T05460301-01

Deep Learning
(Fall 2023)

Ikbeom Jang

ijang@hufs.ac.kr
CES HUFS

Linear Neural Networks

- Linear Regression Implementation from Scratch

Contents

Synthetic Regression Data

Linear Regression Implementation from Scratch

- In this section, we will implement the entire method from scratch:
  - Data pipeline
  - Model
  - Loss function
  - Minibatch stochastic gradient descent optimizer.

- We will rely only on tensors and auto differentiation.

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

Generating the Dataset

- We will construct an artificial dataset according to a linear model with additive noise.

- We generate a dataset containing 1000 examples, each consisting of 2 features sampled from a standard normal distribution.
  - Our synthetic dataset will be a matrix $\boldsymbol{X} \in R^{1000\times 2}$.

- The true parameters generating our dataset will be $\boldsymbol{w} = [2, -3.4]^\top$ and $b = 4.2$, and our synthetic labels will be assigned according to the following linear model with the noise term ϵ:

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{w} + b + \epsilon. \tag{3.2.1}$$

  - Think of ϵ as capturing potential measurement errors on the features and labels.
  - Assume that ϵ obeys a normal distribution with mean of 0.
  - We will set its standard deviation to 0.01.

Generating the Dataset

- The following code generates our synthetic dataset.

```
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 = torch.randn(n, len(w))
        noise = torch.randn(n, 1) * noise
        self.y = torch.matmul(self.X, w.reshape((-1, 1))) + b + noise

data = SyntheticRegressionData(w=torch.tensor([2, -3.4]), b=4.2)
```

- Each row in features consists of a 2-dimensional data example.
- Each row in labels consists of a 1-dimensional label value (a scalar).

```
print('features:', data.X[0],'\nlabel:', data.y[0])
```

Reading the Dataset

- Training models consists of:
  - Making multiple passes over the dataset,
  - Grabbing one minibatch of examples at a time,
  - Using them to update our model.

- The training process is so fundamental to machine learning algorithms,
  - It is worth defining a utility function to shuffle the dataset and access it in minibatches.

- The *get_dataloader* function takes a batch size, a matrix of features, and a vector of labels, yielding minibatches of the size *batch_size*.
  - Each minibatch consists of a tuple of features and labels.

object oriented

→ @ 파이썬에서 module 기존 클래스에 function을 넣어주게 가능해

```
@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):
        batch_indices = torch.tensor(indices[i: i+self.batch_size])
        yield self.X[batch_indices], self.y[batch_indices]
```

function을 후에 추가하는 것도 가능하다

Reading the Dataset

- We want to use reasonably sized minibatches to take advantage of parallelizing operations using GPU hardware.
  - Each example can be fed through our models in parallel and the gradient of the loss function for each example can also be taken in parallel.

- Let us read and print the first small batch of data examples.

```
X, y = next(iter(data.train_dataloader()))
print('X shape:', X.shape, '\ny shape:', y.shape)
```

16GB 인데 Batch가 1GB 이면 비효율적
적당한 memory를 쓰면서 minibatch를 하는것이 좋음

- Notes:
  - The shape of the features in each minibatch gives the minibatch size and the number of input features.
  - Minibatch of labels will have a shape given by $batch\_size$.
  - As we run the iteration, we obtain distinct minibatches successively until the entire dataset has been exhausted.
  - The implemented iteration above is inefficient, it requires loading all the data in memory and performing lots of random memory access.
  - The built-in iterators implemented in a deep learning framework are more efficient and they can deal with both data stored in files and data fed via data streams.

Data Loader

- Rather than writing our own iterator, we can call the existing API in a framework to load data.
- As before, we need a dataset with features X and labels y.
- Beyond that, we set *batch_size* in the built-in data loader and let it take care of shuffling examples efficiently.

```
@d2l.add_to_class(d2l.DataModule)    #@save
def get_tensorloader(self, tensors, train, indices=slice(0, None)):
    tensors = tuple(a[indices] for a in tensors)
    dataset = torch.utils.data.TensorDataset(*tensors)
    return torch.utils.data.DataLoader(dataset, self.batch_size, shuffle=train)

@d2l.add_to_class(SyntheticRegressionData)    #@save
def get_dataloader(self, train):
    i = slice(0, self.num_train) if train else slice(self.num_train, None)
    return self.get_tensorloader((self.X, self.y), train, i)
```

- The new data loader behaves just like the previous one, except that it is more efficient and has some added functionality.

```
X, y = next(iter(data.train_dataloader()))
print('X shape:', X.shape, '\ny shape:', y.shape)
```

Linear Regression Implementation from Scratch

Linear Regression Implementation from Scratch

- We will implement the entire method from scratch, including (i) the model; (ii) the loss function; (iii) a minibatch stochastic gradient descent optimizer; and (iv) the training function that stitches all of these pieces together.
- Finally, we will run our synthetic data generator from previous Section and apply our model on the resulting dataset.
- While modern DL frameworks can automate nearly all of this work, implementing things from scratch is the only way to make sure that you really know what you are doing.
- Moreover, when it is time to customize models, defining our own layers or loss functions, understanding how things work under the hood will prove handy.

- We will rely only on tensors and auto differentiation.

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

Defining the Model

- Before we can begin optimizing our model's parameters by minibatch SGD, we need to have some parameters in the first place.
- In the following we initialize weights by drawing random numbers from a normal distribution with mean 0 and a standard deviation of 0.01.
- The magic number 0.01 often works well in practice, but you can specify a different value through the argument sigma.
- Moreover, we set the bias to 0.
- Note that for object-oriented design we add the code to the __init__ method of a subclass of d2l.Module

```
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)
```

Defining the Model

- Next, we must define our model, relating its inputs and parameters to its outputs.

- To calculate the output of the linear model,
  - Take the matrix-vector dot product of the input features $\boldsymbol{X}$ and the model weights $\boldsymbol{w}$,
  - Add the offset b to each example.

- Note that $\boldsymbol{X}\boldsymbol{w}$ is a vector and b is a scalar.
  - So, the broadcasting mechanism is applied (the scalar is added to each component of the vector).

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

Defining the Loss Function

- Since updating our model requires taking the gradient of our loss function, we ought to define the loss function first.
- We will use the squared loss function.
    - We need to transform the true value y into the predicted value’s shape y_hat.
    - The result returned by the following function will also have the same shape as y_hat.

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

Defining the Optimization Algorithm

- At each step, using one minibatch randomly drawn from our dataset,
    - We will estimate the gradient of the loss with respect to our parameters. 얼마나 빨리 learning rate
    - Next, we will update our parameters in the direction that may reduce the loss. 얼만큼 훈련시킬건지 batch

- The following code applies the minibatch stochastic gradient descent (SGD) update, given a set of parameters, a learning rate (size of the update step, lr), and a batch size.
- Since our loss is computed as an average over the minibatch, we do not need to adjust the learning rate against the batch size.
- In later chapters we will investigate how learning rates should be adjusted for very large minibatches as they arise in distributed large-scale learning.

Defining the Optimization Algorithm

- We define our SGD class to have a similar API as the built-in SGD optimizer.
    - We update the parameters in the step method.
    - The zero_grad method sets all gradients to 0, which must be run before a backpropagation step.

```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_()
```

- Next, we define the configure_optimizers method, which returns an instance of the SGD class.

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

Training

- We will execute the following loop:
  - Initialize parameters $(\boldsymbol{w}, b)$
  - Repeat until done
    - Compute gradient $\boldsymbol{g} \leftarrow \partial(\boldsymbol{w}, b) \frac{1}{|B|} \sum_{i \in B} l(x^{(i)}, y^{(i)}, \boldsymbol{w}, b)$
    - Update parameters $(\boldsymbol{w}, b) \leftarrow (\boldsymbol{w}, b) - \eta \boldsymbol{g}$

Grab a minibatch of training examples → Pass them to the model to obtain a set of predictions → Calculate the loss → Backward through the network, storing the gradients with respect to each parameter → Call the SGD to update the model parameters.

- In each *epoch*, iterate through the entire dataset (using the $data\_iter$ function) once passing through every example in the training dataset.
  - Assuming that the number of examples is divisible by the batch size.

- The number of epochs num_epochs and the learning rate lr are both hyperparameters, which we set to 3 and 0.03, respectively.
  - Setting hyperparameters is tricky and requires some adjustment by trial and error.

본인이 체득해서 하는것이기 때문에 경험으로.

Training

- Next, we must define our model, relating its inputs and parameters to its outputs.

- To calculate the output of the linear model,
  - Take the matrix-vector dot product of the input features $\boldsymbol{X}$ and the model weights $\boldsymbol{w}$,
  - Add the offset b to each example.

- Note that $\boldsymbol{Xw}$ is a vector and b is a scalar.
  - So, the broadcasting mechanism is applied (the scalar is added to each component of the vector).

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

d2l에서만 가능　d2l library를 불러올때　사용자가 정의한 함수를
다른 스크립트에서도 불러올 수 있음

Training

- In most cases, we want a validation dataset to measure our model quality.
- Here we pass the validation dataloader once in each epoch to measure the model performance.
- The *prepare_batch* and *fit_epoch* methods are registered in the *d2l.Trainer* class

```
@d2l.add_to_class(d2l.Trainer) #@save 중요.
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
```

Training

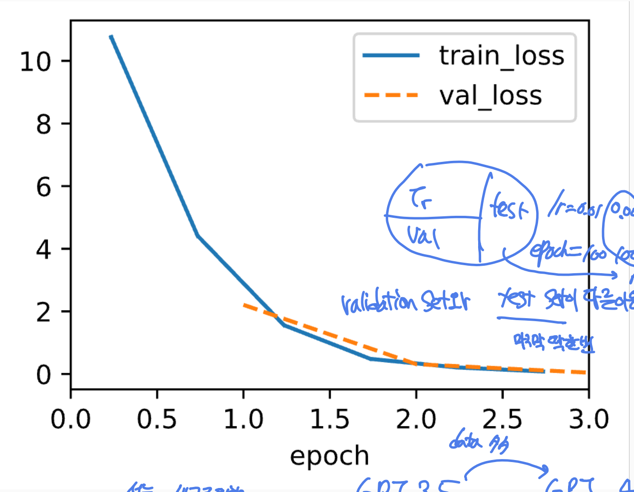
- For training data, we use the SyntheticRegressionData class and pass in some ground truth parameters.
- Then we train our model with the learning rate lr=0.03 and set max_epochs=3.
- In general, both the number of epochs and the learning rate are hyperparameters. test
- In general, setting hyperparameters is tricky and we will usually want to use a three-way split, one set for **training**, a second for **hyperparameter selection**, and the third reserved for the **final evaluation**.
- We elide these details for now but will revise them later.

모델학습 training
모델 성능평가 validation
weight, bias ⇒ model parameter 수정

```
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)
```

validation set
gradient 가는 방향이 제대로 가는지 확인

파라미터 업데이트는 training set으로 하는 것.



결국에는 validation set으로 모델이 overfitting되는 것은 아닌지?

이것도 과하게 하면 모델도 결국 학습되버리는 것.

결국 test set 만나면 validation set에 학습되버린 모델에 적용시키면 결과는 좋지 않음.

test set 모델의 성능평가 모델검증 prediction에서 새로운데이터가 들어와도 잘되는지

GPT 3.5 → GPT 4.0

Tr val Test   Tr val test

성능 비교를위함
3.5 Test를 한번 썼기 때문
3.5 Test → 4.0 val에 사용 가능

납득시킬려면 Test set이 좋아야 성능평가 됨.

Training

- Because we synthesized the dataset ourselves, we know precisely what the true parameters are.

- Thus, we can evaluate our success in training by comparing the true parameters with those that we learned through our training loop.
    - They turn out to be very close to each other.

```
with torch.no_grad():
    print(f'error in estimating w: {data.w - model.w.reshape(data.w.shape)}')
    print(f'error in estimating b: {data.b - model.b}')
```

- Note that we should not take it for granted that we are able to recover the parameters perfectly.
    - We are less concerned with recovering true underlying parameters, and more concerned with parameters that
      lead to highly accurate prediction.

- Even on difficult optimization problems, stochastic gradient descent can often find remarkably good solutions.
    - Owing partly to the fact that, for deep networks, there exist many configurations of the parameters that lead to highly accurate prediction.

Summary

- We saw how a deep network can be implemented and optimized from scratch, using just tensors and auto differentiation, without any need for defining layers or fancy optimizers.
- This section only scratches the surface of what is possible. In the following sections, we will describe additional models based on the concepts that we have just introduced and learn how to implement them more concisely.