Basics of Machine Learning

[15]:
```
print(x,y,x < y)
```
```
tensor([3., 2.]) tensor([4., 1.]) tensor([ True, False])
```

[16]:
```
print(x.sum(),x.mean())
```
```
tensor(5.) tensor(2.5000)
```

The concept of *Broadcasting* is very useful to simplify some calculations. As seen above the elementwise operations, need the tensors to be of the same shape.

Under certain conditions, the elementwise operations can even be applied to tensors of different shape.

[17]:
```
x = torch.tensor([[3.0],[2]])
y = torch.tensor([[4.0,1,2]])
print(x.shape, y.shape)
```
```
torch.Size([2, 1]) torch.Size([1, 3])
```

[18]:
```
z = x+y
print(z,z.shape)
```
```
tensor([[7., 4., 5.],
        [6., 3., 4.]]) torch.Size([2, 3])
```

As you can observe, the tensor x (dimension 2×1) and y (dimension 1×3) are broadcasted to the same shape (2×3) with an subsequent elementwise operation:

$$x + y = \begin{pmatrix} 3 & 3 & 3 \\ 2 & 2 & 2 \end{pmatrix} + \begin{pmatrix} 4 & 1 & 2 \\ 4 & 1 & 2 \end{pmatrix} = \begin{pmatrix} 7 & 4 & 5 \\ 6 & 3 & 4 \end{pmatrix}$$

Further, specific elements can be accessed with indexing and slicing operations

[19]:
```
a = torch.arange(12).reshape(-1,4)
print(a)
```
```
tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
```

[20]:
```
print(a[0,0], a[0,:], a[0:2,0])
```
```
tensor(0) tensor([0, 1, 2, 3]) tensor([0, 4])
```

[21]:
```
print(a[:,3], a[:,-1])
```
```
tensor([ 3,  7, 11]) tensor([ 3,  7, 11])
```

As many some functions need NumPy arrays converting converting tensors is very helpful.

[22]:
```
b = a.numpy()
print(b)
```
```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

[23]:
```
print(torch.tensor(b))
```

```
tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
```

Linear Algebra

Recall that a lot of tensors will be matrices or vectors. Therefore, operations from linear algebra will be very helpful. At first we will take a look at dot products. Let

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{pmatrix} \in \mathbb{R}^p, y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_p \end{pmatrix} \in \mathbb{R}^p,$$

then

$$\langle x, y \rangle = x^T y := \sum_{i=1}^{p} x_i y_i,$$

which is implemented with `torch.dot`.

[24]:
```
x = torch.arange(3, dtype = torch.float32)
y = torch.ones(3, dtype = torch.float32)
print(x, y, torch.dot(x, y))
```

```
tensor([0., 1., 2.]) tensor([1., 1., 1.]) tensor(3.)
```

Let

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,p} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,p} \end{pmatrix} \in \mathbb{R}^{n \times p}$$

and $x \in \mathbb{R}^p$ then the matrix-vector product is defined as

$$Ax := \begin{pmatrix} \sum_{j=1}^{n} a_{1,j} x_j \\ \vdots \\ \sum_{j=1}^{n} a_{n,j} x_j \end{pmatrix} = \begin{pmatrix} a_{1,\bullet}^T x_j \\ \vdots \\ a_{n,\bullet}^T x_j \end{pmatrix} \in \mathbb{R}^n,$$

where $a_{i,\bullet}^T$ denotes the i-th row of A.

We can use `torch.mv`

[25]:
```
A = torch.tensor([[3,1.3,0],[2,5,0.5]])
print(A.shape,x.shape)
```

```
torch.Size([2, 3]) torch.Size([3])
```

[26]:
```
print(torch.mv(A, x))
```

```
tensor([1.3000, 6.0000])
```

Let $B \in \mathbb{R}^{p \times q}$. Then the generalization to matrix-matrix product is straightforward.

$$AB = \begin{pmatrix} a_{1,\bullet}^T b_{\bullet,1} & a_{1,\bullet}^T b_{\bullet,2} & \cdots & a_{1,\bullet}^T b_{\bullet,q} \\ a_{2,\bullet}^T b_{\bullet,1} & a_{2,\bullet}^T b_{\bullet,2} & \cdots & a_{2,\bullet}^T b_{\bullet,q} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,\bullet}^T b_{\bullet,n} & a_{1,\bullet}^T b_{\bullet,2} & \cdots & a_{n,\bullet}^T b_{\bullet,q} \end{pmatrix},$$

where $b_{\bullet,j}$ denotes the j-th column of B. This can be done with `torch.mm`.

[27]:
```
B = torch.ones((3,2))
print(torch.mm(A,B))
```

```
tensor([[4.3000, 4.3000],
        [7.5000, 7.5000]])
```

Let $x \in \mathbb{R}^n$ be vector. A **vector-norm** is a function $f : \mathbb{R}^n \to [0, \infty)$ with the following properties

1. For $\alpha \in \mathbb{R}$ it holds: $f(\alpha x) = |\alpha| f(x)$.
2. For $x, y \in \mathbb{R}^n$ it holds: $f(x + y) \leq f(x) + f(y)$.
3. It holds: $f(x) = 0 \Leftrightarrow x = 0 \in \mathbb{R}^n$.

If only properties 1. and 2. hold, f is called a seminorm.

The most common norm is the euclidean norm

$$\begin{aligned} \|\cdot\|_2 : \mathbb{R}^n &\to [0, \infty) \\ x &\mapsto \left(\sum_{i=1}^n x_i^2 \right)^{1/2}. \end{aligned}$$

[28]:
```
print(x,torch.norm(x))
```

```
tensor([0., 1., 2.]) tensor(2.2361)
```

The euclidean norm is a special case of the L_p-norm

$$\begin{aligned} \|\cdot\|_p : \mathbb{R}^n &\to [0, \infty) \\ x &\mapsto \left(\sum_{i=1}^n x_i^p \right)^{1/p}, \end{aligned}$$

where the L_1-norm is an important case. How does the L_1-norm behave differently than the L_2-norm?

A **matrix-norm** has the same properties as a vector norm, but is defined for matrices (in general norms are defined on vector spaces).

Let $A \in \mathbb{R}^{n \times m}$. The **Frobenius norm**

$$\begin{aligned} \|\cdot\|_F : \mathbb{R}^{n \times m} &\to [0, \infty) \\ A &\mapsto \left(\sum_{i=1}^n \sum_{j=1}^m a_{i,j}^2 \right)^{1/2}. \end{aligned}$$

[29]:
```
print(A,torch.norm(A))
```

```
tensor([[3.0000, 1.3000, 0.0000],
        [2.0000, 5.0000, 0.5000]]) tensor(6.3198)
```

The Frobenius norm is a special case of the $L_{p,q}$-norm

$$\begin{aligned} \|\cdot\|_{p,q} :&\mathbb{R}^{n\times m} \to [0,\infty) \\ &A \mapsto \left(\sum_{j=1}^{m}\left(\sum_{i=1}^{n} a_{i,j}^{p}\right)^{q/p}\right)^{1/q}. \end{aligned}$$

CHAPTER

TWO

BASICS OF MACHINE LEARNING

In general machine learning can be thought of as function approximation. Let $\mathcal{X}$ be the input space and $\mathcal{Y}$ be the target (or output) space. We would like to learn a mapping

$$\begin{aligned} f_0 : \mathcal{X} &\to \mathcal{Y} \\ x &\mapsto y. \end{aligned}$$

How $\mathcal{X}$,$\mathcal{Y}$ and $f_0(\cdot)$ look like depends on the actual learning problem, but mostly we are interested in

$$f_0(x) := \mathbb{E}[Y|X = x].$$

2.1 Definitions and Concepts

There are different subgroups of learning tasks. The most basic ones distinguishes between supervised and unsupervised learning.

2.1.1 Supervised Learning

The most common task in machine learning. In supervised learning the target variable is known during the training/learning process.

Examples:

- *Image Classification* - Given a picture, predict the content (e.g. cars).
- *Sales Prediction* - Given historical data, predict the sales for the next month.
- *Translating Text Sequences* - Given a text, predict the corresponding translation.

In this lecture, we will mostly focus on supervised learning tasks.

2.1.2 Unsupervised Learning

In contrast to supervised learning, there are no labels known during the learning process. In general, unsupervised learning is about finding interesting transformations or clusters in the data (for visualization, compression, denoising etc.)

Examples:

- *Clustering*
- *Dimensionality Reduction*

2.1.3 Semi-supervised Learning

Combinations of supervised and unsupervised algorithms. Often, the first step is to create the target variable from the input and afterwards apply a supervised learning algorithm.

From now on we will assume we are in a supervised learning setting: Given n samples of the input (y_i, x_i), we would like to learn the mapping f_0.

2.1.4 Regression vs. Classification

If the target variable y is quantitative (e.g. $\mathcal{Y} = \mathbb{R}$), the machine learning task is called a **regression** problem and if y is qualitative (e.g. $\mathcal{Y} = \{0, 1\}$ for binary classification), the machine learning task is called a **classification** problem.

Remark that the input variables can be quantitative or qualitative. In general, qualitative variable are encoded via dummy variables.

2.2 Loss Functions

We have specified that we would like to learn a mapping f_0. But how exactly is this mapping defined?

In general, we assume that f_0 is minimizing some average loss l, where

$$l : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^+$$

and

$$f_0 := \arg\min_f \mathbb{E}[l(Y, f(X))].$$

For a regression setting with the most common loss is the **squared loss**:

$$\begin{aligned} l_2 :&\mathbb{R} \times \mathbb{R} \to \mathbb{R}^+ \\ &(y_1, y_2) \mapsto (y_1 - y_2)^2 \end{aligned}$$

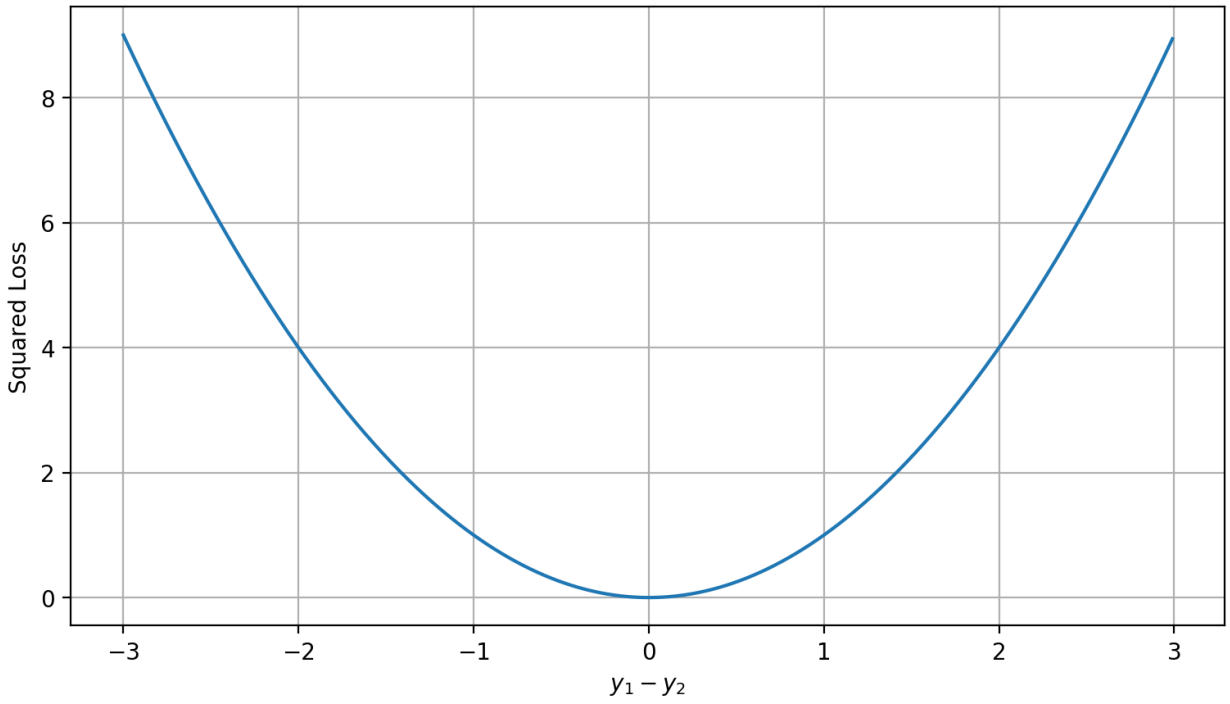


Figure 1: Squared loss.

It is possible to prove that for the l_2 loss it holds

$$\mathbb{E}[Y|X=x] = \arg\min_f \mathbb{E}[l_2(Y, f(X))] = \arg\min_f \mathbb{E}[(Y - f(X))^2].$$

Another important loss function is the **absolute loss**:

$$\begin{aligned} l_1 :& \mathbb{R} \times \mathbb{R} \to \mathbb{R}^+ \\ & (y_1, y_2) \mapsto |y_1 - y_2| \end{aligned}$$

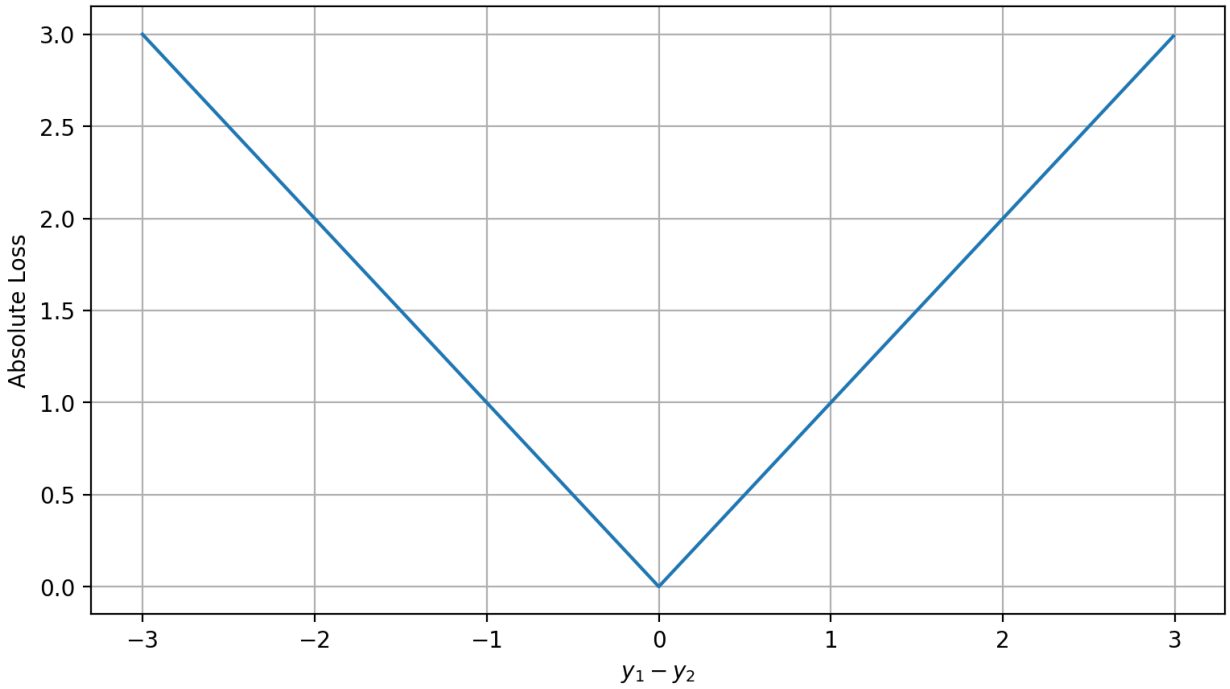


Figure 2: Absolute loss.

By minimizing the l_1 loss, we are trying to model the conditional median (or 0.5-quantile):

$$median(Y|X=x) = \arg\min_f \mathbb{E}[l_1(Y, f(X))] = \arg\min_f \mathbb{E}[|Y - f(X)|].$$

In a classification setting l_2 and l_1 are not applicable. Consider a binary classification problem with $\mathcal{Y} = \{0, 1\}$.

The function we would like to learn is the conditional probability

$$f(x) = P(Y = 1|X = x),$$

such that we can predict the value 1, if the conditional probability is larger than 0.5 (Bayes Classifier).

Motivated by maximum likelihood estimation, given n observations (Y_i, X_i), we would like to minimize

$$\begin{aligned} -\log\left(\prod_{i=1}^n P(Y_i|X_i)\right) &= \sum_{i=1}^n -\log\left(P(Y_i|X_i)\right) \\ &= \sum_{i=1}^n -Y_1 \log\left(P(Y_i = 1|X_i)\right) - (1 - Y_i)\log\left(1 - P(Y_i = 1|X_i)\right) \\ &= \sum_{i=1}^n \underbrace{-Y_1 \log\left(f(X_i)\right) - (1 - Y_i)\log\left(1 - f(X_i)\right)}_{=l(Y_i, f(X_i))}, \end{aligned}$$

where the loss function is given by

$$\begin{aligned} l :&\{0,1\} \times [0,1] \to \mathbb{R}^+ \\ &(y_1, y_2) \mapsto -y_1 \log(y_2) - (1-y_1)\log(1-y_2) \end{aligned}$$

This can be generalized to multiple classes. Categorical data is often represented with the **one-hot encoding**. Given K different categories the output is encodes in

$$y = (0, \dots, 0, 1, 0 \dots, 0)^T,$$

where the position of the 1 identifies the corresponding category.

Given $y \in \mathcal{Y}_K$, where $\mathcal{Y}_K$ denotes all one-hot encodings over K categories, the **cross entropy loss** is defined as

$$\begin{aligned} l :&\mathcal{Y}_K \times [0,1]^K \to \mathbb{R}^+ \\ &(y_1, y_2) \mapsto -\sum_{j=1}^{K} y_{1,j} \log(y_{2,j}) \end{aligned}$$

By minimizing the cross entropy loss we are trying to model the conditional probability $P(Y|X)$.

The final predicted class is then motivated by the Bayes Classifier:

$$\arg \max_{k \in \{1,\dots,K\}} P(Y = k | X = x).$$

2.3 Risk Minimization

All machine learning algorithms are learning a parametrized functions

$$\begin{aligned} f : \mathcal{X} \times \Theta &\to \mathcal{Y} \\ (x, \theta) &\mapsto y, \end{aligned}$$

where the f is a fixed function and $\theta \in \Theta$ is parameter vector (usually $\Theta = \mathbb{R}^d$, where d can be very large).

The challenge is to learn the parameter vector θ_0, which minimizes the risk (average loss)

$$\theta_0 := \arg\min_{\theta \in \Theta} \mathbb{E}[l(Y, f(X, \theta))].$$

Since we do **not** know the distribution of (Y, X), we instead rely on the empirical measure to estimate θ_0:

$$\hat{\theta} := \arg\min_{\theta \in \Theta} \mathbb{E}_n[l(Y, f(X, \theta))] = \arg\min_{\theta \in \Theta} \frac{1}{n} \sum_{i=1}^{n} l(Y_i, f(X_i, \theta)),$$

For example, consider the following univariate linear model

$$Y = a + bX + \varepsilon,$$

where $a, b \in \mathbb{R}$ and ε is an error term. Then it holds

$$\theta_0 = (a, b)^T = \arg\min_{(\alpha,\beta)^T \in \mathbb{R}^2} \mathbb{E}[(Y - \alpha - \beta X)^2].$$

In this setting, given n iid. observations of (Y_i, X_i), the **ordinary least squares estimate** is defined as

$$\hat{\theta} := (X^T X)^{-1} X^T Y,$$

where $X = (X_1, \dots, X_n)$ and $Y = (Y_1, \dots, Y_n)$.

Actually, this is an analytic solution to

$$\hat{\theta} = \arg \min_{(\alpha,\beta)^T \in \mathbb{R}^2} \mathbb{E}_n[(Y - \alpha - \beta X)^2] = \arg \min_{(\alpha,\beta)^T \in \mathbb{R}^2} \frac{1}{n} \sum_{i=1}^{n} (Y_i - \alpha - \beta X_i)^2.$$

In general, we are not able to solve

$$\hat{\theta} := \arg \min_{\theta \in \Theta} \mathbb{E}_n[l(Y, f(X, \theta))]$$

analytically. Instead deep learning (and a lot of other modern machine learning methods such as boosting) relies on gradient descent to try to minimize the empirical loss.

Remark that using gradient descent methods might result in finding local minima.

2.4 Gradient Descent

Let $f : \mathbb{R}^d \to \mathbb{R}$. The gradient of f is defined as

$$\nabla_x f(x) := \left(\frac{\partial f(x)}{\partial x_1}, \ldots, \frac{\partial f(x)}{\partial x_d} \right)^T.$$

Assume we start in some random point $x^{(0)}$ and would like to find the point

$$\tilde{x} := \arg \min_{x \in \mathcal{N}(x^{(0)})} f(x),$$

where $\mathcal{N}(x)$ is a neighborhood of x_0.

Since the gradient can be interpreted as the direction of the fastest increase of $f(x)$, the gradient descent methods updates the value of x_0 in the direction of the negative gradient

$$x^{(1)} = x^{(0)} - \nu \nabla_x f(x^{(0)}),$$

where $\nu \in \mathbb{R}^+$ is called the step size (typically chosen very small).

Iteratively repeating this procedure

$$x^{(n)} = x^{(n-1)} - \nu \nabla_x f(x^{(n-1)}),$$

will hopefully converge to a local minimum (can be guaranteed under certain assumptions as convexity and lipschitz-differentiability).

As gradient descent methods are crucial to deep learning algorithms, the tensor class supports automatic differentiation. In the following, we will introduce a simple example. Let

$$\begin{aligned} f : \mathbb{R}^2 &\to \mathbb{R} \\ (x_1, x_2)^T &\mapsto x_1^2 + x_2^2 \end{aligned}$$

and assume we start in the point $x^{(0)} = (-0.9, 0.8)^T$.

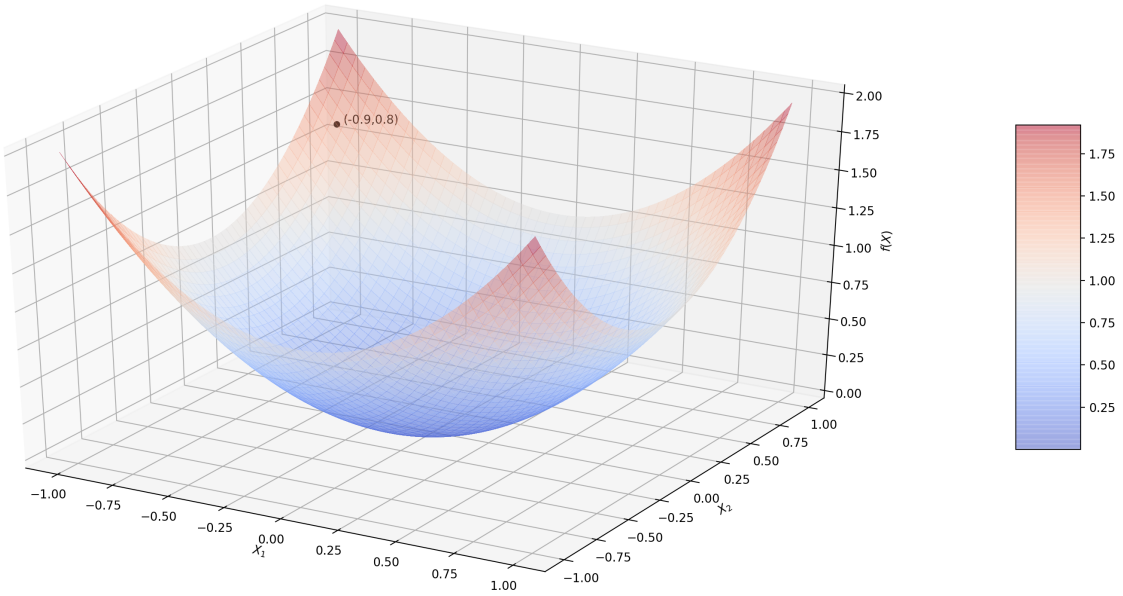


Figure 3: Gradient descent.

If we now calculate the gradient, we obtain

$$\nabla_x f(x^{(0)}) := \left(\frac{\partial f(x^{(0)})}{\partial x_1}, \frac{\partial f(x^{(0)})}{\partial x_2} \right)^T = (2x_1, 2x_2)^T = (-1.8, 1.6)^T .$$

Assuming a stepsize of $\nu_1 = 0.1$, we obtain

$$x^{(1)} = x^{(0)} - \nu_1 \nabla_x f(x^{(0)}) = (-0.9, 0.8)^T - 0.1\,(-1.8, 1.6)^T = (-0.72, 0.64)^T$$

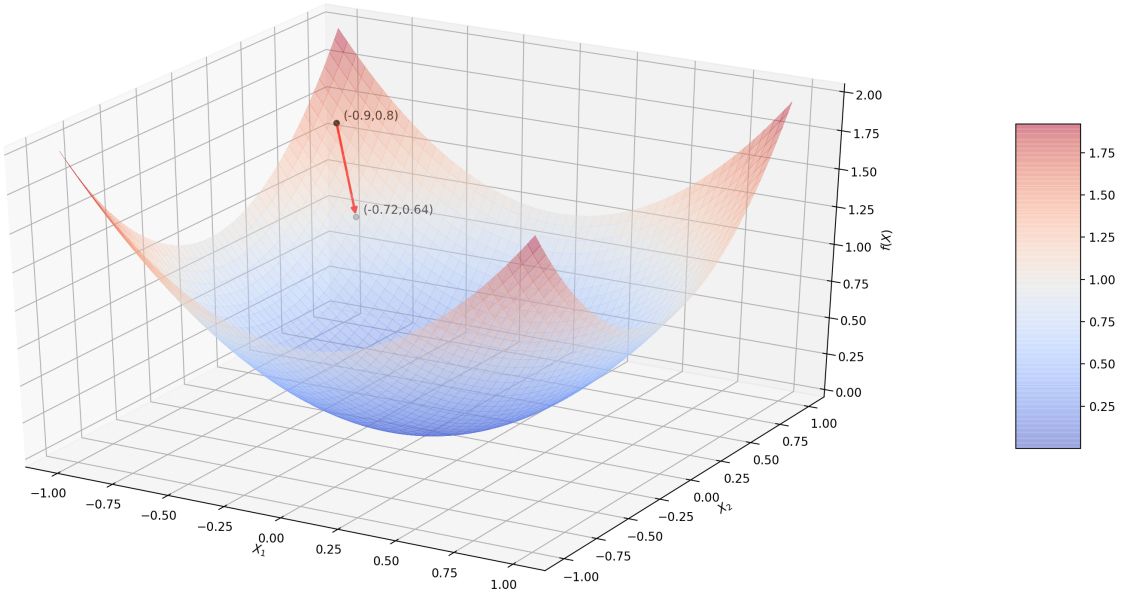


Figure 4: Gradient descent.

The next step would be (again with $\nu_2 = 0.1$)

$$x^{(2)} = x^{(1)} - \nu_2 \nabla_x f(x^{(1)}) = (-0.72, 0.64)^T - 0.1\,(-1.44, 1.28)^T = (-0.576, 0.512)^T$$

and by iterating this procedure we would converge to $\tilde{x} = (0, 0)^T$.

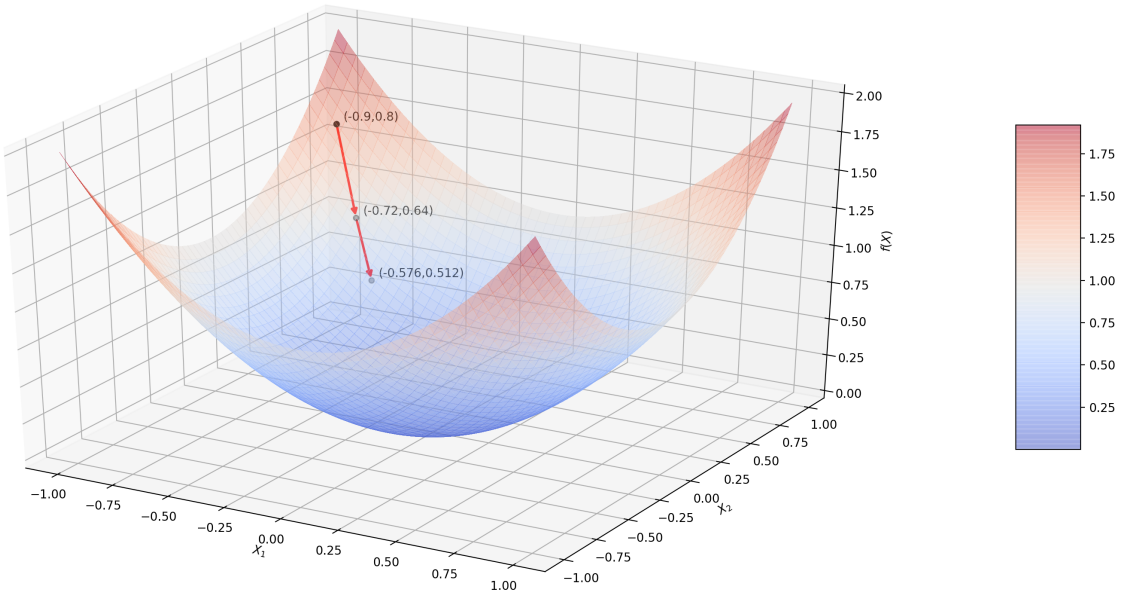


Figure 5: Gradient descent.

Instead of calculating the gradient by hand, we can instead use the automatic differentiation of pytorch. Here, automatic differentiation mean that pytorch builds an computational graph, to be able to calculate the gradient after a combination of "simple" operations.

[1]:
```
import torch
x_0 = torch.tensor([-.9,.8],requires_grad = True)
print(x_0)
```

```
tensor([-0.9000,  0.8000], requires_grad=True)
```

Here `requires_grad=True` specifies a place to store the gradient, since in deep learning models these are updated thousands of times (which will produce memory issues if we allocate new memory for every update).

In the next step, we will calculate $f(x)$:

[2]:
```
fx = sum(x_0**2)
print(fx)
```

```
tensor(1.4500, grad_fn=<AddBackward0>)
```

We can automatically calculate the gradient of $f(x_0)$ by calling the function for backpropagation:

[3]:
```
fx.backward()
print(x_0.grad)
```

```
tensor([-1.8000,  1.6000])
```

Here, the backpropagation traces the computational graph backwards to x_0 (in this case only one step) and calculates the partial derivatives. To detach the calculations from the computational graph we can use

[4]:
```
print(x_0.detach())
```

```
tensor([-0.9000,  0.8000])
```

2.5 Training a Machine Learning Model

In the previous sections, we have defined learning/training a machine learning model as learning parameters/weights, which minimize the empirical risk

$$\hat{\theta} := \arg\min_{\theta \in \Theta} \mathbb{E}_n[l(Y, f(X, \theta))].$$

In deep learning this is generally done with gradient descent methods due to the complicated form of $f(x, \theta)$.

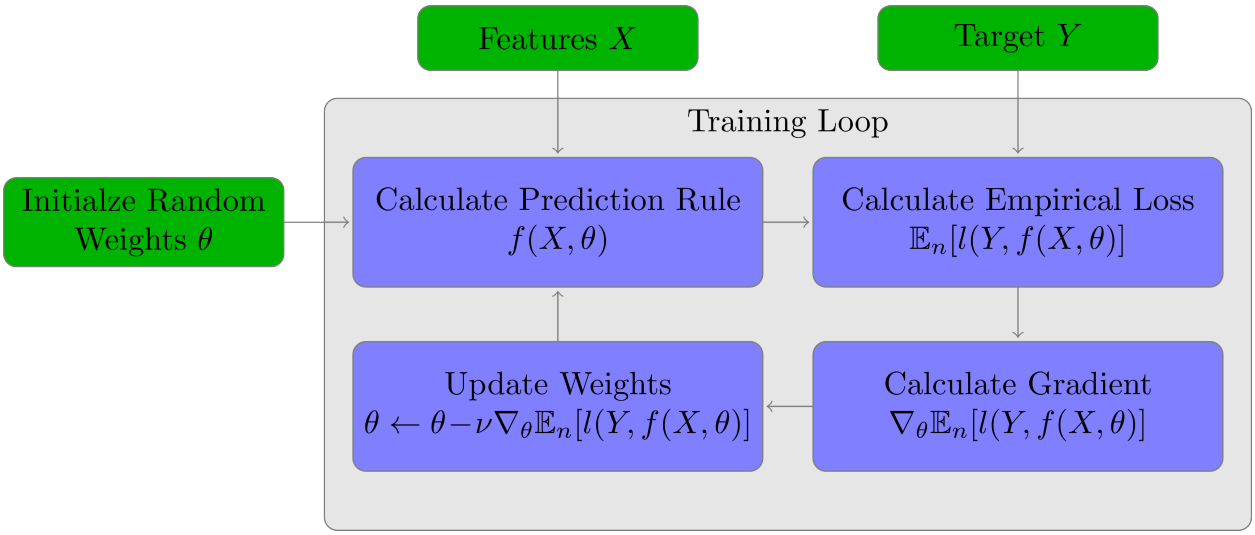


Figure 6: Training procedure.

2.5.1 Basic Linear Regression

In a basic multivariate linear regression model, we assume that the output Y is a linear function of the inputs $X = (1, X_1, \ldots, X_p)^T$

$$Y = w_0 + w_1 X_1 + \cdots + w_p X_p + \epsilon = w^T X + \epsilon.$$

Here, $w = (w_0, w_1, \ldots, w_p)^T$ is a vector of coefficients (or weights) and ε is an unobserved error term.

Under the assumption $\mathbb{E}[\varepsilon|X] = 0$ this corresponds to

$$\mathbb{E}[Y|X = x] = w^T x.$$

Therefore, we can interpret this as minimizing the average squared loss, with the weights w being the parameter vector θ_0 we would like to learn.

$$w = \arg\min_{b \in \mathbb{R}^{p+1}} \mathbb{E}[(Y - b^T X)^2]$$

For linear models there exists an analytical solution (under some mild assumptions). Instead of directly using the analytical solution to estimate the weights w, we apply the gradient descent method.

In the following, we implement basic linear regression including a small bivariate simulation example. We generate $n = 200$ observations with the following process

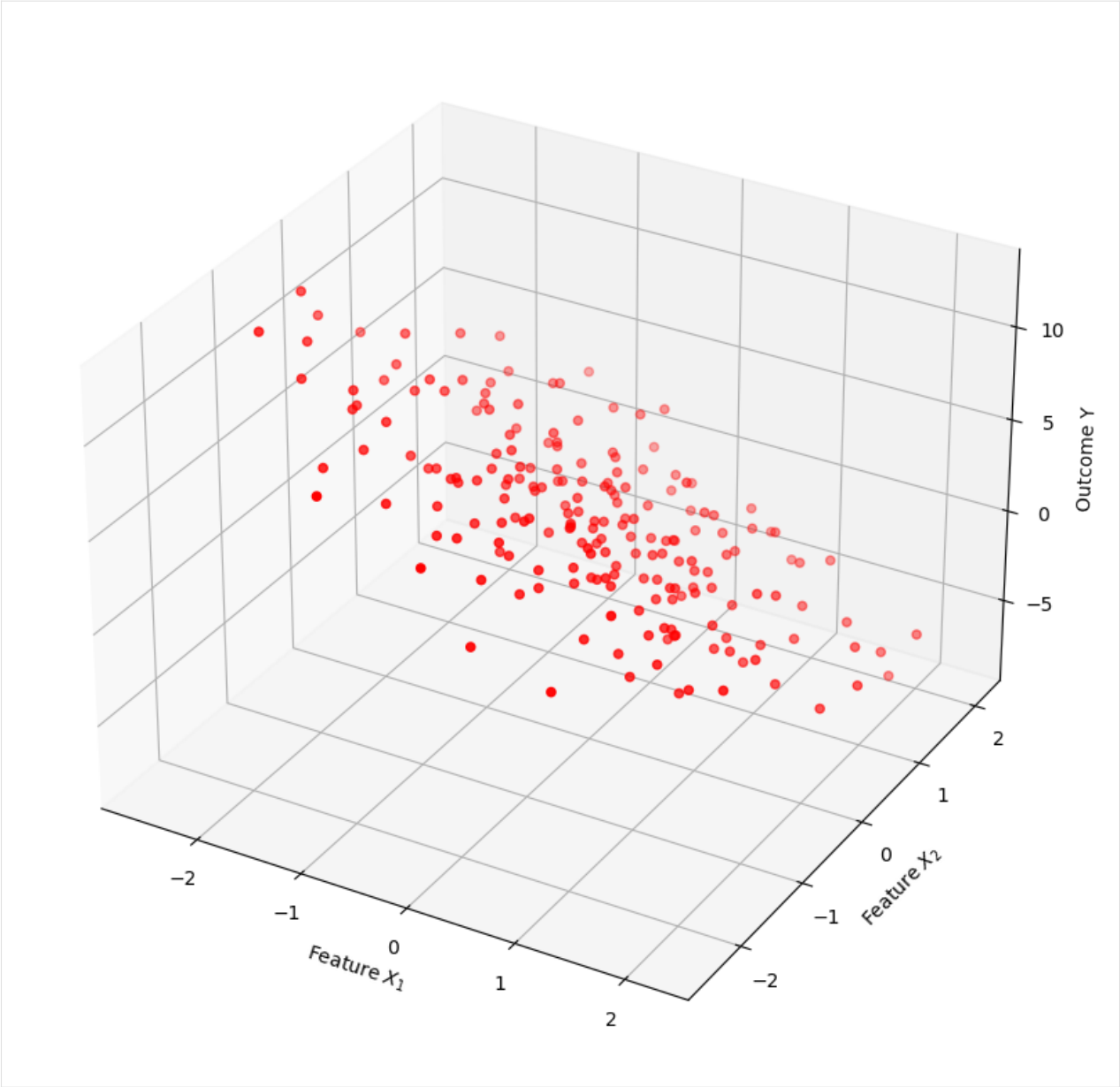
$$Y = 2 - 3X_1 - 1.5X_2 + \varepsilon,$$

where X_1, X_2 and ε follow independent standard normal distributions.

[5]:
```
import random; import torch
def dgp_linear_regression (weights,intercept, n):
    X = torch.normal(0,1,(n,len(weights)))
    Y = intercept + torch.mv(X,weights) + torch.normal(0,1,(1,n))
    return X,Y.reshape((-1,1))

true_weights = torch.tensor([-3.0, -1.5])
true_intercept = 2

#generate data
n_obs = 200
torch.manual_seed(42) #set a seed for replicability
features, labels = dgp_linear_regression(true_weights,true_intercept, n_obs)
```

Remark that we defined the intercept (or bias) separately (which is more convenient and saves computational resources). Let us take a look at the generated data.

[6]:
```
%matplotlib inline
import numpy as np; import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D; from mpl_toolkits.mplot3d import proj3d

fig = plt.figure(figsize=(20,10), dpi= 100); ax = fig.add_subplot(111,projection='3d')
x1 = features[:,0]
x2 = features[:,1]
y = labels
ax.scatter(x1,x2,y, marker='o', color='red')
ax.set_xlabel("Feature $X_1$"); ax.set_ylabel("Feature $X_2$"); ax.set_zlabel("Outcome Y
↪")
plt.show()
```



We will initialize our weights randomly with distribution $\mathcal{N}(0, 0.1)$ and set the bias to 0.

[7]:
```
weights = torch.normal(0,.01,size=(2,1),requires_grad= True)
intercept = torch.zeros(1,requires_grad=True)
print(weights)
print(intercept)
```

```
tensor([[-0.0143],
        [-0.0038]], requires_grad=True)
tensor([0.], requires_grad=True)
```

At first let us calculate the predicted values (with random weights).

[8]:
```
Y_hat = intercept + torch.mm(features,weights)
print(Y_hat[0:5])
```

```
tensor([[-0.0332],
        [-0.0049],
        [-0.0050],
        [ 0.0067],
        [ 0.0045]], grad_fn=<SliceBackward>)
```

Next, we want to calculate the empirical loss.

[9]:
```
training_loss = torch.mean((Y_hat-labels.reshape(Y_hat.shape))**2)
print(training_loss)
```

```
tensor(16.7457, grad_fn=<MeanBackward0>)
```

Here, `training_loss.backward()` will compute the gradient of loss with respect to all Tensors with `requires_grad=True`. The gradient takes the following form:

[10]:
```
training_loss.backward()
print(weights.grad)
print(intercept.grad)
```

```
tensor([[6.7407],
        [2.4970]])
tensor([-3.9418])
```

Next, we update the parameters

[11]:
```
learning_rate = 0.001
with torch.no_grad():
    weights -= learning_rate*weights.grad
    intercept -= learning_rate*intercept.grad

    # Manually zero the gradients after updating weights
    weights.grad.zero_()
    intercept.grad.zero_()

print(weights)
print(intercept)
```

```
tensor([[-0.0210],
        [-0.0063]], requires_grad=True)
tensor([0.0039], requires_grad=True)
```

Remark that we used `torch.no_grad()`, because the parameters have `requires_grad=True`, but we don't want to track this step in autograd. Additionally, we have to reset the gradients to zero, if we want to repeat this step multiple times.

An alternative way is to operate on `weights.data` and `weights.grad.data`. Here, *tensor.data* gives a tensor that shares the storage with tensor, but doesn't track history.

[12]:
```
weights.data -= learning_rate*weights.grad.data
intercept.data -= learning_rate*intercept.grad.data
# Manually zero the gradients after updating weights
weights.grad.data.zero_()
intercept.grad.data.zero_()
```

(continues on next page)

(continued from previous page)

```
print(weights)
print(intercept)
```

```
tensor([[-0.0210],
        [-0.0063]], requires_grad=True)
tensor([0.0039], requires_grad=True)
```

We will now combine all of this in a loop (with a slightly larger learning rate) to iteratively minimize the empirical risk

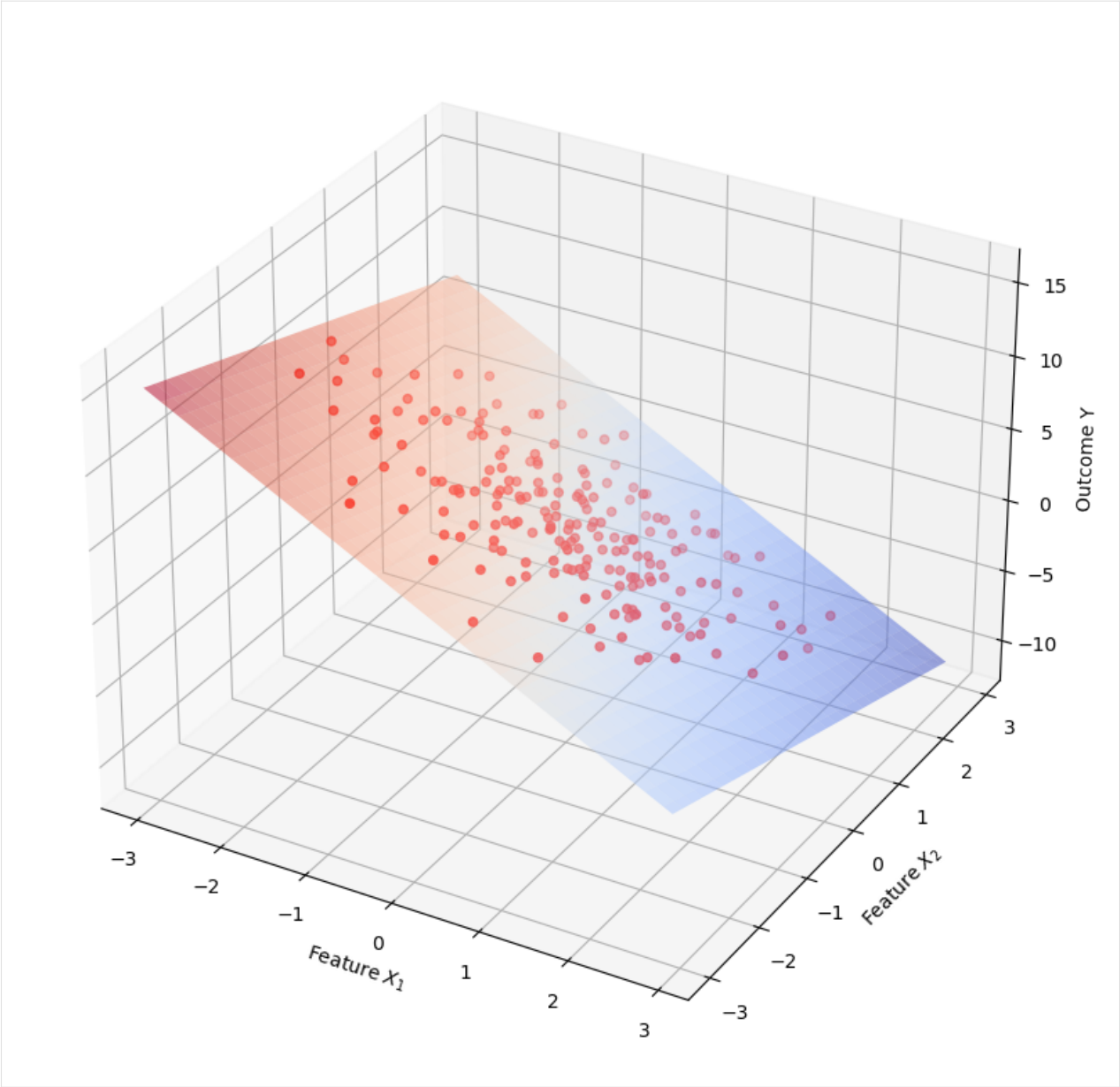
[13]:
```
learning_rate = 0.03
for i in range(500):
    Y_hat = intercept + torch.matmul(features,weights)
    training_loss = torch.mean((Y_hat-labels.reshape(Y_hat.shape))**2)
    training_loss.backward()
    with torch.no_grad():
        weights -= learning_rate*weights.grad
        intercept -= learning_rate*intercept.grad

        # Manually zero the gradients after updating weights
        weights.grad.zero_()
        intercept.grad.zero_()

print(weights)
print(intercept)
```

```
tensor([[-3.0315],
        [-1.4270]], requires_grad=True)
tensor([1.9621], requires_grad=True)
```

Our parameters are quite close to the true values. Lets take a look at our final model

[14]:
```
from matplotlib import cm
xs = np.tile(np.arange(61), (61,1))
ys = np.tile(np.arange(61), (61,1)).T
base_grid = np.arange(-3,3,0.1)
xs = np.tile(base_grid, (len(base_grid),1))
ys = np.tile(base_grid, (len(base_grid),1)).T

zs = xs*weights[0].detach().numpy()+ys*weights[1].detach().numpy()+intercept.detach().
↪numpy()
ax.plot_surface(xs,ys,zs, alpha=0.5, cmap =cm.coolwarm)
fig
```

[14]:



The linear model can be represented by the following graph structure.

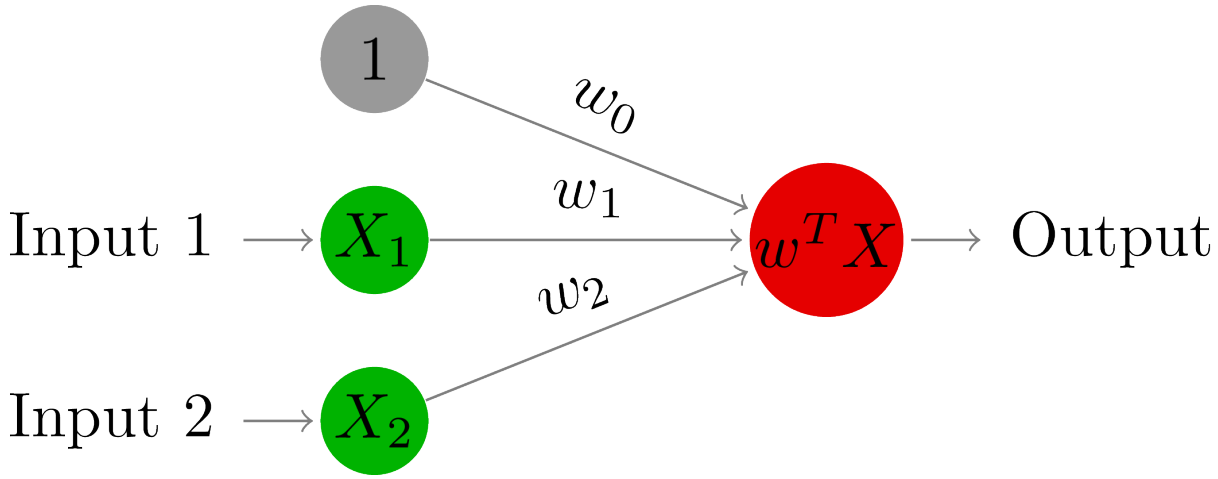


Figure 7: Linear Model.

2.5.2 Stochastic Gradient Descent

The training loop from the previous chapter, displays the basic idea of training a machine learning model with gradient descent.

In practice, if the number of observations n is really large and the function $f(x, \theta)$ is more complex we will run into computational problems really fast. In general, the calculation of the gradient on all samples is too costly.

An easy solution is to use a stochastic version of the gradient descent method called **stochastic gradient descent**.

Instead of using all samples in each iteration of the training loop, we will only use a small random subset of the observations, where the subset is called **batch**. Stochastic gradient descent generally refers to only using a single observation for each update step (batch size of one). Very small subsets are usually called **mini-batch**, whereas using the whole sample is called **full batch**.

The basic idea is that on a random sample the gradient is still a unbiased (but noisy) estimate of the true gradient

$$\nabla_\theta \mathbb{E}[l(Y, f(X, \theta))].$$

In practice deep learning is done in a series of **epochs**, where one **epoch** consists of one backpropagation pass over the whole dataset.

In the previous example, we did use **full batch** and each iteration corresponded to one epoch.

General Training Procedure:

1. Set the number of epochs and the batch size.
2. Randomly devide the training set into batches of the specified batch size.
3. Iterate the training loop over all batches (each iteration a batch is randomly drawn without replacement).
4. Repeat steps 2 and 3 for each epoch.

We will repeat the linear regression example with stochastic gradient descent. At first, we will need a data loader, which automatically provides the batches.

[15]:
```
def data_iter(batch_size, features, labels):
    n = len(labels)
    indices = list(range(n))
    random.shuffle(indices)
    for i in range(0,n,batch_size):
```

(continues on next page)

(continued from previous page)

```
        batch_indices = torch.tensor(indices[i:min(i+batch_size,n)])
        yield features[batch_indices], labels[batch_indices]

for X,Y in data_iter(5, features, labels): #test the batches
    print(X, "\n", Y)
    break
```

```
tensor([[ 0.7755,  2.0265],
        [ 0.7440, -0.4816],
        [-1.4364, -1.1299],
        [ 0.2649,  1.2732],
        [-0.0127,  0.2408]])
 tensor([[-4.4026],
        [ 0.3873],
        [ 7.3203],
        [-1.3465],
        [ 1.7540]])
```

Now we can repeat the model training.

[16]:
```
weights = torch.normal(0,.01,size=(2,1),requires_grad= True)
intercept = torch.zeros(1,requires_grad=True)
learning_rate = 0.03
batch_size = 10
n_epochs = 5
for epoch in range(n_epochs):
    for X,Y in data_iter(batch_size, features, labels):
        Y_hat = intercept + torch.matmul(X,weights)
        training_loss = torch.mean((Y_hat-Y.reshape(Y_hat.shape))**2)
        training_loss.backward()
        with torch.no_grad():
            weights -= learning_rate*weights.grad
            intercept -= learning_rate*intercept.grad
            weights.grad.zero_()
            intercept.grad.zero_()

print(weights)
print(intercept)
```

```
tensor([[-3.0272],
        [-1.4357]], requires_grad=True)
tensor([1.9261], requires_grad=True)
```

There exist a lot of different variants of the stochastic gradient descent algorithms, which are explained in more detail in later sections.

2.5.3 Building Linear Regression using High-Level APIs

Next, we will show you how to implement the linear regression model concisely by using high-level APIs of deep learning frameworks. The first step is to replace our own data loader by the existing API in a framework to read data. We pass in features and labels as arguments and specify `batch_size` when instantiating a data iterator object. Besides, the boolean value `is_train` indicates whether or not we want the data iterator object to shuffle the data on each epoch (pass through the dataset).

[17]:
```
from torch.utils import data
def load_array(data_arrays, batch_size, is_train=True):
    """Construct a PyTorch data iterator."""
    dataset = data.TensorDataset(*data_arrays)
    return data.DataLoader(dataset, batch_size, shuffle=is_train)

batch_size = 10
data_iter = load_array((features, labels), batch_size)
```

Now we can use `data_iter` in much the same way as we called the `data_iter` function in the previous section. To verify that it is working, we can read and print the first batch of examples. Here, we use iter to construct a Python iterator and use next to obtain the first item from the iterator.

[18]:
```
print(next(iter(data_iter)))
```

```
[tensor([[ 0.3488,  0.9676],
        [ 0.8008,  1.6806],
        [ 0.5750, -0.6417],
        [-1.0546,  1.2780],
        [-2.5095,  0.4880],
        [ 0.0109, -0.3387],
        [ 0.3189, -0.4245],
        [ 0.4038, -0.7140],
        [-1.4364, -1.1299],
        [-0.2516,  0.8599]]), tensor([[ 2.3363],
        [-1.7777],
        [ 1.9061],
        [ 4.6668],
        [ 7.4127],
        [ 3.1654],
        [ 1.1641],
        [ 3.8666],
        [ 7.3203],
        [ 2.2417]])]
```

[19]:
```
from torch import nn
lin_reg = nn.Sequential(nn.Linear(2, 1))
```

Here, `nn` is an abbreviation for neural networks and `nn.Linear(2,1)` applies a linear transformation as in the previous represented graph.

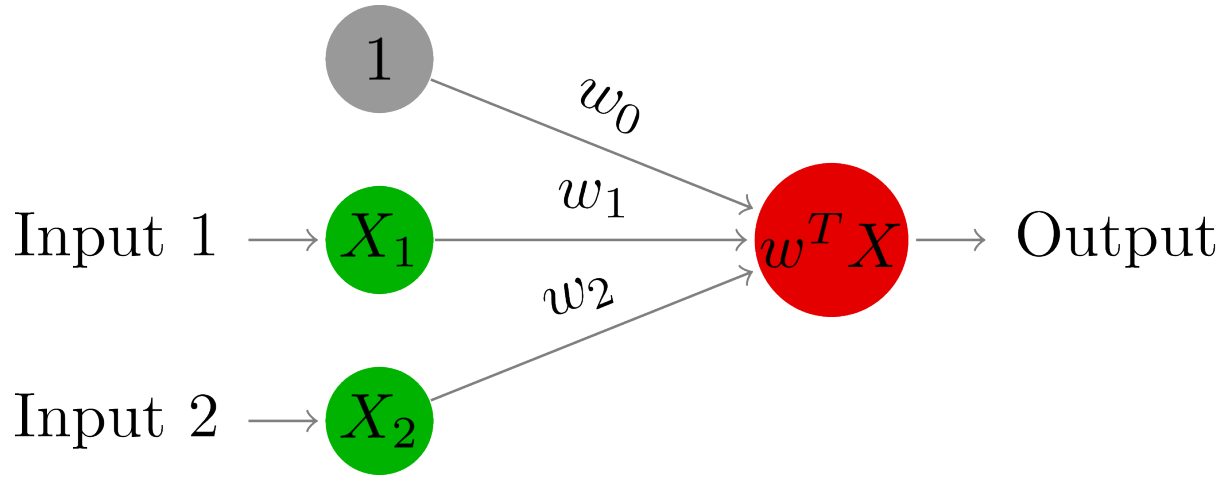


Figure 8: Linear Model.

We still have to initalize the weights (with $w_0 = bias$).

[20]:
```
lin_reg[0].weight.data.normal_(0, 0.01)
lin_reg[0].bias.data.fill_(0)
print(lin_reg[0].weight.data)
print(lin_reg[0].bias.data)
```

```
tensor([[0.0104, 0.0002]])
tensor([0.])
```

We can specify our loss, by using already existing classes:

[21]:
```
loss = nn.MSELoss()
```

For optimization we will use `torch.optim`, which is a package implementing various popular optimization algorithms.

[22]:
```
trainer = torch.optim.SGD(lin_reg.parameters(), lr=0.01)
```

Here, `torch.optim.SGD` constructs a stochastic gradient descent optimizer for all weights of `lin_reg` with the `lr` being the learning rate.

[23]:
```
num_epochs = 5
for epoch in range(num_epochs):
    for X, y in data_iter:
        training_loss = loss(lin_reg(X) ,y)
        trainer.zero_grad() #reset the gradient to zero
        training_loss.backward()
        trainer.step()

    training_loss = loss(lin_reg(features), labels) #calculate the loss on the whole␣
↪training sample
    print(f'epoch {epoch + 1}, loss {training_loss:f}')
```

```
epoch 1, loss 7.699501
epoch 2, loss 3.822890
epoch 3, loss 2.175913
epoch 4, loss 1.463140
epoch 5, loss 1.161647
```

[24]:
```
w = lin_reg[0].weight.data
print('Estimated weights w:',  w)
b = lin_reg[0].bias.data
print('Estimated intercept/bias:', b)
```

```
Estimated weights w: tensor([[-2.7299, -1.1695]])
Estimated intercept/bias: tensor([1.6950])
```

2.6 Assessing Model Performance

We have learned our machine learning model by minimizing the empirical risk

$$\mathbb{E}_n[l(Y, f(X, \theta))].$$

But our goal is to obtain a model which performs well (having a low loss on average) on unseen data.

The following simple example highlights, why this might fail.

In most cases the target function f_0 is much more complicated than just a plain linear function of the raw inputs.

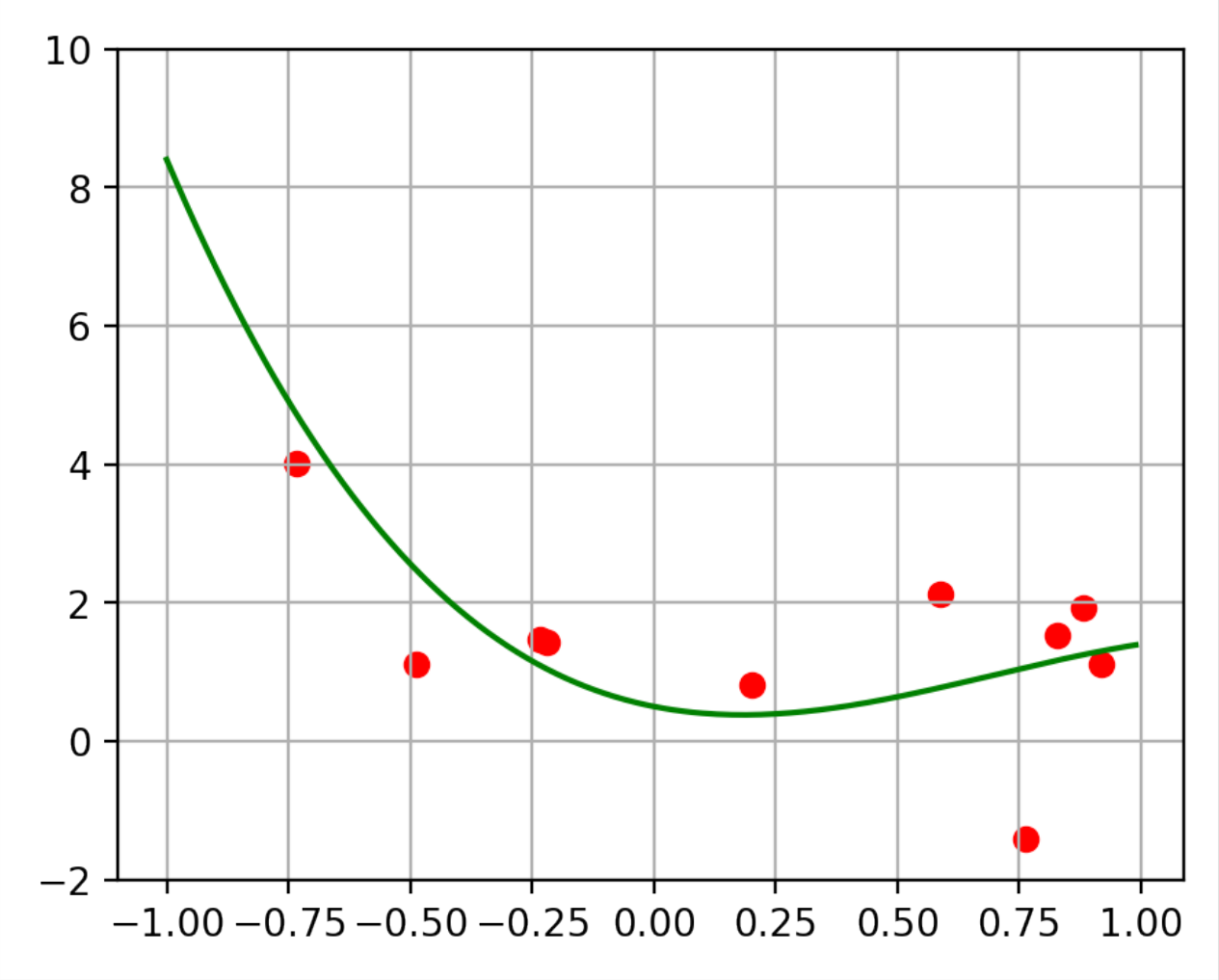
Let uns consider the following univariate example:

$$Y = 0.5 - 1.4X + 4.4X^2 - 2.1X^3 + \varepsilon,$$

where $X \sim \mathcal{U}[-1, 1)$ and ε is drawn from a standard normal distribution.

[25]:
```
n = 10 #number of observations
torch.manual_seed(42) #set a seed for replicability

features = 2*torch.rand(n,1)-1
labels = 0.5-1.4*features + 4.4*features**2 - 2.1*features**3 + torch.normal(0,1,(1,n)).
↪reshape(-1,1)
```

[26]:
```
fig = plt.figure(figsize=(5,4), dpi= 200)
plt.scatter(features, labels, c='red')
x = torch.arange(-1,1,0.01)
fx = 0.5-1.4*x + 4.4*x**2 -2.1*x**3
plt.plot(x,fx,c='green')
plt.grid(True)
plt.ylim((-2, 10))
plt.show()
```

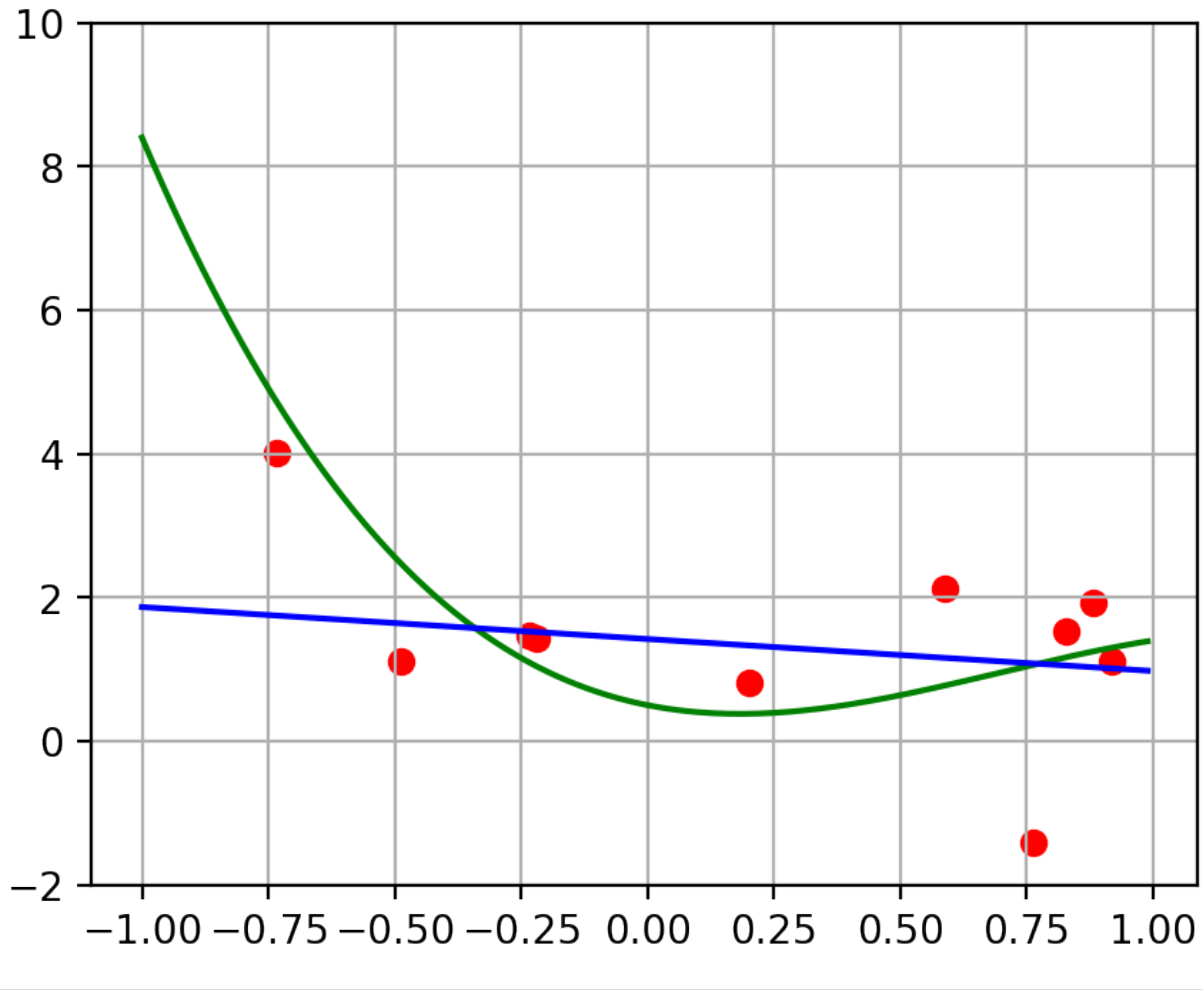


A linear regression is too simple capture the data generating process.

[27]:
```
batch_size = 10
data_iter = load_array((features, labels), batch_size)
lin_reg = nn.Sequential(nn.Linear(1, 1))
lin_reg[0].weight.data.normal_(0, 0.01)
lin_reg[0].bias.data.fill_(0)
loss = nn.MSELoss()
trainer = torch.optim.SGD(lin_reg.parameters(), lr=0.03)
num_epochs = 50
for epoch in range(num_epochs):
    for X, y in data_iter:
        training_loss = loss(lin_reg(X) ,y)
        trainer.zero_grad() #reset the gradient to zero
        training_loss.backward()
        trainer.step()
training_loss = loss(lin_reg(features), labels) #calculate the loss on the whole␣
↪training sample
print(f'epoch {epoch + 1}, loss {training_loss:f}')
```

```
epoch 50, loss 1.384482
```

[28]:
```
fig = plt.figure(figsize=(5,4), dpi= 200)
plt.scatter(features, labels, c='red')
fx_hat= lin_reg[0].bias.data.detach()+lin_reg[0].weight.data.detach()*x
plt.plot(x,fx,c='green')
plt.plot(x,fx_hat.reshape(-1,1),c='blue')
plt.grid(True)
plt.ylim((-2, 10))
plt.show()
```



This is called **underfitting** since the machine learning model is too simple and therefore not able to approximate f_0.

We can increase the capability of linear regression by using polynomial regression, which increases the flexibility of our machine learning model. But if the flexability is too large, the training process might not converge to a helpful solution, but instead starts to "learn" the noise in the data.

[29]:
```
max_degr = 10
polynomial_features = features
for degr in range(1,max_degr):
    polynomial_features = torch.cat((polynomial_features,features**(degr+1)), dim = 1)
```

[30]:
```
batch_size = 10
```

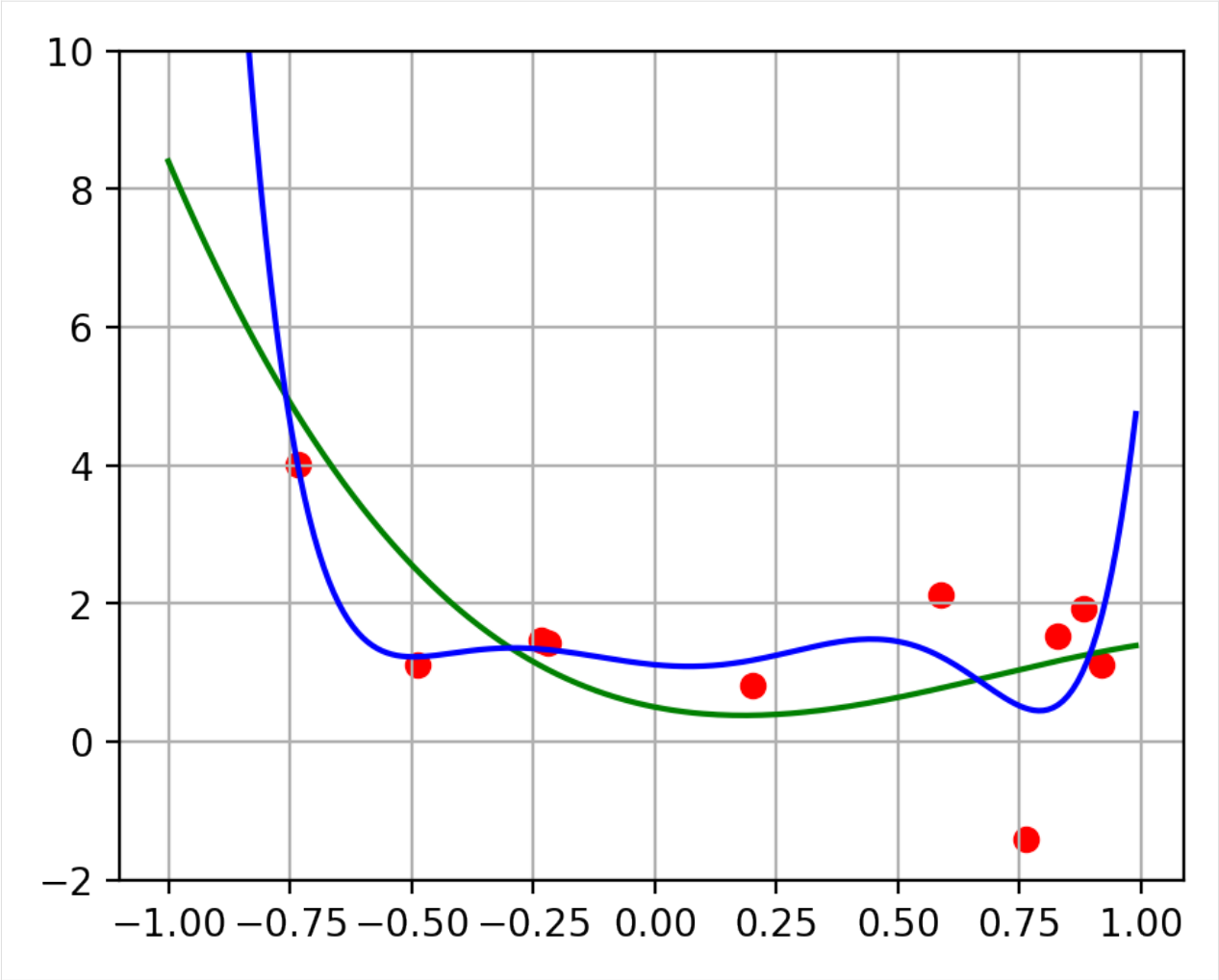
(continues on next page)

(continued from previous page)

```
data_iter = load_array((polynomial_features, labels), batch_size)
lin_reg = nn.Sequential(nn.Linear(max_degr, 1))
lin_reg[0].weight.data.normal_(0, 0.01)
lin_reg[0].bias.data.fill_(0)
loss = nn.MSELoss()
trainer = torch.optim.SGD(lin_reg.parameters(), lr=.4)
num_epochs = 50000
for epoch in range(num_epochs):
    for X, y in data_iter:
        training_loss = loss(lin_reg(X) ,y)
        trainer.zero_grad() #reset the gradient to zero
        training_loss.backward()
        trainer.step()
training_loss = loss(lin_reg(polynomial_features), labels) #calculate the loss on the␣
↪whole training sample
print(f'epoch {epoch + 1}, loss {training_loss:f}')
```

```
epoch 50000, loss 0.675656
```

[31]:
```
fig = plt.figure(figsize=(5,4), dpi= 200)
plt.scatter(features, labels, c='red')
fx_hat_2 = lin_reg[0].bias.data.detach()+lin_reg[0].weight.data[0,0].detach()*x
for degr in range(1,max_degr):
    fx_hat_2 +=lin_reg[0].weight.data[0,degr].detach()*x**(degr+1)
plt.plot(x,fx,c='green')
plt.plot(x,fx_hat_2.reshape(-1,1),c='blue')
plt.grid(True)
plt.ylim((-2, 10))
plt.show()
```



This process of "learning" noise is called **overfitting**. For example in the context of image recognition this might correspond to learning irrelevant artifacts like backgrounds or shadows.

For the mean-squared error under- and overfitting can be motivated with the **bias-variance-tradeoff**:

$$\mathbb{E}[(Y - f(X,\theta))^2] = \underbrace{\mathbb{E}\left[\left(\mathbb{E}[f(X,\theta)] - f(X,\theta)\right)^2\right]}_{\text{Variance}} + \underbrace{\left(\mathbb{E}[Y|X] - \mathbb{E}[f(X,\theta)]\right)^2}_{\text{Bias}^2}$$
$$+ \underbrace{\text{Var}\big(Y - \mathbb{E}[Y|X]\big)}_{\text{irreducible error}}$$

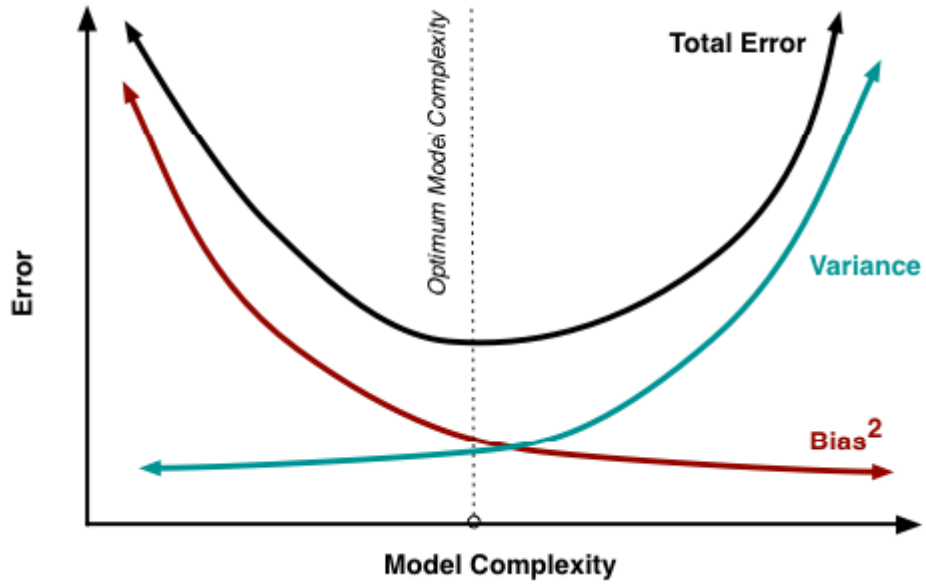


Figure 9: Bias-Variance tradeoff.

To avoid under- and overfitting it is important to monitor the model performance during the training procedure. In general, this is done by **sample splitting**.

Sample splitting refers to creating several subsets of your sample, which are used for different purposes.

- **Training Set**: Used to train the weights of the machine learning algorithm.
- **Testing Set**: Used to evaluate the risk of the final model.
- **Validation Set**: Sometimes the training set is split further to monitor the training process and choose some tuning parameters.

The terms testing set and validation set are not uniquely defined and sometimes used interchangeably.

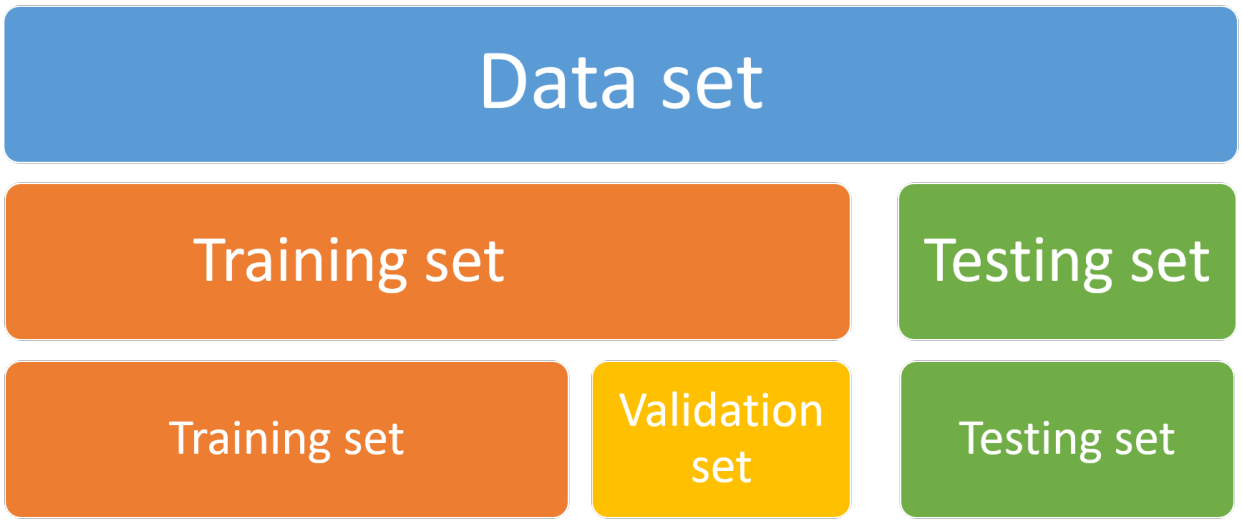


Figure 10: Sample splitting.

The test set simulates, how the model generalizes to unseen/new data.

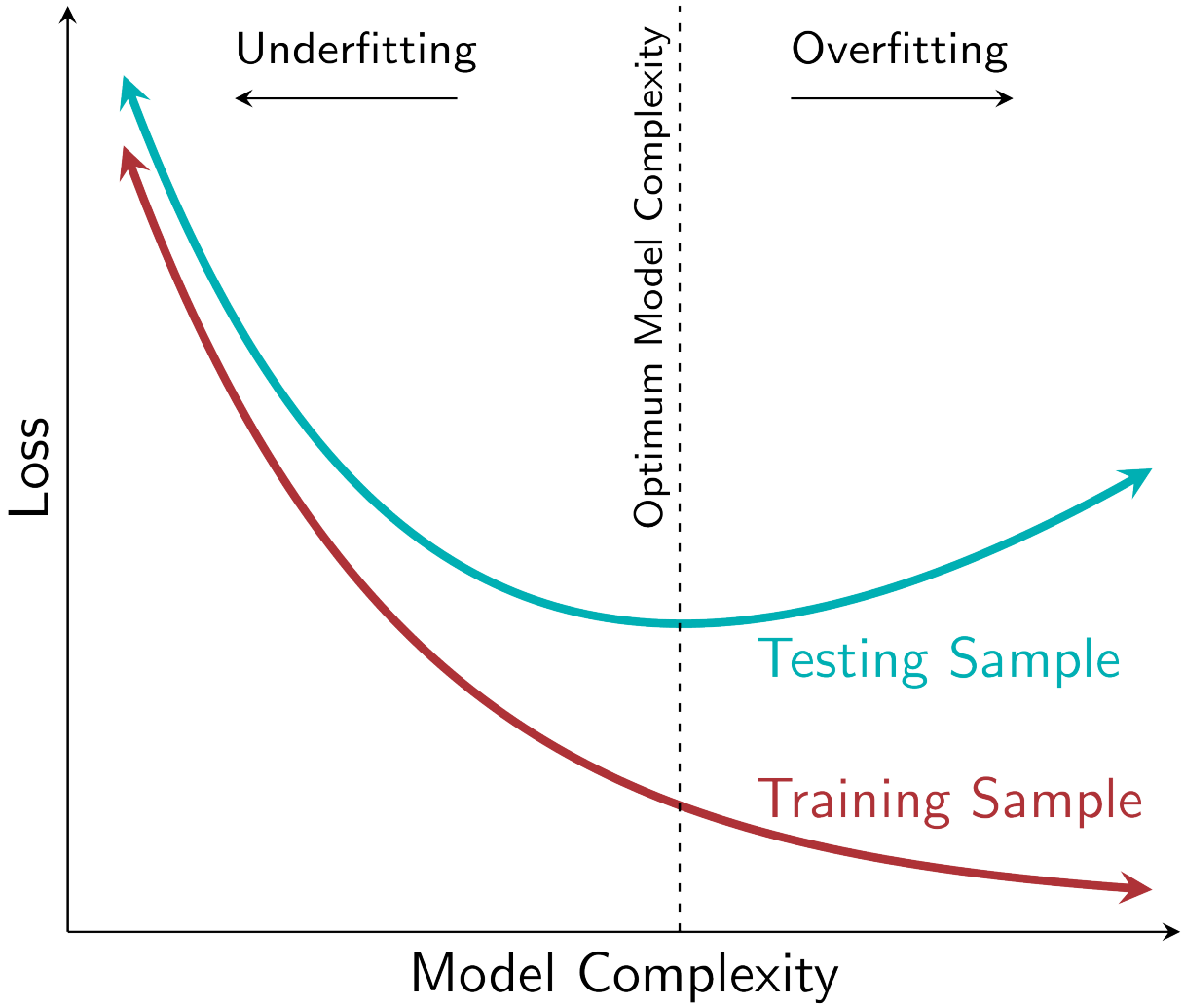


Figure 11: Under- and Overfitting.

We now revisit the example above. Suppose we have had an equally sized test sample which we did not use for training.

[32]:
```
n_test = 20 #number of observations
test_features = 2*torch.rand(n,1)-1
test_polynomial_features = test_features
for degr in range(1,max_degr):
    test_polynomial_features = torch.cat((test_polynomial_features,test_
↪features**(degr+1)), dim = 1)

test_labels = 0.5-1.4*test_features + 4.4*test_features**2 - 2.1*test_features**3 +␣
↪torch.normal(0,1,(1,n)).reshape(-1,1)

print(f'epoch {epoch + 1}, loss {training_loss:f}') #recap the training loss
test_loss = loss(lin_reg(test_polynomial_features), test_labels) #calculate the loss on␣
↪the whole training sample
print(f'epoch {epoch + 1}, loss {test_loss:f}')
```

```
epoch 50000, loss 0.675656
epoch 50000, loss 45.393623
```