

## ✓ COSE474-2024F: Deep Learning

### ✓ 0.1 Installation

`! pip install d2l==1.0.3`

```

Collecting d2l==1.0.3
  Downloading d2l-1.0.3-py3-none-any.whl.metadata (556 bytes)
Collecting jupyter==1.0.0 (from d2l==1.0.3)
  Downloading jupyter-1.0.0-py2.py3-none-any.whl.metadata (995 bytes)
Collecting numpy==1.23.5 (from d2l==1.0.3)
  Downloading numpy-1.23.5-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_
Collecting matplotlib==3.7.2 (from d2l==1.0.3)
  Downloading matplotlib-3.7.2-cp310-cp310-manylinux_2_17_x86_64.manylinux2
Collecting matplotlib-inline==0.1.6 (from d2l==1.0.3)
  Downloading matplotlib-inline-0.1.6-py3-none-any.whl.metadata (2.8 kB)
Collecting requests==2.31.0 (from d2l==1.0.3)
  Downloading requests-2.31.0-py3-none-any.whl.metadata (4.6 kB)
Collecting pandas==2.0.3 (from d2l==1.0.3)
  Downloading pandas-2.0.3-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_
Collecting scipy==1.10.1 (from d2l==1.0.3)
  Downloading scipy-1.10.1-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_
58.9/58.9 kB 3.8 MB/s eta 0:0
Requirement already satisfied: notebook in /usr/local/lib/python3.10/dist-p
Collecting qtconsole (from jupyter==1.0.0->d2l==1.0.3)
  Downloading qtconsole-5.6.0-py3-none-any.whl.metadata (5.0 kB)
Requirement already satisfied: jupyter-console in /usr/local/lib/python3.10
Requirement already satisfied: nbconvert in /usr/local/lib/python3.10/dist-
Requirement already satisfied: ipykernel in /usr/local/lib/python3.10/dist-
Requirement already satisfied: ipywidgets in /usr/local/lib/python3.10/dist
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.1
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/di
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/d
Collecting pyparsing<3.1,>=2.3.1 (from matplotlib==3.7.2->d2l==1.0.3)
  Downloading pyparsing-3.0.9-py3-none-any.whl.metadata (4.2 kB)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/pytho
Requirement already satisfied: traitlets in /usr/local/lib/python3.10/dist-
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/di
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/python3.10/
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/p
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/di
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-p

```

```

Requirement already satisfied: ipython-genutils in /usr/local/lib/python3.1
Requirement already satisfied: ipython>=5.0.0 in /usr/local/lib/python3.10/
Requirement already satisfied: jupyter-client in /usr/local/lib/python3.10/
Requirement already satisfied: tornado>=4.2 in /usr/local/lib/python3.10/di
Requirement already satisfied: widgetsnbextension~=3.6.0 in /usr/local/lib/
Requirement already satisfied: jupyterlab-widgets>=1.0.0 in /usr/local/lib/
Requirement already satisfied: prompt-toolkit!=3.0.0,!<3.0.1,<3.1.0,>=2.0.0
Requirement already satisfied: pygments in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: lxml in /usr/local/lib/python3.10/dist-packa
Requirement already satisfied: beautifulsoup4 in /usr/local/lib/python3.10/
Requirement already satisfied: bleach in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: defusedxml in /usr/local/lib/python3.10/dist
Requirement already satisfied: entrypoints>=0.2.2 in /usr/local/lib/python3
Requirement already satisfied: Jinja2>=3.0 in /usr/local/lib/python3.10/dis
Requirement already satisfied: jupyter-core>=4.7 in /usr/local/lib/python3.
Requirement already satisfied: jupyterlab-pygments in /usr/local/lib/python
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10
Requirement already satisfied: mistune<2,>=0.8.1 in /usr/local/lib/python3.
Requirement already satisfied: nbclient>=0.5.0 in /usr/local/lib/python3.10
Requirement already satisfied: nbformat>=5.1 in /usr/local/lib/python3.10/d
Requirement already satisfied: pandocfilters>=1.4.1 in /usr/local/lib/pytho
Requirement already satisfied: tinycss2 in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: pyzmq<25,>=17 in /usr/local/lib/python3.10/d
Requirement already satisfied: argon2-cffi in /usr/local/lib/python3.10/dis
Requirement already satisfied: nest-asyncio>=1.5 in /usr/local/lib/python3.
Requirement already satisfied: Send2Trash>=1.8.0 in /usr/local/lib/python3.
Requirement already satisfied: terminado>=0.8.3 in /usr/local/lib/python3.1
Requirement already satisfied: prometheus-client in /usr/local/lib/python3.
Requirement already satisfied: nbclassic>=0.4.7 in /usr/local/lib/python3.1
Collecting qtpy>=2.4.0 (from qtconsole->jupyter==1.0.0->d2l==1.0.3)
  Downloading QtPy-2.4.1-py3-none-any.whl.metadata (12 kB)
Requirement already satisfied: setuptools>=18.5 in /usr/local/lib/python3.1
Collecting jedi>=0.16 (from ipython>=5.0.0->ipykernel->jupyter==1.0.0->d2l=
  Using cached jedi-0.19.1-py2.py3-none-any.whl.metadata (22 kB)
Requirement already satisfied: decorator in /usr/local/lib/python3.10/dist-
Requirement already satisfied: pickleshare in /usr/local/lib/python3.10/dis
Requirement already satisfied: backcall in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: pexpect>4.3 in /usr/local/lib/python3.10/dis
Requirement already satisfied: platformdirs>=2.5 in /usr/local/lib/python3.
Requirement already satisfied: notebook-shim>=0.2.3 in /usr/local/lib/pytho
Requirement already satisfied: fastjsonschema>=2.15 in /usr/local/lib/pytho
Requirement already satisfied: jsonschema>=2.6 in /usr/local/lib/python3.10
Requirement already satisfied: wcwidth in /usr/local/lib/python3.10/dist-pa
Requirement already satisfied: ptyprocess in /usr/local/lib/python3.10/dist
Requirement already satisfied: argon2-cffi-bindings in /usr/local/lib/pytho
Requirement already satisfied: soupsieve>1.2 in /usr/local/lib/python3.10/d
Requirement already satisfied: webencodings in /usr/local/lib/python3.10/di
Requirement already satisfied: parso<0.9.0,>=0.8.3 in /usr/local/lib/python
Requirement already satisfied: attrs>=22.2.0 in /usr/local/lib/python3.10/d
Requirement already satisfied: jsonschema-specifications>=2023.03.6 in /usr
Requirement already satisfied: referencing>=0.28.4 in /usr/local/lib/python
Requirement already satisfied: rpds-py>=0.7.1 in /usr/local/lib/python3.10/
Requirement already satisfied: jupyter-server<3,>=1.8 in /usr/local/lib/pyt
Requirement already satisfied: cffi>=1.0.1 in /usr/local/lib/python3.10/dis

```

```

Requirement already satisfied: pycparser in /usr/local/lib/python3.10/dist-
Requirement already satisfied: anyio<4,>=3.1.0 in /usr/local/lib/python3.10
Requirement already satisfied: websocket-client in /usr/local/lib/python3.1
Requirement already satisfied: sniffio>=1.1 in /usr/local/lib/python3.10/di
Requirement already satisfied: exceptiongroup in /usr/local/lib/python3.10/
Downloading d2l-1.0.3-py3-none-any.whl (111 kB)
----- 111.7/111.7 kB 8.3 MB/s eta 0:0
Downloading jupyter-1.0.0-py2.py3-none-any.whl (2.7 kB)
Downloading matplotlib-3.7.2-cp310-cp310-manylinux_2_17_x86_64.manylinux201
----- 11.6/11.6 MB 55.0 MB/s eta 0:00
Downloading matplotlib-inline-0.1.6-py3-none-any.whl (9.4 kB)
Downloading numpy-1.23.5-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x8
----- 17.1/17.1 MB 55.0 MB/s eta 0:00
Downloading pandas-2.0.3-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x8
----- 12.3/12.3 MB 56.1 MB/s eta 0:00
Downloading requests-2.31.0-py3-none-any.whl (62 kB)
----- 62.6/62.6 kB 4.4 MB/s eta 0:00:
Downloading scipy-1.10.1-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x8
----- 34.4/34.4 MB 19.7 MB/s eta 0:00
Downloading pyparsing-3.0.9-py3-none-any.whl (98 kB)
----- 98.3/98.3 kB 7.0 MB/s eta 0:00:
Downloading qtconsole-5.6.0-py3-none-any.whl (124 kB)
----- 124.7/124.7 kB 8.1 MB/s eta 0:0
Downloading QtPy-2.4.1-py3-none-any.whl (93 kB)
----- 93.5/93.5 kB 6.5 MB/s eta 0:00:
Using cached jedi-0.19.1-py2.py3-none-any.whl (1.6 MB)
Installing collected packages: requests, qtpy, pyparsing, numpy, matplotlib
  Attempting uninstall: requests
    Found existing installation: requests 2.32.3
    Uninstalling requests-2.32.3:
      Successfully uninstalled requests-2.32.3
  Attempting uninstall: pyparsing
    Found existing installation: pyparsing 3.1.4
    Uninstalling pyparsing-3.1.4:
      Successfully uninstalled pyparsing-3.1.4
  Attempting uninstall: numpy
    Found existing installation: numpy 1.26.4
    Uninstalling numpy-1.26.4:
      Successfully uninstalled numpy-1.26.4
  Attempting uninstall: matplotlib-inline
    Found existing installation: matplotlib-inline 0.1.7
    Uninstalling matplotlib-inline-0.1.7:
      Successfully uninstalled matplotlib-inline-0.1.7
  Attempting uninstall: scipy
    Found existing installation: scipy 1.13.1
    Uninstalling scipy-1.13.1:
      Successfully uninstalled scipy-1.13.1
  Attempting uninstall: pandas
    Found existing installation: pandas 2.1.4
    Uninstalling pandas-2.1.4:
      Successfully uninstalled pandas-2.1.4
  Attempting uninstall: matplotlib
    Found existing installation: matplotlib 3.7.1
    Uninstalling matplotlib-3.7.1:

```

```
Successfully uninstalled matplotlib-3.7.1
```

```
ERROR: pip's dependency resolver does not currently take into account all t
albucore 0.0.16 requires numpy>=1.24, but you have numpy 1.23.5 which is in
albumations 1.4.15 requires numpy>=1.24.4, but you have numpy 1.23.5 whi
bigframes 1.18.0 requires numpy>=1.24.0, but you have numpy 1.23.5 which is
chex 0.1.86 requires numpy>=1.24.1, but you have numpy 1.23.5 which is inco
google-colab 1.0.0 requires pandas==2.1.4, but you have pandas 2.0.3 which
google-colab 1.0.0 requires requests==2.32.3, but you have requests 2.31.0
jax 0.4.33 requires numpy>=1.24, but you have numpy 1.23.5 which is incompa
jaxlib 0.4.33 requires numpy>=1.24, but you have numpy 1.23.5 which is inco
mizani 0.11.4 requires pandas>=2.1.0, but you have pandas 2.0.3 which is in
pandas-stubs 2.1.4.231227 requires numpy>=1.26.0; python_version < "3.13",
plotnine 0.13.6 requires pandas<3.0.0,>=2.1.0, but you have pandas 2.0.3 wh
xarray 2024.9.0 requires numpy>=1.24, but you have numpy 1.23.5 which is in
xarray 2024.9.0 requires pandas>=2.1, but you have pandas 2.0.3 which is in
Successfully installed d2l-1.0.3 jedi-0.19.1 jupyter-1.0.0 matplotlib-3.7.2
```

## ✓ 2.1 Data Manipulation

1. **Data Handling Requirements:** To accomplish tasks, we need methods to both store and manipulate data.
2. **Two Essential Tasks with Data:**
  - (i) Acquire data.
  - (ii) Process data once it is stored inside the computer.
3. **Data Storage:** Acquiring data is pointless without storage mechanisms.
4. **Introduction to Tensors:** Multi-dimensional arrays, called tensors, are fundamental for data storage and manipulation.
5. **Relation to NumPy:** If familiar with the NumPy package, understanding tensors will be easy, as tensors are similar to NumPy arrays.
6. **Tensor Class in Deep Learning:**
  - Tensor classes (e.g., `ndarray` in MXNet, `Tensor` in PyTorch, and `TensorFlow`) are comparable to NumPy's `ndarray` but with additional features.
7. **Key Tensor Features:**
  - **Automatic Differentiation:** Tensors support automatic differentiation, which is crucial for deep learning.
  - **GPU Acceleration:** Tensors use GPUs for faster numerical computation, unlike NumPy, which operates on CPUs.
8. **Impact on Neural Networks:** These tensor features make neural networks easier to code and faster to run.

### ➤ 2.1.1 Getting Started

[ ] ↪ 10 cells hidden

### ✓ 2.1.2. Indexing and Slicing

- Tensor elements can be accessed via **indexing**, starting from 0.
- **Negative indexing** can be used to access elements relative to the end of the tensor.
- **Slicing** allows access to ranges of indices (e.g., `X[start:stop]` ), including the start index but excluding the stop index.
- For higher-order tensors, a single index or slice applies to **axis 0**.
- Example: `[-1]` selects the last row, and `[1:3]` selects the second and third rows.

`X[-1], X[1:3]`

```
⇒ (tensor([ 8.,  9., 10., 11.]),
    tensor([[ 4.,  5.,  6.,  7.],
            [ 8.,  9., 10., 11.])))
```

`X[1, 2] = 17`

`X`

```
⇒ tensor([[ 0.,  1.,  2.,  3.],
          [ 4.,  5., 17.,  7.],
          [ 8.,  9., 10., 11.]])
```

`X[:2, :] = 12`

`X`

```
⇒ tensor([[12., 12., 12., 12.],
          [12., 12., 12., 12.],
          [ 8.,  9., 10., 11.]])
```

### ✓ 2.1.3. Operations

Double-click (or enter) to edit

`torch.exp(x)`

```
⇒ tensor([162754.7969, 162754.7969, 162754.7969, 162754.7969, 162754.7969,
          162754.7969, 162754.7969, 162754.7969, 2980.9580, 8103.0840,
          22026.4648, 59874.1406])
```

- Likewise, we denote *binary* scalar operators, which map pairs of real numbers to a (single) real number via the signature  $f : \mathbb{R}, \mathbb{R} \rightarrow \mathbb{R}$ .
- Here, we produced the vector-valued by lifting the scalar function to an elementwise vector operation

```
x = torch.tensor([1.0, 2, 4, 8])
y = torch.tensor([2, 2, 2, 2])
x + y, x - y, x * y, x / y, x ** y

⇒ (tensor([ 3.,  4.,  6., 10.]),
    tensor([-1.,  0.,  2.,  6.]),
    tensor([ 2.,  4.,  8., 16.]),
    tensor([0.5000, 1.0000, 2.0000, 4.0000]),
    tensor([ 1.,  4., 16., 64.]))
```

In addition to elementwise operations, we can

- perform linear algebra operations like dot products and matrix multiplications (discussed in Section 2.3)
- also concatenate tensors by stacking them along a specified axis, such as concatenating matrices along rows (axis 0) or columns (axis 1).

```
X = torch.arange(12, dtype=torch.float32).reshape((3,4))
Y = torch.tensor([[2.0, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
torch.cat((X, Y), dim=0), torch.cat((X, Y), dim=1)
```

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

```
X == Y
```

```
⇒ tensor([[False,  True, False,  True],
          [False, False, False, False],
          [False, False, False, False]])
```

```
X.sum()
```

```
↗ tensor(66.)
```

## ✓ 2.1.4. Broadcasting

Elementwise binary operations can be performed on tensors of different shapes using broadcasting. Broadcasting involves two steps:

- expanding one or both tensors by copying elements along axes with length 1 to match the shapes
- performing the elementwise operation on the transformed tensors.

```
a = torch.arange(3).reshape((3, 1))
b = torch.arange(2).reshape((1, 2))
a, b
```

```
↗ (tensor([[0],
          [1],
          [2]]),
   tensor([[0, 1]]))
```

```
a + b
```

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

## ✓ 2.1.5. Saving Memory

Running operations can allocate new memory for results. For instance, in  $Y = X + Y$ , the original memory for  $Y$  is dereferenced, and  $Y$  points to newly allocated memory. Using Python's `id()` function, we can see that after  $Y = Y + X$ , the memory address of  $Y$  changes because Python creates new memory for the result before updating  $Y$  to reference this new location.



```
before = id(Y)
Y = Y + X
id(Y) == before
```

↔ False

```
Z = torch.zeros_like(Y)
print('id(Z):', id(Z))
Z[:] = X + Y
print('id(Z):', id(Z))
```

↔ id(Z): 135006201976112  
id(Z): 135006201976112

```
before = id(X)
X += Y
id(X) == before
```

↔ True

## ✓ 2.1.6. Conversion to Other Python Objects

The torch tensor and NumPy array will share their underlying memory, and changing one through an in-place operation will also change the other.

```
A = X.numpy()
B = torch.from_numpy(A)
type(A), type(B)
```

↔ (numpy.ndarray, torch.Tensor)

```
a = torch.tensor([3.5])
a, a.item(), float(a), int(a)
```

↔ (tensor([3.5000]), 3.5, 3.5, 3)

## ✓ 2.1.7 Summary

The tensor class is the core interface for data storage and manipulation in deep learning libraries, offering features like construction routines; indexing and slicing; basic mathematics operations; broadcasting; memory-efficient assignment; and conversion to and from other Python objects.

## ✓ 2.2 Data Preprocessing

Until now, we've used synthetic data in ready-made tensors. In real-world deep learning, we must preprocess messy data in various formats. The pandas library helps with this, and while this section isn't a full tutorial, it provides a quick overview of common pandas routines.

### ✓ 2.2.1. Reading the Dataset

- CSV files are commonly used to store tabular data, where each line represents a record with fields separated by commas (e.g., "Albert Einstein, March 14 1879, Ulm, Federal polytechnic school, field of gravitational physics").
- To demonstrate loading CSV files with pandas, we create a sample file, `house_tiny.csv`, containing home data.
- Each row represents a home, and the columns represent attributes like the number of rooms (`NumRooms`), roof type (`RoofType`), and price (`Price`).

```
import os

os.makedirs(os.path.join '..', 'data'), exist_ok=True)
data_file = os.path.join '..', 'data', 'house_tiny.csv')
with open(data_file, 'w') as f:
    f.write(''NumRooms,RoofType,Price
NA,NA,127500
2,NA,106000
4,Slate,178100
NA,NA,140000'')
```

```
import pandas as pd
```

```
data = pd.read_csv(data_file)
print(data)
```

```
↗
```

	NumRooms	RoofType	Price
0	NaN	NaN	127500
1	2.0	NaN	106000
2	4.0	Slate	178100
3	NaN	NaN	140000

### ✓ 2.2.2. Data Preparation

- The first step in processing a dataset is separating input and target columns, either by name or index using pandas' `iloc`.
- Missing values in CSVs, often marked as NaN (e.g., in "3,,270000"), are common issues in data science. They can be handled by either imputation (replacing missing values with estimates) or deletion (removing rows or columns with missing data).
- For categorical fields, imputation can treat NaN as its own category.

```
inputs, targets = data.iloc[:, 0:2], data.iloc[:, 2]
inputs = pd.get_dummies(inputs, dummy_na=True)
print(inputs)
```

```
↗
```

	NumRooms	RoofType_Slate	RoofType_nan
0	NaN	False	True
1	2.0	False	True
2	4.0	True	False
3	NaN	False	True

```
inputs = inputs.fillna(inputs.mean())
print(inputs)
```

```
↗
```

	NumRooms	RoofType_Slate	RoofType_nan
0	3.0	False	True
1	2.0	False	True
2	4.0	True	False
3	3.0	False	True

### ✓ 2.2.3. Conversion to the Tensor Format

```
import torch

X = torch.tensor(inputs.to_numpy(dtype=float))
y = torch.tensor(targets.to_numpy(dtype=float))
X, y

↔ (tensor([[3., 0., 1.],
           [2., 0., 1.],
           [4., 1., 0.],
           [3., 0., 1.]], dtype=torch.float64),
    tensor([127500., 106000., 178100., 140000.], dtype=torch.float64))
```

## ✓ 2.2.4. Discussion

- You can partition data columns, impute missing values, and load pandas data into tensors.
- Advanced data processing skills will be covered in Section 5.7.
- Data can be complex, often spread across multiple files or extracted from relational databases.
- Different data types include text, images, audio, and point clouds, requiring specialized tools and algorithms.
- Data quality issues such as outliers and recording errors must be addressed before model training.
- Data visualization tools like seaborn, Bokeh, and matplotlib help inspect data and identify problems.

## ✓ 2.3 Linear Algebra

### ✓ 2.3.1. Scalars

```
x = torch.tensor(3.0)
y = torch.tensor(2.0)

x + y, x * y, x / y, x**y

↔ (tensor(5.), tensor(6.), tensor(1.5000), tensor(9.))
```

### ✓ 2.3.2. Vectors

Vectors can be thought of as fixed-length arrays of scalars, where each scalar is an element (also called an entry or component) of the vector. In real-world datasets, these elements have real-world significance. For example:

- In predicting loan default risk, a vector's components might represent an applicant's income, employment duration, and number of previous defaults.
- In studying heart attack risk, a vector might represent a patient, with components corresponding to vital signs, cholesterol levels, and daily exercise minutes.

```
x = torch.arange(3)
```

```
x
```

```
↔ tensor([0, 1, 2])
```

```
x[2]
```

```
↔ tensor(2)
```

```
len(x)
```

```
↔ 3
```

```
x.shape
```

```
↔ torch.Size([3])
```

### ✓ 2.3.3. Matrices

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}.$$

:eqlabel: eq\_matrix\_def

In code, we represent a matrix  $\mathbf{A} \in \mathbb{R}^{m \times n}$  by a 2<sup>nd</sup>-order tensor with shape  $(m, n)$ . **[We can convert any appropriately sized  $m \times n$  tensor into an  $m \times n$  matrix]** by passing the desired shape to `reshape`:

```
A = torch.arange(6).reshape(3, 2)
```

A

```
⇒ tensor([[0, 1],
          [2, 3],
          [4, 5]])
```

Sometimes we want to flip the axes. When we exchange a matrix's rows and columns, the result is called its transpose.

$$\mathbf{A}^T = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}.$$

In code, we can access any (**matrix's transpose**) as follows:

A.T

```
⇒ tensor([[0, 2, 4],
          [1, 3, 5]])
```

```
A = torch.tensor([[1, 2, 3], [2, 0, 4], [3, 4, 5]])
```

```
A == A.T
```

```
⇒ tensor([[True, True, True],
          [True, True, True],
          [True, True, True]])
```

### ✓ 2.3.4. Tensors

We call software objects of the tensor class “tensors” precisely because they too can have arbitrary numbers of axes.

We denote general tensors by capital letters with a special font face (e.g.,  $\mathbf{X}$ ,  $\mathbf{Y}$ , and  $\mathbf{Z}$ ) and their indexing mechanism (e.g.,  $x_{ijk}$  and  $[\mathbf{X}]_{1,2i-1,3}$ ) follows naturally from that of matrices.

```
torch.arange(24).reshape(2, 3, 4)
```

```
⇒ tensor([[[ 0,  1,  2,  3],
           [ 4,  5,  6,  7],
           [ 8,  9, 10, 11]],
          [[12, 13, 14, 15],
           [16, 17, 18, 19],
           [20, 21, 22, 23]]])
```

### ✓ 2.3.5. Basic Properties of Tensor Arithmetic

```
A = torch.arange(6, dtype=torch.float32).reshape(2, 3)
B = A.clone() # Assign a copy of A to B by allocating new memory
A, A + B
```

```
⇒ (tensor([[0., 1., 2.],
           [3., 4., 5.]]),
   tensor([[ 0.,  2.,  4.],
           [ 6.,  8., 10.])))
```

The **[elementwise product of two matrices is called their *Hadamard product*]** (denoted  $\odot$ ).

We can spell out the entries of the Hadamard product of two matrices  $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m \times n}$ :

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \dots & a_{1n}b_{1n} \\ a_{21}b_{21} & a_{22}b_{22} & \dots & a_{2n}b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{m1} & a_{m2}b_{m2} & \dots & a_{mn}b_{mn} \end{bmatrix}.$$

### ✓ 2.3.6. Reduction

Often, we wish to calculate **[the sum of a tensor's elements.]** To express the sum of the elements in a vector  $\mathbf{x}$  of length  $n$ , we write  $\sum_{i=1}^n x_i$ . There is a simple function for it:

```
x = torch.arange(3, dtype=torch.float32)
x, x.sum()
```

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

To express **[sums over the elements of tensors of arbitrary shape]**, we simply sum over all its axes. For example, the sum of the elements of an  $m \times n$  matrix  $\mathbf{A}$  could be written

$$\sum_{i=1}^m \sum_{j=1}^n a_{ij}.$$

```
A.shape, A.sum()
```

```
↔ (torch.Size([2, 3]), tensor(15.))
```

```
A.shape, A.sum(axis=0).shape
```

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

```
A.shape, A.sum(axis=1).shape
```

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

```
A.sum(axis=[0, 1]) == A.sum() # Same as A.sum()
```

```
↔ tensor(True)
```

```
A.mean(), A.sum() / A.numel()
```

```
↔ (tensor(2.5000), tensor(2.5000))
```

```
A.mean(axis=0), A.sum(axis=0) / A.shape[0]
```

```
↔ (tensor([1.5000, 2.5000, 3.5000]), tensor([1.5000, 2.5000, 3.5000]))
```

### ✓ 2.3.7. Non-Reduction Sum



```
sum_A = A.sum(axis=1, keepdims=True)
sum_A, sum_A.shape
```

```
↔ (tensor([[ 3.],
           [12.]]),
    torch.Size([2, 1]))
```

```
A / sum_A
```

```
↔ tensor([[0.0000, 0.3333, 0.6667],
          [0.2500, 0.3333, 0.4167]])
```

```
A.cumsum(axis=0)
```

```
↔ tensor([[0., 1., 2.],
          [3., 5., 7.]])
```

### ✓ 2.3.8. Dot Products

- One of the most fundamental operations is the dot product.
- Given two vectors  $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$ , their *dot product*  $\mathbf{x}^\top \mathbf{y}$  (also known as *inner product*,  $\langle \mathbf{x}, \mathbf{y} \rangle$ ) is a sum over the products of the elements at the same position:

$$\mathbf{x}^\top \mathbf{y} = \sum_{i=1}^d x_i y_i.$$

```
y = torch.ones(3, dtype = torch.float32)
x, y, torch.dot(x, y)
```

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

```
torch.sum(x * y)
```

```
↔ tensor(3.)
```

### ✓ 2.3.9. Matrix–Vector Products

To start off, we visualize our matrix in terms of its row vectors

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix},$$


where each  $\mathbf{a}_i^\top \in \mathbb{R}^n$  is a row vector representing the  $i^{\text{th}}$  row of the matrix  $\mathbf{A}$ .

**[The matrix--vector product  $\mathbf{Ax}$  is simply a column vector of length  $m$ , whose  $i^{\text{th}}$  element is the dot product  $\mathbf{a}_i^\top \mathbf{x}$ :]**

$$\mathbf{Ax} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{x} \\ \mathbf{a}_2^\top \mathbf{x} \\ \vdots \\ \mathbf{a}_m^\top \mathbf{x} \end{bmatrix}.$$

We can think of multiplication with a matrix  $\mathbf{A} \in \mathbb{R}^{m \times n}$  as a transformation that projects vectors from  $\mathbb{R}^n$  to  $\mathbb{R}^m$ . These transformations are remarkably useful.

`A.shape, x.shape, torch.mv(A, x), A@x`

 `(torch.Size([2, 3]), torch.Size([3]), tensor([ 5., 14.]), tensor([ 5., 14.]))`

## ✓ 2.3.10. Matrix–Matrix Multiplication

Say that we have two matrices  $\mathbf{A} \in \mathbb{R}^{n \times k}$  and  $\mathbf{B} \in \mathbb{R}^{k \times m}$ :

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nk} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k1} & b_{k2} & \cdots & b_{km} \end{bmatrix}.$$

Let  $\mathbf{a}_i^\top \in \mathbb{R}^k$  denote the row vector representing the  $i^{\text{th}}$  row of the matrix  $\mathbf{A}$  and let  $\mathbf{b}_j \in \mathbb{R}^k$  denote the column vector from the  $j^{\text{th}}$  column of the matrix  $\mathbf{B}$ :

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_n^\top \end{bmatrix}, \quad \mathbf{B} = [\mathbf{b}_1 \quad \mathbf{b}_2 \quad \cdots \quad \mathbf{b}_m].$$

To form the matrix product  $\mathbf{C} \in \mathbb{R}^{n \times m}$ , we simply compute each element  $c_{ij}$  as the dot product between the  $i^{\text{th}}$  row of  $\mathbf{A}$  and the  $j^{\text{th}}$  column of  $\mathbf{B}$ , i.e.,  $\mathbf{a}_i^\top \mathbf{b}_j$ :

$$\mathbf{C} = \mathbf{AB} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_n^\top \end{bmatrix} [\mathbf{b}_1 \quad \mathbf{b}_2 \quad \cdots \quad \mathbf{b}_m] = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{b}_1 & \mathbf{a}_1^\top \mathbf{b}_2 & \cdots & \mathbf{a}_1^\top \mathbf{b}_m \\ \mathbf{a}_2^\top \mathbf{b}_1 & \mathbf{a}_2^\top \mathbf{b}_2 & \cdots & \mathbf{a}_2^\top \mathbf{b}_m \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{a}_n^\top \mathbf{b}_1 & \mathbf{a}_n^\top \mathbf{b}_2 & \cdots & \mathbf{a}_n^\top \mathbf{b}_m \end{bmatrix}.$$

```
B = torch.ones(3, 4)
torch.mm(A, B), A@B
```

```
↔ (tensor([[ 3.,  3.,  3.,  3.],
           [12., 12., 12., 12.]]),
   tensor([[ 3.,  3.,  3.,  3.],
           [12., 12., 12., 12.])))
```

### ✓ 2.3.11. Norms

Some of the most useful operators in linear algebra are norms. Informally, the norm of a vector tells us how big it is.

A norm is a function  $\|\cdot\|$  that maps a vector to a scalar and satisfies the following three properties:

1. Given any vector  $\mathbf{x}$ , if we scale (all elements of) the vector by a scalar  $\alpha \in \mathbb{R}$ , its norm scales accordingly:

$$\|\alpha\mathbf{x}\| = |\alpha|\|\mathbf{x}\|.$$

2. For any vectors  $\mathbf{x}$  and  $\mathbf{y}$ : norms satisfy the triangle inequality:

$$\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|.$$

3. The norm of a vector is nonnegative and it only vanishes if the vector is zero:

$$\|\mathbf{x}\| \geq 0 \text{ for all } \mathbf{x} \text{ and } \|\mathbf{x}\| = 0 \text{ if and only if } \mathbf{x} = \mathbf{0}.$$

The Euclidean norm that we all learned in elementary school geometry when calculating the hypotenuse of a right triangle is the square root of the sum of squares of a vector's elements. Formally, this is called **[the  $\ell_2$  norm]** and expressed as

(

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^n x_i^2}.$$

)

The method `norm` calculates the  $\ell_2$  norm.

```
u = torch.tensor([3.0, -4.0])
torch.norm(u)
```

```
↔ tensor(5.)
```

**[The  $\ell_1$  norm]** is also common and the associated measure is called the Manhattan distance. By definition, the  $\ell_1$  norm sums the absolute values of a vector's elements:

(

$$\|\mathbf{x}\|_1 = \sum_{i=1}^n |x_i|.$$

)

```
torch.abs(u).sum()
```

```
↔ tensor(7.)
```

```
torch.norm(torch.ones((4, 9)))
```

```
↔ tensor(6.)
```

## 2.3.12. Discussion

- Scalars, vectors, matrices, and tensors are the basic mathematical objects used in linear algebra and have zero, one, two, and an arbitrary number of axes, respectively.
- Tensors can be sliced or reduced along specified axes via indexing, or operations such as `sum` and `mean`, respectively.
- Elementwise products are called Hadamard products. By contrast, dot products, matrix–vector products, and matrix–matrix products are not elementwise operations and in general return objects having shapes that are different from the the operands.
- Compared to Hadamard products, matrix–matrix products take considerably longer to compute (cubic rather than quadratic time).
- Norms capture various notions of the magnitude of a vector (or matrix), and are commonly applied to the difference of two vectors to measure their distance apart.
- Common vector norms include the  $\ell_1$  and  $\ell_2$  norms, and common matrix norms include the *spectral* and *Frobenius* norms.

## ✓ 2.5. Automatic Differentiation

```
import torch
```

### ✓ 2.5.1. A Simple Function

```
x = torch.arange(4.0)
```

```
x
```

```
↔ tensor([0., 1., 2., 3.])
```

Avoid allocating new memory every time we take a derivative because deep learning requires successively computing derivatives with respect to the same parameters a great many times, and we might risk running out of memory.

```
# Can also create x = torch.arange(4.0, requires_grad=True)
x.requires_grad_(True)
x.grad # The gradient is None by default
```

```
y = 2 * torch.dot(x, x)
y
```

```
↩ tensor(28., grad_fn=<MulBackward0>)
```

```
y.backward()
x.grad
```

```
↩ tensor([ 0.,  4.,  8., 12.])
```

```
x.grad == 4 * x
```

```
↩ tensor([True, True, True, True])
```

- PyTorch does not automatically reset the gradient buffer when we record a new gradient.
- Instead, the new gradient is added to the already-stored gradient.
- To reset the gradient buffer, we can call `x.grad.zero_()` as follows:

```
x.grad.zero_() # Reset the gradient
y = x.sum()
y.backward()
x.grad
```

```
↩ tensor([1., 1., 1., 1.])
```

## ✓ 2.5.2. Backward for Non-Scalar Variables

- When  $(y)$  is a vector, its derivative with respect to another vector  $(x)$  is represented by a matrix called the **Jacobian**, containing the partial derivatives of each component of  $(y)$  with respect to each component of  $(x)$ .
- However, in machine learning, we often sum the gradients of each component of  $(y)$  with respect to the full vector  $(x)$ , resulting in a gradient vector with the same shape as  $(x)$ .

```
x.grad.zero_()
y = x * x
y.backward(gradient=torch.ones(len(y))) # Faster: y.sum().backward()
x.grad
```

```
↔ tensor([0., 2., 4., 6.])
```

Because deep learning frameworks vary in how they interpret gradients of non-scalar tensors, PyTorch takes some steps to avoid confusion. Invoking backward on a non-scalar elicits an error unless we tell PyTorch how to reduce the object to a scalar.

### ✓ 2.5.3. Detaching Computation

Sometimes, we want to exclude certain calculations from the computational graph to prevent gradient computation for intermediate terms. For example, if  $(z = x * y)$  and  $(y = x * x)$ , but we only want to compute the direct influence of  $(x)$  on  $(z)$  (ignoring the influence through  $(y)$ ), we can detach  $(y)$  from the graph. By creating a new variable  $(u)$  with the same value as  $(y)$ , but without its computational history, gradients won't flow through  $(u)$ . Thus, the gradient of  $(z = x * u)$  will be  $(u)$ , not  $(3 * x^2)$ .

```
x.grad.zero_()
y = x * x
u = y.detach()
z = u * x

z.sum().backward()
x.grad == u
```

```
↔ tensor([True, True, True, True])
```

```
x.grad.zero_()
y.sum().backward()
x.grad == 2 * x

⇒ tensor([True, True, True, True])
```

## ✓ 2.5.4. Gradients and Python Control Flow

One benefit of using automatic differentiation is that even if building the computational graph of a function required passing through a maze of Python control flow (e.g., conditionals, loops, and arbitrary function calls), we can still calculate the gradient of the resulting variable.

```
def f(a):
    b = a * 2
    while b.norm() < 1000:
        b = b * 2
    if b.sum() > 0:
        c = b
    else:
        c = 100 * b
    return c

a = torch.randn(size=(), requires_grad=True)
d = f(a)
d.backward()

a.grad == d / a

⇒ tensor(True)
```

## ✓ 2.5.5. Discussion



- Automatic differentiation boosts productivity in deep learning.
- Autograd simplifies gradient calculations for large models.
- Optimizing autograd libraries is crucial for efficiency.
- Tools like compilers and graph manipulation improve speed and memory use.
- Key steps:
  1. Attach gradients to variables.
  2. Record target value computation.
  3. Perform backpropagation.
  4. Access the gradient.

## ✓ 3.1 Linear Regression

- Regression problems are used to predict numerical values.
- Common regression applications include predicting prices (e.g., homes, stocks), length of stay in hospitals, and retail sales demand.
- Not all prediction problems are regression problems; some involve classification for category prediction.
- Example: predicting house prices based on area (square feet) and age (years).
- To build a model for house price prediction, a dataset with sales price, area, and age is needed.
- In machine learning:
  - The dataset is called the training set.
  - Each row in the dataset is an example, instance, or data point.
  - The value to be predicted (e.g., price) is the label or target.
  - Variables like age and area used for prediction are called features or covariates.

```
%matplotlib inline
import math
import time
import numpy as np
import torch
from d2l import torch as d2l
```

### ✓ 3.1.1. Basics

- **Linear Regression** is one of the simplest and most popular methods for solving regression problems, dating back to the 19th century.
- Assumes a linear relationship between features  $\mathbf{x}$  and target  $y$ , where the expected value of the target can be expressed as a weighted sum of the features plus some noise.

- The **model** expresses the expected target value as:

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

- $\mathbf{w}$ : vector of weights.
- $b$ : bias or intercept term.
- This model allows for prediction by finding the best weights and bias that minimize the error between predictions and actual values.
- **Loss Function:** The most common loss function for regression problems is the squared error, which penalizes large prediction errors.

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

- Minimizing this loss gives us the best-fitting parameters for our linear model.
- **Analytic Solution:** Linear regression has an exact solution if the matrix  $\mathbf{X}^\top \mathbf{X}$  (*designmatrix*) is invertible.  $\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$
- **Gradient Descent:** When an analytic solution is not feasible, we can use **Minibatch Stochastic Gradient Descent (SGD)** to iteratively update the model parameters to reduce error.

- SGD updates parameters using small random batches of data rather than the entire dataset.
- Update rule:

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

- $\eta$ : learning rate.
- **Prediction:** After training the model, we use the learned weights and bias to make predictions on new examples.
- **Generalization:** The goal is not just to minimize loss on the training set, but also to generalize well to unseen data.

### ✓ 3.1.2. Vectorization for Speed

Here are the times taken by the two methods for adding two 10,000-dimensional vectors:

- **Using Python for-loop:** ~0.0413 seconds
- **Using vectorized operation:** ~0.0002 seconds

As seen, the vectorized operation is significantly faster than using a Python for-loop. This highlights the importance of vectorization in processing large datasets efficiently.

```
n = 10000
a = torch.ones(n)
b = torch.ones(n)

c = torch.zeros(n)
t = time.time()
for i in range(n):
    c[i] = a[i] + b[i]
f'{time.time() - t:.5f} sec'
```

 '0.39568 sec'

The second method is dramatically faster than the first. Vectorizing code often yields order-of-magnitude speedups.

### ✓ 3.1.3. The Normal Distribution and Squared Loss

### 1. Squared Loss Motivation:

- The squared loss objective minimizes the distance between predicted and actual values, effectively yielding the conditional expectation  $E[Y | X]$  in linear relationships and penalizing outliers significantly.

### 2. Historical Background:

- Linear regression was developed in the early 19th century, with Gauss and Legendre as key figures; Gauss also discovered the normal distribution.

### 3. Mathematical Definition:

- A normal distribution with mean  $\mu$  and variance  $\sigma^2$  (standard deviation  $\sigma$ ) is given as

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

### 4. Connection to Regression:

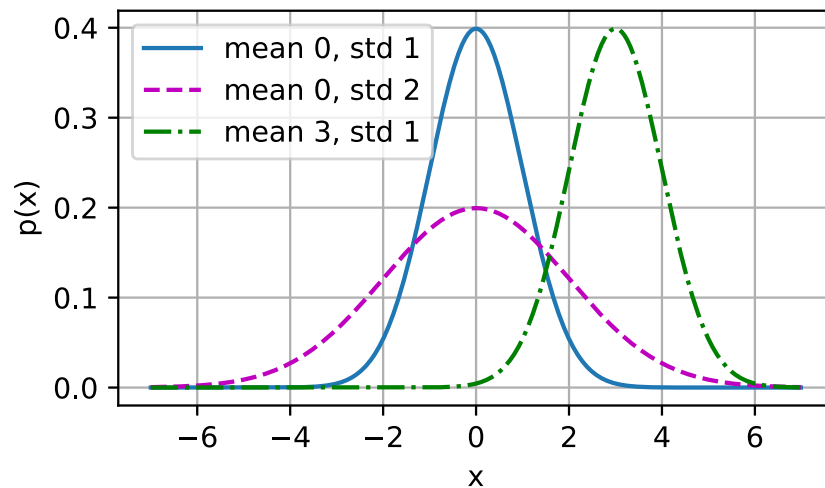
- The normal distribution's properties justify the use of squared loss in linear regression, providing a probabilistic foundation for the model.

These points summarize the essential connections between normal distribution and squared loss in a more compact format.

```
def normal(x, mu, sigma):
    p = 1 / math.sqrt(2 * math.pi * sigma**2)
    return p * np.exp(-0.5 * (x - mu)**2 / sigma**2)
```

```
# Use NumPy again for visualization
x = np.arange(-7, 7, 0.01)

# Mean and standard deviation pairs
params = [(0, 1), (0, 2), (3, 1)]
d2l.plot(x, [normal(x, mu, sigma) for mu, sigma in params], xlabel='x',
          ylabel='p(x)', figsize=(4.5, 2.5),
          legend=[f'mean {mu}, std {sigma}' for mu, sigma in params])
```



## 1. Impact of Mean and Variance:

- Changing the mean shifts the distribution along the x-axis, while increasing the variance spreads it out, lowering its peak.

## 2. Modeling Noisy Measurements:

- In linear regression, observations can be modeled as:

$$y = \mathbf{w}^\top \mathbf{x} + b + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, \sigma^2).$$

## 3. Likelihood Function:

- The likelihood of observing  $y$  given

$\mathbf{x}$

is expressed as:

$$P(y | \mathbf{x}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y - \mathbf{w}^\top \mathbf{x} - b)^2\right).$$

## 4. Maximum Likelihood Estimation (MLE):

- The best parameters  $\mathbf{w}$  and  $b$  maximize the likelihood of the dataset:

$$P(\mathbf{y} | \mathbf{X}) = \prod_{i=1}^n p(y^{(i)} | \mathbf{x}^{(i)}).$$

## 5. Negative Log-Likelihood:

- Minimizing the negative log-likelihood leads to:

$$-\log P(\mathbf{y} | \mathbf{X}) = \sum_{i=1}^n \left( \frac{1}{2\sigma^2} (y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)} - b)^2 + \text{constant} \right).$$

- Ignoring the constant term simplifies this to the squared error loss.

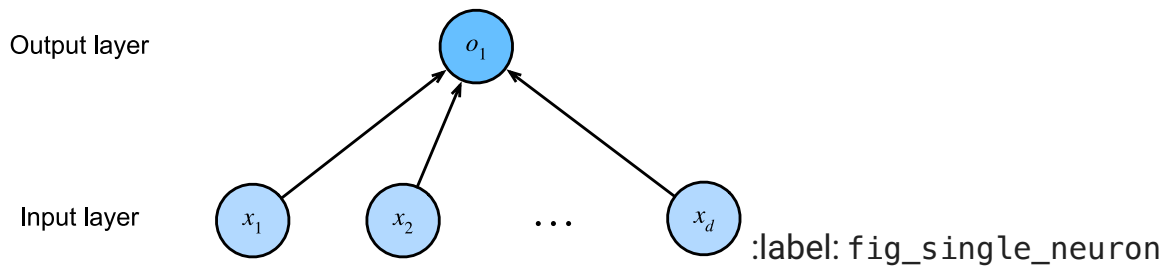
## 6. Conclusion:

- Minimizing mean squared error is equivalent to MLE of a linear model assuming additive Gaussian noise, with the solution independent of the variance

$\sigma$

These points highlight the essential connections between normal distribution, noise in measurements, and the principles of linear regression.

## ✓ 3.1.4. Linear Regression as a Neural Network



### 1. Neural Network Representation:

- Linear regression can be viewed as a single-layer neural network where each input feature is represented by an input neuron, all connected directly to the output neuron.

### 2. Diagram Representation:

- The connectivity pattern of the network is illustrated in a diagram (see Figure :numref:fig\_single\_neuron), emphasizing how inputs are connected to the output, without detailing the specific weights and biases.

### 3. Input and Output Structure:

- The inputs are denoted as  $x_1, \dots, x_d$ , where  $d$  represents the number of input features (feature dimensionality).
- The network has a single output neuron  $o_1$ , as it predicts only one numerical value.

### 4. Summary of Linear Regression:

- Linear regression is effectively a single-layer fully connected neural network, laying the foundation for understanding more complex networks that will be explored in later chapters.

These points encapsulate the concept of linear regression as a neural network in a clear and concise manner.

#### ✓ 3.1.4.1. Biology

## 1. Historical Context:

- Linear regression predates computational neuroscience but serves as a foundational concept for models of artificial neurons developed by cyberneticists like Warren McCulloch and Walter Pitts.

## 2. Biological Neuron Structure:

- A biological neuron consists of:
  - **Dendrites:** Input terminals receiving information.
  - **Nucleus:** The processing unit (CPU) of the neuron.
  - **Axon:** The output wire transmitting information.
  - **Axon Terminals:** Output terminals connecting to other neurons via synapses.

## 3. Information Processing:

- Inputs  $x_i$  from other neurons are weighted by **synaptic weights**  $w_i$ , influencing their activation or inhibition through the product  $x_i w_i$ .
- The weighted inputs are aggregated in the nucleus as:

$$y = \sum_i x_i w_i + b,$$

possibly followed by nonlinear processing

$$\sigma(y)$$

- The processed information is then transmitted through the axon to its destination, which could be another neuron or an actuator.

## 4. Complex Behavior from Simple Units:

- The concept of combining multiple neurons to create complex behaviors reflects the study of biological neural systems, emphasizing the importance of connectivity and learning algorithms.

## 5. Broader Inspirations in Deep Learning:

- While biological systems inspired early neural network models, modern deep learning draws from a diverse range of fields, including mathematics, linguistics, psychology, statistics, and computer science, rather than solely from biology.

These points encapsulate the relationship between biological neurons and artificial neural networks, along with the broader influences on deep learning today.



## ✓ 3.1.5. Summary

### 1. Introduction to Linear Regression:

- Traditional linear regression focuses on minimizing squared loss to determine the parameters of a linear function based on the training set.

### 2. Objective Motivation:

- The choice of squared loss is justified through practical considerations and an interpretation of linear regression as maximum likelihood estimation, assuming linearity and Gaussian noise.

### 3. Connections to Statistics:

- The discussion included computational aspects and statistical interpretations, highlighting the relevance of linear models.

### 4. Linear Models as Neural Networks:

- Linear regression can be represented as simple neural networks, with inputs directly connected to outputs.

### 5. Foundational Components for Future Models:

- While the focus will shift beyond linear models, they serve as a foundation for essential components like parametric forms, differentiable objectives, optimization via minibatch stochastic gradient descent, and evaluation on unseen data.

This summary captures the main ideas of the section regarding linear regression and its foundational role in the context of neural networks and broader modeling concepts.

## ✓ 3.2. Object-Oriented Design for Implementation

- Discussed components: data, model, loss function, optimization algorithm.
- Linear regression is a fundamental machine learning model.
- Training involves similar components as other models.
- **API Design for Implementation:**
  - Importance of designing APIs before implementation.
  - Components in deep learning treated as objects.
  - Proposal to define classes for objects and their interactions.
  - Object-oriented design streamlines presentation and is beneficial for projects.
- **Inspired by Open-Source Libraries (e.g., PyTorch Lightning):**
  - High-level class structure:
    - **Module:** Contains models, loss functions, optimization methods.
    - **DataModule:** Provides data loaders for training and validation.
    - **Trainer:** Combines Module and DataModule; facilitates training on various hardware platforms.
  - Most code in the book will adapt the Module and DataModule classes.
  - Trainer class discussion will focus on GPUs, CPUs, parallel training, and optimization algorithms.

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

### ✓ 3.2.1. Utilities

- Need for utilities to enhance readability and usability in Jupyter notebooks.
- Challenge: Long class definitions conflict with the notebook's need for short, explainable code fragments.
- **Utility Function for Method Registration:**
  - Allows registration of functions as methods in a class after the class is created.
  - Methods can be registered even after instances of the class are created.
  - Facilitates splitting class implementation into multiple, manageable code blocks.

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

"@save" is not an allowed annotation -  
allowed values include [@param, @title,  
@markdown].

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

```
a = A()
```

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

```
a.do()
```

```
⇒ Class attribute "b" is 1
```

```
class HyperParameters: #@save
    """The base class of hyperparameter
    def save_hyperparameters(self, ignore):
        raise NotImplementedError
```

"@save" is not an allowed annotation -  
allowed values include [@param, @title,  
@markdown].

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

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

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

```

class ProgressBoard(d2l.HyperParameters):
    """The board that plots data points"""
    def __init__(self, xlabel=None, ylabel=None, ylim=None, xscale='linear',
                 ls=['-', '--', '-.', ''], fig=None, axes=None, title=None):
        self.save_hyperparameters()

    def draw(self, x, y, label, every_n):
        raise NotImplementedError

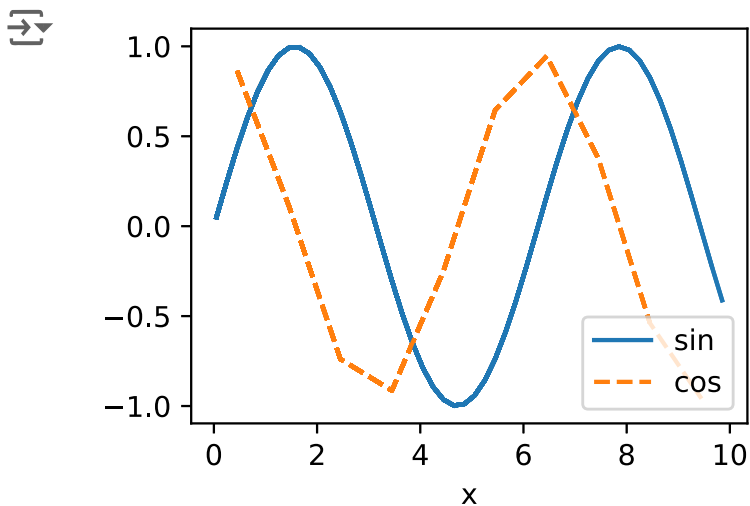
```

"@save" is not an allowed annotation -  
allowed values include [@param, @title,  
@markdown].

```

board = d2l.ProgressBoard('x')
for x in np.arange(0, 10, 0.1):
    board.draw(x, np.sin(x), 'sin', every_n=2)
    board.draw(x, np.cos(x), 'cos', every_n=10)

```



### ✓ 3.2.2. Models

- **Module Class Overview:**

- Base class for all models to be implemented.
- Requires at least three methods:
  1. **`__init__`:**
    - Stores the learnable parameters.
  2. **`training_step`:**
    - Accepts a data batch and returns the loss value.
  3. **`configure_optimizers`:**
    - Returns the optimization method(s) for updating learnable parameters.

- **Optional Method:**

- **`validation_step`:**
  - Reports evaluation measures.

- **Reusability Consideration:**

- **forward Method:**
  - Sometimes used to separate output computation, enhancing code reusability.

```

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

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

    def forward(self, X):
        assert hasattr(self, 'net'), 'Module does not have attribute net'
        return self.net(X)

    def plot(self, key, value, train):
        """Plot a point in animation."""
        assert hasattr(self, 'trainer')
        self.board.xlabel = 'epoch'
        if train:
            x = self.trainer.train_batch_idx
            self.trainer.num_train_batches = self.trainer.num_train_batches + 1
            n = self.trainer.num_train_batches
            self.plot_train_per_epoch = plot_train_per_epoch
        else:
            x = self.trainer.epoch + 1
            n = self.trainer.num_validation_batches
            self.plot_valid_per_epoch = plot_valid_per_epoch
        self.board.draw(x, value.to(d2l.device), key,
                        ('train_' if train else 'valid_'),
                        every_n=int(n))

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

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

    def configure_optimizers(self):
        raise NotImplementedError

```

"@save" is not an allowed annotation - allowed values include [@param, @title, @markdown].



### ✓ 3.2.3. Data

- **DataModule Class Overview:**

- Serves as the base class for handling data.

- **Key Methods:**

- **`__init__`:**
  - Prepares the data, including downloading and preprocessing as necessary.
- **`train_dataloader`:**
  - Returns the data loader for the training dataset.
  - Functions as a Python generator, yielding a data batch for each use.
  - The yielded batch is used in the `training_step` method of the Module class to compute loss.
- **Optional Method:**
  - **`val_dataloader`:**
    - Returns the data loader for the validation dataset.
    - Functions similarly to `train_dataloader`, yielding batches for the `validation_step` method in the Module class.

```
class DataModule(d2l.HyperParameters):
    """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(trai

    def val_dataloader(self):
        return self.get_dataloader(trai
```

"@save" is not an allowed annotation -  
allowed values include [`@param`, `@title`,  
`@markdown`].



## ✓ 3.2.4. Training

```

class Trainer(d2l.HyperParameters): #@
    """The base class for training mode
    def __init__(self, max_epochs, num_
        self.save_hyperparameters()
        assert num_gpus == 0, 'No GPU s

    def prepare_data(self, data):
        self.train_dataloader = data.tr
        self.val_dataloader = data.val_
        self.num_train_batches = len(se
        self.num_val_batches = (len(sel
            if self

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

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

    def fit_epoch(self):
        raise NotImplementedError

```

"@save" is not an allowed annotation -  
allowed values include [@param, @title,  
@markdown].



### ✓ 3.2.5. Summary



- **Object-Oriented Design Emphasis:**

- Classes demonstrate how objects store data and interact with one another.
- Future implementations will enhance these classes, including the use of `@add_to_class`.

- **D2L Library:**

- Contains fully implemented classes.
- A lightweight toolkit that simplifies structured modeling for deep learning.
- Facilitates the reuse of components across projects with minimal changes.

- **Modularity Benefits:**

- Enables easy replacements of individual components, such as:
  - Optimizer
  - Model
  - Dataset
- Promotes conciseness and simplicity in code, beneficial for both the book's content and personal projects.

## ✓ 3.4. Linear Regression Implementation from Scratch

- **Linear Regression Implementation:**

- Will implement the method from scratch, including:

1. **Model**
2. **Loss function**
3. **Minibatch stochastic gradient descent optimizer**
4. **Training function** to integrate all components.

- **Synthetic Data Application:**

- Will run a synthetic data generator and apply the model to the dataset.

- **Importance of Implementation from Scratch:**

- Ensures a deeper understanding of how things work.
- Critical for customizing models, defining custom layers, or loss functions.

- **Focus on Tensors and Automatic Differentiation:**

- The implementation will use only tensors and automatic differentiation.
- A more concise version utilizing deep learning frameworks will be introduced later, but the structure will remain the same.

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

### ✓ 3.4.1. Defining the Model

- **Initializing Model Parameters:**

- Before using minibatch SGD, parameters need to be initialized.
- **Weights** are initialized by drawing random numbers from a normal distribution:
  - Mean = 0
  - Standard deviation = 0.01 (can be adjusted using the `sigma` argument).
- **Bias** is initialized to 0.

- **Object-Oriented Design:**

- The parameter initialization is implemented in the `__init__` method of a subclass of `d2l.Module` (introduced in Section 3.2.2).

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

"@save" is not an allowed annotation - allowed values include [`@param`, `@title`, `@markdown`].

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

"@save" is not an allowed annotation - allowed values include [`@param`, `@title`, `@markdown`].

## ✓ 3.4.2. Defining the Loss Function

- **Defining the Loss Function:**

- The squared loss function (as introduced in 3.1.5) will be used.

- **Shape Transformation:**

- The true value `y` is transformed to match the shape of the predicted value `y_hat`.

- **Loss Calculation:**

- The method returns the loss value with the same shape as `y_hat`.
- The final returned value is the **averaged loss** across all examples in the minibatch.

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

"@save" is not an allowed annotation -  
allowed values include [@param, @title,  
@markdown].

### ✓ 3.4.3. Defining the Optimization Algorithm

- **Minibatch SGD for Parameter Optimization:**

- Although linear regression has a closed-form solution, we focus on **minibatch stochastic gradient descent (SGD)** to prepare for training general neural networks.

- **SGD Process:**

- At each step, a **minibatch** is randomly drawn from the dataset.
- The **gradient of the loss** with respect to the parameters is estimated.
- Parameters are updated in the direction that reduces the loss.

- **Learning Rate (lr) and Batch Size:**

- The code updates the parameters using a learning rate (lr).
- Since the loss is averaged over the minibatch, the learning rate doesn't need adjustment for batch size.

- **Future Consideration:**

- Adjusting the learning rate for large minibatches will be discussed in later chapters for large-scale distributed learning.

```
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_()
```

"@save" is not an allowed annotation -  
allowed values include [@param, @title,  
@markdown].

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

"@save" is not an allowed annotation -  
allowed values include [@param, @title,  
@markdown].



### ✓ 3.4.4. Training

- **Training Loop Overview:**

- We now have all components in place (parameters, loss function, model, and optimizer).
- The training loop is a critical part of every deep learning model covered in this book.

- **Epochs and Iterations:**

- **Epoch:** Pass through the entire training dataset once.
- **Iteration:** In each iteration, grab a minibatch, compute the loss, and update the parameters.

- **Training Process:**

- For each minibatch:
  1. **Compute loss** through the model's `training_step` method.
  2. **Compute gradients** for each parameter.
  3. **Update parameters** using the optimization algorithm.

- **Loop Summary:**

1. Initialize parameters  $(\mathbf{w}, b)$ .
2. Repeat until done:
  - Compute gradient  $\mathbf{g}$  for minibatch loss.
  - Update parameters:  $(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \eta \mathbf{g}$ .

- **Validation Dataset:**

- The synthetic regression dataset doesn't include validation data.
- In real scenarios, validation data is passed once per epoch to measure model performance.

- **Object-Oriented Design:**

- `prepare_batch` and `fit_epoch` methods are part of the `d2l.Trainer` class, helping structure training within the object-oriented framework.

```

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

@d2l.add_to_class(d2l.Trainer)  #@save
def fit_epoch(self):
    self.model.train()
    for batch in self.train_dataloader:
        loss = self.model.training_step
        self.optim.zero_grad()
        with torch.no_grad():
            loss.backward()
            if self.gradient_clip_val > 0:
                self.clip_gradients(self.optim)
        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.val_batch_idx += 1

```

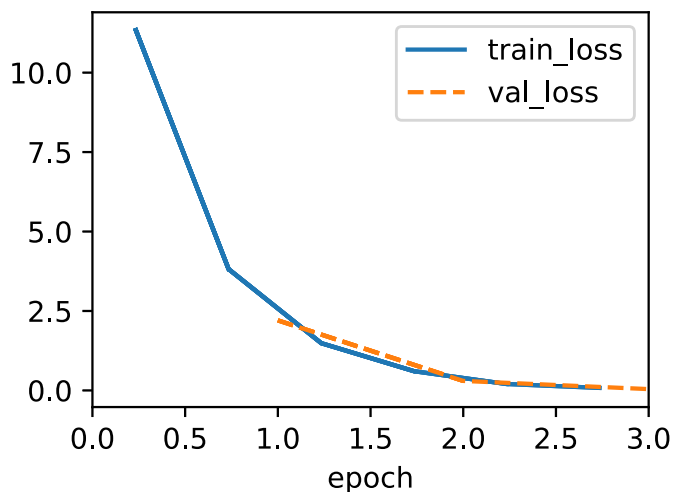
"@save" is not an allowed annotation - allowed values include [@param, @title, @markdown].

"@save" is not an allowed annotation - allowed values include [@param, @title, @markdown].

```

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)

```



```
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}')  
  
↔ error in estimating w: tensor([ 0.0726, -0.1553])  
   error in estimating b: tensor([0.2380])
```

### ✓ 3.4.5. Summary

- **Progress Toward Deep Learning Systems:**

- Implemented a fully functional neural network model and training loop.

- **Components Built:**

- Data loader
- Model
- Loss function
- Optimization procedure
- Visualization and monitoring tool

- **Object Composition:**

- A Python object was used to encapsulate all components necessary for training a model.

- **Practicality:**

- While not professional-grade, the current implementation is functional and can solve small problems efficiently.

- **Future Improvements:**

- Upcoming sections will focus on making the implementation:
  - **More concise** by reducing boilerplate code.
  - **More efficient** by leveraging the full potential of GPUs.

### ✓ 4.1. Softmax Regression



- **Introduction to Linear Regression**

- Covered in Section 3.1, with detailed implementations in Sections 3.4 (from scratch) and 3.5 (using high-level APIs).

- **Purpose of Regression**

- Used to answer "how much?" or "how many?" questions (e.g., predicting house prices, baseball wins, hospital stay durations).

- **Limitations of Regression**

- Prices cannot be negative; logarithmic transformation may be more effective.
- Nonnegative discrete variables (like hospital stays) may require specialized approaches (e.g., survival modeling).

- **Estimation Beyond Squared Errors**

- Emphasizes that estimation involves more than just minimizing errors.
- Supervised learning includes various methodologies, not just regression.

- **Shift to Classification Problems**

- Focuses on "which category?" questions (e.g., spam detection, customer sign-up likelihood, image classification).

- **Types of Classification**

- Hard assignments: definitive category assignments.
- Soft assignments: assessing probabilities for each category.

- **Multi-Label Classification**

- Addresses scenarios where items can belong to multiple categories (e.g., a news article covering various topics).
- References: Tsoumakas and Katakis (2007) for an overview, Huang et al. (2015) for image tagging algorithms.

## ✓ 4.1.1. Classification

- **Input Description**

- Each input is a  $2 \times 2$  grayscale image.
- Represented by four pixel values:  $x_1, x_2, x_3, x_4$ .

- **Categories**

- Each image belongs to one of three categories: "cat," "chicken," or "dog."

- **Label Representation Choices**

- **Integer Encoding:**

- Use  $y \in 1, 2, 3$  to represent categories:
  - 1 for "dog"
  - 2 for "cat"
  - 3 for "chicken"
- Effective for storage, but not ideal for unordered categories.

- **Ordinal Regression:**

- Applicable if categories have a natural order (e.g., age groups).
- Reference for ranking loss functions:
  - Moon et al. (2010).
  - Beutel et al. (2014) for Bayesian approaches with multi-modal responses.

- **Preferred Method: One-Hot Encoding**

- A categorical representation with a vector for each class.
- Vector size equals the number of categories.
- For our example:
  - "cat": (1, 0, 0)
  - "chicken": (0, 1, 0)
  - "dog": (0, 0, 1)
- Thus, labels are represented as:

$$y \in \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}.$$

## Linear Model

- **Model Requirements**

- A model with multiple outputs is needed—one for each class.
- For linear classification, as many affine functions as outputs are required.
- Typically, one fewer function is needed due to redundancy, but using a full set simplifies the approach.

- **Affine Functions for Outputs**

- For 4 features and 3 categories, we need:
  - **12 weights** (4 features  $\times$  3 categories)
  - **3 biases**

- Each output ( $o_i$ ) is calculated as follows: [

$$o_1 = x_1 w_{11} + x_2 w_{12} + x_3 w_{13} + x_4 w_{14} + b_1,$$

$$o_2 = x_1 w_{21} + x_2 w_{22} + x_3 w_{23} + x_4 w_{24} + b_2,$$

$$o_3 = x_1 w_{31} + x_2 w_{32} + x_3 w_{33} + x_4 w_{34} + b_3.$$

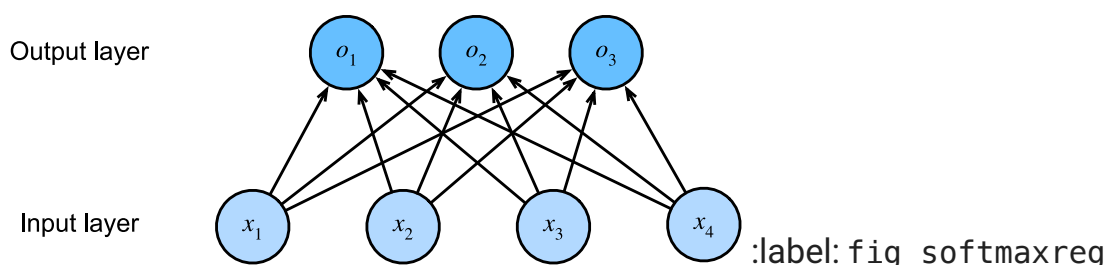
]

- **Network Structure**

- The model is a single-layer neural network, also known as a fully connected layer.
- Each output depends on all inputs.

- **Matrix Notation**

- For simplicity, the outputs can be expressed in vector and matrix form: [  $\mathbf{o} = \mathbf{W} \mathbf{x} + \mathbf{b}$  ]
- Here, ( $\mathbf{W}$ ) is a (3  $\times$  4) matrix of weights, and ( $\mathbf{b}$ ) is a bias vector in ( $\mathbb{R}^3$ ).



### 4.1.1.2. The Softmax

- **Direct Minimization Challenges**

- Minimizing the difference between outputs  $\mathbf{o}$  and labels  $\mathbf{y}$  has issues:
  - No guarantee that outputs  $o_i$  sum to 1 (as probabilities should).
  - No guarantee that outputs  $o_i$  are nonnegative or do not exceed 1.
- These problems make the estimation fragile, particularly with outliers.

### • Example Issue

- A positive dependency between features (e.g., bedrooms in a house) could lead to probabilities exceeding 1, especially for high-value items (e.g., mansions).

### • Solutions for Output Constraints

#### ◦ Probit Model:

- Assumes outputs are corrupted versions of labels by noise  $\epsilon$  from a normal distribution:

$$\mathbf{y} = \mathbf{o} + \boldsymbol{\epsilon}, \quad \epsilon_i \sim \mathcal{N}(0, \sigma^2)$$

- While it introduces nonlinearity, it complicates optimization compared to the softmax.

#### ◦ Softmax Function:

- Defines class probabilities as:

$$P(y = i) \propto \exp(o_i)$$

- Ensures:

- Nonnegativity of probabilities.
- Monotonic relationship with outputs  $o_i$ .

- Normalization to sum to 1:

$$\hat{y}_i = \frac{\exp(o_i)}{\sum_j \exp(o_j)}$$

- The output  $\hat{\mathbf{y}}$  represents the estimated probabilities.

### • Properties of Softmax

- The largest coordinate of

$\mathbf{o}$

corresponds to the most likely class in  $\hat{\mathbf{y}}$ .

- The argmax can be computed directly from  $o_j$  without explicitly calculating softmax:

$$\operatorname{argmax}_j \hat{y}_j = \operatorname{argmax}_j o_j$$

- **Historical Context**

- The concept of softmax traces back to Gibbs (1902) and Boltzmann, who modeled distributions over energy states in gas molecules.
- Boltzmann's formula:

$$\text{Prevalence} \propto \exp\left(-\frac{E}{kT}\right)$$

- Where  $E$  is energy,  $T$  is temperature, and  $k$  is the Boltzmann constant.
- In statistics, adjusting "temperature" influences the favorability of states (low vs. high energy).
- Energy-based models draw on this perspective to describe optimization in deep learning.

### 4.1.1.3. Vectorization

## Vectorization for Efficient Computation

- **Purpose of Vectorization**

- To enhance computational efficiency in processing minibatches of data.

- **Minibatch Setup**

- Given a minibatch  $\mathbf{X} \in \mathbb{R}^{n \times d}$ :
  - $n$ : number of examples
  - $d$ : dimensionality (number of inputs)
- Output categories:  $q$ 
  - Weights:  $\mathbf{W} \in \mathbb{R}^{d \times q}$
  - Biases:  $\mathbf{b} \in \mathbb{R}^{1 \times q}$

- **Matrix Operations**

- Compute outputs:

$$\mathbf{O} = \mathbf{XW} + \mathbf{b},$$

$$\hat{\mathbf{Y}} = \text{softmax}(\mathbf{O}).$$

- This reduces the dominant computation to a matrix-matrix product  $\mathbf{XW}$ .

- **Rowwise Softmax Calculation**

- Each row of  $\mathbf{X}$  corresponds to a data example.
- Perform softmax rowwise:
  - Exponentiate each entry in the row of  $\mathbf{O}$ .
  - Normalize by dividing by the sum of exponentials.

- **Numerical Stability Considerations**

- Be cautious of numerical overflow or underflow when exponentiating large numbers.
- Deep learning frameworks typically handle these issues automatically.

## ✓ 4.1.2. Loss Function

### Overview

- To optimize the mapping from features  $\mathbf{x}$  to probabilities  $\hat{\mathbf{y}}$ , we use **maximum likelihood estimation** (MLE).

## Log-Likelihood

- The softmax function provides  $\hat{\mathbf{y}}$  as the estimated conditional probabilities for each class, such as  $\hat{y}_1 = P(y = \text{cat} \mid \mathbf{x})$ .
- For a dataset  $\mathbf{X}$  with labels  $\mathbf{Y}$  in one-hot encoding:

$$P(\mathbf{Y} \mid \mathbf{X}) = \prod_{i=1}^n P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)})$$

- This factorization is valid under the assumption of independent labels.
- To facilitate optimization, we minimize the negative log-likelihood:

$$-\log P(\mathbf{Y} \mid \mathbf{X}) = \sum_{i=1}^n -\log P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}) = \sum_{i=1}^n l(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}),$$

where the loss function  $l$  is defined as:

$$l(\mathbf{y}, \hat{\mathbf{y}}) = - \sum_{j=1}^q y_j \log \hat{y}_j.$$

- This is known as **cross-entropy loss**. The loss is non-negative and equals zero only when predictions are made with certainty.

## Softmax and Cross-Entropy Loss

- To connect the softmax function with cross-entropy loss:

$$l(\mathbf{y}, \hat{\mathbf{y}}) = - \sum_{j=1}^q y_j \log \frac{\exp(o_j)}{\sum_{k=1}^q \exp(o_k)}.$$

Simplifying this, we get:

$$l(\mathbf{y}, \hat{\mathbf{y}}) = \log \sum_{k=1}^q \exp(o_k) - \sum_{j=1}^q y_j o_j.$$

- **Gradient Calculation:**

- The derivative with respect to any logit  $o_j$  is:

$$\partial_{o_j} l(\mathbf{y}, \hat{\mathbf{y}}) = \text{softmax}(\mathbf{o})_j - y_j.$$

- This indicates that the gradient is the difference between the model's predicted probability and the true label, similar to regression.

## Generalizing to Distributions

- If the label  $\mathbf{y}$  is now a probability vector (e.g.,  $(0.1, 0.2, 0.7)$ ), the cross-entropy loss remains applicable, interpreting it as the expected value of the loss for a distribution over labels.
- The cross-entropy loss measures the efficiency of encoding the actual outcome  $\mathbf{y}$  based on the predicted outcome  $\hat{\mathbf{y}}$ . For further details on information theory, see references like Cover & Thomas (1999) or MacKay (2003).

### ✓ 4.1.3. Information Theory Basics



## ✓ Overview

- **Information theory** focuses on encoding, decoding, transmitting, and manipulating data.

### Entropy

- **Entropy** quantifies the amount of information in a distribution  $P$ :

$$H[P] = \sum_j -P(j) \log P(j).$$

- The minimum bits required to encode data from distribution  $P$  is at least  $H[P]$  "nats," with one nat equivalent to approximately 1.44 bits (base  $e$  vs. base 2).

### Surprisal

- **Surprisal** quantifies unexpectedness:

$$-\log P(j)$$

- If events are predictable (e.g., a constant stream), they are easy to compress and transmit. High unpredictability results in higher surprisal.

### Cross-Entropy Revisited

- **Cross-entropy** from  $P$  to  $Q$  measures the expected surprisal when using  $Q$  to predict data from  $P$ :

$$H(P, Q) = \sum_j -P(j) \log Q(j).$$

- The minimum cross-entropy occurs when  $P = Q$ , resulting in

$$H(P, P) = H(P)$$

### Summary

- The cross-entropy classification objective can be viewed as:
  1. Maximizing the likelihood of observed data.
  2. Minimizing surprisal (and hence the bits needed for communication).

Double-click (or enter) to edit

## ✓ 4.1.4. Summary and Discussion

- **First Nontrivial Loss Function:** We explored a loss function for optimizing discrete output spaces, taking a probabilistic approach by treating categories as draws from a probability distribution.
- **Softmax Activation:** Introduced the softmax function, which converts neural network outputs into valid discrete probability distributions.
- **Derivative Behavior:** Observed that the derivative of the cross-entropy loss with softmax resembles the squared error derivative, focusing on the difference between expected behavior and predictions.
- **Connections to Other Fields:** Highlighted links to statistical physics and information theory, offering a broader perspective on the concepts.
- **Computational Considerations:** Addressed the high computational cost of fully connected layers ( $\mathcal{O}(dq)$ ) and discussed methods to reduce this through approximation and compression techniques, such as:
  - Deep Fried Convnets, which use permutations and Fourier transforms.
  - Advanced structural matrix approximations.
  - Quaternion-like decompositions for further cost reduction ( $\mathcal{O}(\frac{dq}{n})$ ), trading some accuracy for efficiency.
- **Research Challenges:** Emphasized the ongoing research aimed at optimizing execution efficiency on modern GPUs, balancing compact representations and computational load rather than merely minimizing operations.

## 4.2. The Image Classification Dataset

## 4.3. The Base Classification Model

## 4.4. Softmax Regression Implementation from Scratch

## 5.1. Multilayer Perceptrons

## 5.2. Implementation of Multilayer Perceptrons

### ✓ 5.3. Forward Propagation, Backward Propagation, and Computational Graphs

Start coding or [generate](#) with AI.