주어진 매개변수 세트와 학습률 lr에 대한 업데이트를 적용합니다.

```
class SGD(d2l.HyperParameters):  #@save
    """Minibatch stochastic gradient descent."""
    def __init__(self, lr):
        self.save_hyperparameters()

    def apply_gradients(self, grads_and_vars):
        for grad, param in grads_and_vars:
            param.assign_sub(self.lr * grad)
```

다음으로 configure_optimizers 메서드를 정의합니다. 이 메서드는 SGD 클래스의 인스턴스를 반환합니다.

```
@d2l.add_to_class(LinearRegressionScratch)  #@save
def configure_optimizers(self):
    return SGD(self.lr)
```

3.4 Training

각 반복에서 훈련 예제의 미니배치를 가져오고 모델의 training_step 메서드를 사용하여 손실을 계산합니다. 그런 다음 각 매개변수에 대한 기울기를 계산합니다. 마지막으로 최적화 알고리즘을 호출하여 모델 매개변수를 업데이트합니다. 요약하면 다음과 같은 루프를 실행합니다

- Initialize parameters $(\mathbf{w}, b)$
- Repeat until done
  - Compute gradient $\mathbf{g} \leftarrow \partial_{(\mathbf{w},b)} \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} l(\mathbf{x}^{(i)}, y^{(i)}, \mathbf{w}, b)$
  - Update parameters $(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \eta \mathbf{g}$

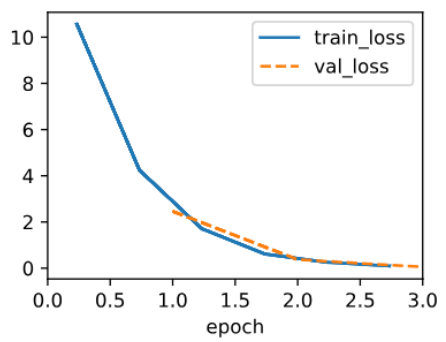
그리고 각 에포크마다 검증 데이터로더를 한 번 전달하여 모델 성능을 측정합니다.

```python
@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.training = True
    for batch in self.train_dataloader:
        with tf.GradientTape() as tape:
            loss = self.model.training_step(self.prepare_batch(batch))
        grads = tape.gradient(loss, self.model.trainable_variables)
        if self.gradient_clip_val > 0:
            grads = self.clip_gradients(self.gradient_clip_val, grads)
        self.optim.apply_gradients(zip(grads, self.model.trainable_variables))
        self.train_batch_idx += 1
    if self.val_dataloader is None:
        return
    self.model.training = False
    for batch in self.val_dataloader:
        self.model.validation_step(self.prepare_batch(batch))
        self.val_batch_idx += 1
```

모델을 훈련시키기 전에 훈련 데이터가 필요합니다. 여기서는 SyntheticRegressionData클래스를 사용하고 학습률을 lr=0.03로 설정하고 max_epochs=3으로 모델을 훈련시켰습니다.

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



3.5 Concise Implementation of Linear Regression

딥 러닝 프레임워크의 고수준 API를 사용하여 3.4절에서 소개한 선형 회귀 모델을 간결하게 구현할 수 있습니다.

1.Defining the Model

Keras에서 완전 연결 레이어는 Dense 클래스로 정의됩니다. 여기서는 단일 스칼라 출력을 생성하려고 하기 때문에 해당 숫자를 1로 설정합니다. 나중에 net(X)를 실행할 때 처음으로 모델을 통해 데이터를 전달하려고 하면 Keras가 자동으로 각 레이어의 입력 수를 추론하기 때문에 이 선형 레이어에 몇 개의 입력이 들어가는지 알려줄 필요가 없습니다

```
class LinearRegression(d2l.Module):  #@save
    """The linear regression model implemented with high-level APIs."""
    def __init__(self, lr):
        super().__init__()
        self.save_hyperparameters()
        initializer = tf.initializers.RandomNormal(stddev=0.01)
        self.net = tf.keras.layers.Dense(1, kernel_initializer=initializer)
```

forward 메서드에서는 미리 정의된 레이어의 내장 call 메서드를 호출하여 출력을 계산합니다.

```
@d2l.add_to_class(LinearRegression)  #@save
def forward(self, X):
    return self.net(X)
```

3.2 Defining the Loss function

MeanSquaredError 클래스는 평균 제곱 오차를 계산합니다. 기본적으로 예제별 평균 손실을 반환합니다.

```
@d2l.add_to_class(LinearRegression)  #@save
def loss(self, y_hat, y):
    fn = tf.keras.losses.MeanSquaredError()
    return fn(y, y_hat)
```

3.3 Defining the Optimization Algorithm

Keras는 optimizers 모듈에서 SGD 알고리즘과 여러 변형을 지원합니다.

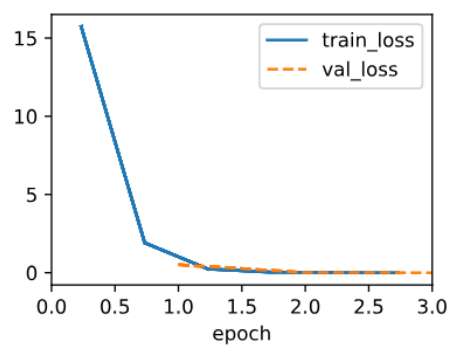
```
@d2l.add_to_class(LinearRegression)  #@save
def configure_optimizers(self):
    return tf.keras.optimizers.SGD(self.lr)
```

3.4 Training

모델을 딥러닝 프레임워크의 고수준 API를 통해 표현하면 더 적은 코드 줄이 필요함을 알 수 있을 것입니다. 매개변수를 개별적으로 할당하거나 손실 함수를 정의하거나 미니배치 SGD를 구현할 필요가 없었습니다. 훨씬 더 복잡한 모델을 다룰 때 고수준 API의 장점은 크게 증가할 것입니다.

이제 기본적인 구성 요소가 모두 준비되었으므로 훈련 루프 자체는 이전에 처음부터 구현한 것과 동일합니다. 따라서 모델을 훈련하기 위해 fit 메서드를 호출하기만 하면 됩니다.

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



3.6 Generalization

고전적인 이론에서는 간단한 모델과 풍부한 데이터가 있는 경우 훈련 및 일반화 오차가 서로 가까울 것으로 기대됩니다. 그러나 더 복잡한 모델이나/또는 더 적은 예제를 사용하는 경우 훈련 오차가 감소할 것으로 예상되지만, 일반화 오차와의 차이는 커질 것으로 기대됩니다.

일반화 오차와의 차이가 너무 적다면, 모델이 우리가 모델링하려는 패턴을 충분히 잡아내지 못하도록 너무 단순하다는 것을 의미할 수 있습니다. 이것을 언더피팅(underfitting)이라고 합니다.

반면, 훈련오차가 일반화 오차보다 훨씬 낮은 경우(일반화 오차와의 차이가 큰 경우)에는 심각한 오버피팅(overfitting)이 나타납니다.

모델을 고정시키면 훈련 데이터셋의 샘플 수가 적을수록 오버피팅을 더 자주(그리고 심각하게) 경험할 가능성이 높습니다. 따라서 일반화 오차를 줄이기 위해서는 **훈련 데이터의 양을 늘리는 것이** 좋습니다.

모델 선택 프로세스에서 테스트 데이터를 사용하면 테스트 데이터에 오버피팅될 수 있는 위험이 있습니다. 그러나 모델 선택을 위해 훈련 데이터만 의존해서도 안 됩니다. 왜냐하면 모델을 훈련하는 데 사용한 데이터를 통해 일반화 오차를 추정할 수 없기 때문입니다. 이 문제를 해결하기 위한 일반적인 방법은 데이터를 훈련, 검증 및 테스트 데이터셋으로 나누는 것입니다. 만약 이 마저도 훈련데이터가 부족해 충분한 데이터를 확보하여 적절한 검증 세트를 구성하는 것조차 어렵다면 k-fold교차 검증을 사용합니다.

3.7 Weight Decay

과적합을 완화하는 방법 중 하나는 더 많은 훈련 데이터를 수집하는 것입니다. 그러나 이는 비용이 많이 들거나 시간이 많이 걸리거나 완전히 제어할 수 없는 경우가 있어 단기간에 불가능할 수 있습니다. 따라서 현재로서는 이미 보유한 데이터로 가능한 한 많은 고품질 데이터를 갖고 있다고 가정하고, 데이터셋이 주어졌을 때 사용 가능한 도구에 중점을 두어야 합니다.

Weight Decay(가중치 감쇠)는 과적합을 줄이기 위한 정규화(regularization) 기법 중 하나로, 모델의 복잡성을 제어하기 위해 가중치 파라미터에 대한 패널티를 부과하는 방법입니다. 이는 특히 딥러닝에서 널리 사용되는 기법 중 하나입니다.

Weight Decay의 아이디어는 큰 가중치 값이 모델을 더 복잡하게 만들고, 이로 인해 훈련 데이터에 대한 과적합이 발생할 수 있다는 것입니다. 따라서 가중치 값을 작게 유지하여 모델을 더 간단하게 만들고, 일반화 성능을 향상시키려는 것입니다.

수식적으로, 가중치 감쇠는 손실 함수에 추가된 패널티 항으로 표현됩니다. 주로 L2 노름(norm)을 사용하며, 이를 손실 함수에 더한 후 전체 손실을 최소화하는 방향으로 모델을 훈련합니다. 가중치 감쇠를 통해 학습 과정에서 가중치가 작아지도록 유도되며, 이는 모델이 더 간단한 결정 경계

를 학습하도록 도움을 줍니다. 이는 모델의 일반화 성능을 향상시키고 새로운 데이터에 대한 예측을 더 일반적으로 만들어줄 수 있습니다.

L2 norm을 아래와 같이 정의하고

```
def l2_penalty(w):
    return tf.reduce_sum(w**2) / 2
```

정의한 L2 norm을 적용하여 학습시켰더니

```
class WeightDecayScratch(d2l.LinearRegressionScratch):
    def __init__(self, num_inputs, lambd, lr, sigma=0.01):
        super().__init__(num_inputs, lr, sigma)
        self.save_hyperparameters()

    def loss(self, y_hat, y):
        return (super().loss(y_hat, y) +
                self.lambd * l2_penalty(self.w))
```

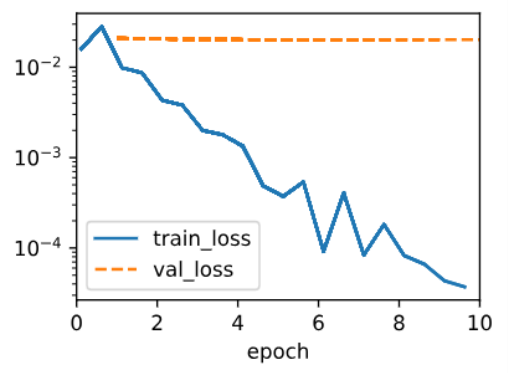
```
data = Data(num_train=20, num_val=100, num_inputs=200, batch_size=5)
trainer = d2l.Trainer(max_epochs=10)

def train_scratch(lambd):
    model = WeightDecayScratch(num_inputs=200, lambd=lambd, lr=0.01)
    model.board.yscale='log'
    trainer.fit(model, data)
    print('L2 norm of w:', float(l2_penalty(model.w)))
```

L2 norm을 적용하지 않았을 때보다 검증 에러가 줄은 것을 확인할 수 있었습니다.

```
train_scratch(0)
```

```
L2 norm of w: 0.010886103846132755
```



```
train_scratch(3)
```

```
L2 norm of w: 0.0017435649642720819
```