COSE474-2024F: DEEP LEARNING

0.1 Installation

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

```
Requirement already satisfied: d2l==1.0.3 in /usr/local/lib/python3.10/dist-packages (1.0.3)
Requirement already satisfied: jupyter==1.0.0 in /usr/local/lib/python3.10/dist-packages (from d2l==1.0.3) (1.0.0)
Requirement already satisfied: numpy==1.23.5 in /usr/local/lib/python3.10/dist-packages (from d2l==1.0.3) (1.23.5)
Requirement already satisfied: matplotlib==3.7.2 in /usr/local/lib/python3.10/dist-packages (from d2l==1.0.3) (3.7.2
Requirement already satisfied: matplotlib-inline==0.1.6 in /usr/local/lib/python3.10/dist-packages (from d2l==1.0.3)
Requirement already satisfied: requests==2.31.0 in /usr/local/lib/python3.10/dist-packages (from d2l==1.0.3) (2.31.0
Requirement already satisfied: pandas==2.0.3 in /usr/local/lib/python3.10/dist-packages (from d2l==1.0.3) (2.0.3)
Requirement already satisfied: scipy==1.10.1 in /usr/local/lib/python3.10/dist-packages (from d2l==1.0.3) (1.10.1)
Requirement already satisfied: notebook in /usr/local/lib/python3.10/dist-packages (from jupyter==1.0.0->d2l==1.0.3)
Requirement already satisfied: qtconsole in /usr/local/lib/python3.10/dist-packages (from jupyter==1.0.0->d2l==1.0.3
Requirement already satisfied: jupyter-console in /usr/local/lib/python3.10/dist-packages (from jupyter==1.0.0->d2l=
Requirement already satisfied: nbconvert in /usr/local/lib/python3.10/dist-packages (from jupyter==1.0.0->d2l==1.0.3
Requirement already satisfied: ipykernel in /usr/local/lib/python3.10/dist-packages (from jupyter==1.0.0->d2l==1.0.3
Requirement already satisfied: ipywidgets in /usr/local/lib/python3.10/dist-packages (from jupyter==1.0.0->d2l==1.0.
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib==3.7.2->
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (from matplotlib==3.7.2->d2l=
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib==3.7.2-
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib==3.7.2-
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib==3.7.2->d
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib==3.7.2->d2l
Requirement already satisfied: pyparsing<3.1,>=2.3.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib==3.
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.10/dist-packages (from matplotlib==3.7
Requirement already satisfied: traitlets in /usr/local/lib/python3.10/dist-packages (from matplotlib-inline==0.1.6->
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas==2.0.3->d2l==1.0
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/python3.10/dist-packages (from pandas==2.0.3->d2l==1
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests==2
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests==2.31.0->d2l==
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests==2.31.0-
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests==2.31.0->
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.7->matplo
Requirement already satisfied: ipython-genutils in /usr/local/lib/python3.10/dist-packages (from ipykernel->jupyter==
Requirement already satisfied: ipython>=5.0.0 in /usr/local/lib/python3.10/dist-packages (from ipykernel->jupyter==1.
Requirement already satisfied: jupyter-client in /usr/local/lib/python3.10/dist-packages (from ipykernel->jupyter==1.
Requirement already satisfied: tornado>=4.2 in /usr/local/lib/python3.10/dist-packages (from ipykernel->jupyter==1.0.
Requirement already satisfied: widgetsnbextension~=3.6.0 in /usr/local/lib/python3.10/dist-packages (from ipywidgets-
Requirement already satisfied: jupyterlab-widgets>=1.0.0 in /usr/local/lib/python3.10/dist-packages (from ipywidgets-
Requirement already satisfied: prompt-toolkit!=3.0.0,!=3.0.1,<3.1.0,>=2.0.0 in /usr/local/lib/python3.10/dist-package
Requirement already satisfied: pygments in /usr/local/lib/python3.10/dist-packages (from jupyter-console->jupyter==1.
Requirement already satisfied: lxml in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0.0->d2l==
Requirement already satisfied: beautifulsoup4 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.
Requirement already satisfied: bleach in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0.0->d2l
Requirement already satisfied: defusedxml in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0.0-
Requirement already satisfied: entrypoints>=0.2.2 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter
Requirement already satisfied: jinja2>=3.0 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0.0
Requirement already satisfied: jupyter-core>=4.7 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter=
Requirement already satisfied: jupyterlab-pygments in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyte
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1
Requirement already satisfied: mistune<2,>=0.8.1 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter=
Requirement already satisfied: nbclient>=0.5.0 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1
Requirement already satisfied: nbformat>=5.1 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0
Requirement already satisfied: pandocfilters>=1.4.1 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyt
Requirement already satisfied: tinycss2 in /usr/local/lib/python3.10/dist-packages (from nbconvert->jupyter==1.0.0->d
Requirement already satisfied: pyzmq<25,>=17 in /usr/local/lib/python3.10/dist-packages (from notebook->jupyter==1.0.
Requirement already satisfied: argon2-cffi in /usr/local/lib/python3.10/dist-packages (from notebook->jupyter==1.0.0-
Requirement already satisfied: nest-asyncio>=1.5 in /usr/local/lib/python3.10/dist-packages (from notebook->jupyter==
Requirement already satisfied: Send2Trash>=1.8.0 in /usr/local/lib/python3.10/dist-packages (from notebook->jupyter==
Requirement already satisfied: terminado>=0.8.3 in /usr/local/lib/python3.10/dist-packages (from notebook->jupyter==1
Requirement already satisfied: prometheus-client in /usr/local/lib/python3.10/dist-packages (from notebook->jupyter==
```

2.1 Data Manipulation

```
import torch
```

```
x = torch.arange(12, dtype=torch.float32)
x
```

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

```
x.numel()
```

```
12
```

```
x.shape
```

```
torch.Size([12])
```

```
X = x.reshape(3, 4)
X
```

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

```
torch.zeros((2, 3, 4))
```

```
tensor([[[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]],

        [[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]]])
```

```
torch.ones((2, 3, 4))
```

```
tensor([[[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]],

        [[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]]])
```

```
torch.randn(3, 4)
```

```
tensor([[-0.7075,  1.9021,  0.0149,  0.7629],
        [ 1.2723, -0.0989,  0.3217, -0.5446],
        [ 1.9003, -1.0701,  0.2239, -0.5614]])
```

```
torch.tensor([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
```

```
tensor([[2, 1, 4, 3],
        [1, 2, 3, 4],
        [4, 3, 2, 1]])
```

```
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.]])
```

```
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])
```

```
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.]))
```

```
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.,  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.)
```

```
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]])
```

```
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): 138391842370032
id(Z): 138391842370032
```

```
before = id(x)
x += y
id(x) == before
```

```
True
```

```
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)
```

Discussions/Takeaways/Exercises

How do the features of tensors in deep learning frameworks, such as PyTorch, enhance data manipulation compared to traditional array structures like NumPy's ndarray? Tensors can automatically calculate gradients, which helps making the NN training easier and faster. When working on multidimensional data, it is easier to work with as it is able to handle calculations between different dimensions and shapes.

2.2 Data Preprocessing

```
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
```

```
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
```

```
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))
```

Discussions/Takeaways/Exercises

1. Data Preparation: Separate input data from the target values. Missing values in the dataset are marked as NaN. We can deal with these either by deleting them or by imputation (filling them with estimated values). For categorical data like roof types, missing values can be treated as a separate category using `pd.get_dummies()`. For numerical data, we often replace missing values with the mean of the column.
2. Converting to Tensors: After cleaning, convert the data into tensors (a data format for deep learning) using `torch.tensor()`.
3. Data quality is crucial, as real-world data can include outliers, errors, or faulty measurements.

2.3 Linear Algebra

```
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.))
```

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

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

```
x[2]
```

```
tensor(2)
```

```
len(x)
```

```
3
```

```
x.shape
```

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

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

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

```
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]])
```

```
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]]])
```

```
A = torch.arange(6, dtype=torch.float32).reshape(2, 3)
B = A.clone()
A, A + B
```

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

```
A * B
```

```
tensor([[ 0.,  1.,  4.],
        [ 9., 16., 25.]])
```

```
a = 2
X = torch.arange(24).reshape(2, 3, 4)
a + X, (a * X).shape
```

```
(tensor([[[ 2,  3,  4,  5],
          [ 6,  7,  8,  9],
          [10, 11, 12, 13]],

         [[14, 15, 16, 17],
          [18, 19, 20, 21],
          [22, 23, 24, 25]]]),
 torch.Size([2, 3, 4]))
```

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

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

```
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.sum(axis=[0, 1]) == 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]))
```

```
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.]])
```

```
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.)
```

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

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

```
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.]]))
```

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

```
tensor(5.)
```

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

```
tensor(7.)
```

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

```
tensor(6.)
```

Discussions/Takeaways/Exercises

```
## Prove that the transpose of the transpose of a matrix is the matrix itself:  (A⊤)⊤=A .

A = torch.rand(3, 3)
ATT = A.T.T
print("Matrix A:\n", A)
print("Transpose of Transpose of A:\n", ATT)
print("Are they equal? ", torch.equal(A, ATT))
```

```
Matrix A:
 tensor([[0.9435, 0.3224, 0.9642],
        [0.8715, 0.0551, 0.4734],
        [0.7024, 0.1754, 0.1699]])
Transpose of Transpose of A:
 tensor([[0.9435, 0.3224, 0.9642],
        [0.8715, 0.0551, 0.4734],
        [0.7024, 0.1754, 0.1699]])
Are they equal?  True
```

```
## Given two matrices A and B, show that sum and transposition commute: A⊤+B⊤=(A+B)⊤.
B = torch.rand(3, 3)
transpose_sum = A.T + B.T
sum_transpose = (A + B).T
print("Transpose of A + Transpose of B:\n", transpose_sum)
print("Transpose of (A + B):\n", sum_transpose)
print("equal? ", torch.equal(transpose_sum, sum_transpose))
```

```
Transpose of A + Transpose of B:
 tensor([[0.9593, 1.3795, 1.3323],
        [0.6672, 0.6170, 1.1053],
        [1.7941, 0.7806, 0.4123]])
Transpose of (A + B):
 tensor([[0.9593, 1.3795, 1.3323],
        [0.6672, 0.6170, 1.1053],
        [1.7941, 0.7806, 0.4123]])
equal?  True
```

```
## Given any square matrix  A , is  A+A⊤  always symmetric? Can you prove the result by using only the results of the p
A_square = torch.rand(3, 3)
A_plus_AT = A_square + A_square.T
print("A + A.T:\n", A_plus_AT)
print("Is (A + A.T) symmetric? ", torch.equal(A_plus_AT, A_plus_AT.T))
```

```
A + A.T:
 tensor([[0.9308, 1.0228, 1.3845],
        [1.0228, 1.0647, 0.6317],
        [1.3845, 0.6317, 1.5789]])
Is (A + A.T) symmetric?  True
```

Key points:

- Scalars, vectors, matrices, and tensors are fundamental objects with zero, one, two, and multiple axes.
- Tensors can be sliced or reduced along axes using indexing, sum, or mean operations.
- Hadamard products are elementwise, while dot products and matrix multiplications return different-shaped results.
- Matrix-matrix products are more time-consuming to compute than Hadamard products.
- Norms measure the size or distance of vectors and matrices, with common examples being the ℓ1, ℓ2, spectral, and Frobenius norms.

2.5. Automatic Differentiation

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

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

```
x.requires_grad_(True)
x.grad
```

```
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])
```

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

```
tensor([1., 1., 1., 1.])
```

```
x.grad.zero_()
y = x * x
y.backward(gradient=torch.ones(len(y)))
x.grad
```

```
tensor([0., 2., 4., 6.])
```

```
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])
```

```
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)
```

Discussions/Takeaways/Exercises

The basics: (i) attach gradients to those variables with respect to which we desire derivatives; (ii) record the computation of the target value; (iii) execute the backpropagation function; and (iv) access the resulting gradient. Exercises:

1. Why is the second derivative more expensive to compute than the first? The second derivative requires computing the derivative of the gradient, adding another layer of complexity and calculation.
2. Running backpropagation again without resetting can lead to errors, as gradients are accumulated unless explicitly cleared. Impact of changing a in the control flow example:
3. When might you want to use forward, and when backward, differentiation? Use forward differentiation when fewer intermediate steps are involved (e.g., for small inputs) and backward differentiation for larger models or datasets where parallelization and memory efficiency are crucial.

How does automatic differentiation improve the efficiency and scalability of deep learning models, particularly when dealing with complex neural networks? Automatic differentiation allows deep learning models to be trained more efficiently and at scale by reducing errors, saving time, optimizing memory, and adapting to model complexity dynamically.

3.1. Linear Regression

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

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

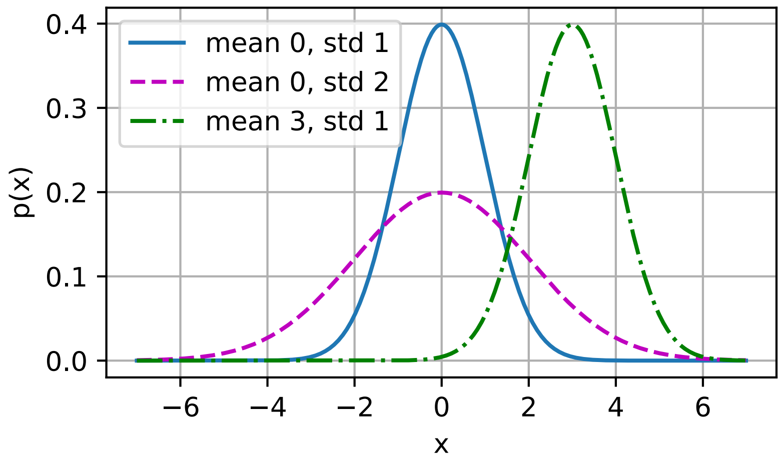
```
c = torch.zeros(n)
t = time.time()
for i in range(n):
    c[i] = a[i] + b[i]
f'{time.time() - t:.5f} sec'
```

```
'0.21973 sec'
```

```
t = time.time()
d = a + b
f'{time.time() - t:.5f} sec'
```

```
'0.00031 sec'
```

```
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)


x = np.arange(-7, 7, 0.01)

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])
```



Discussions/Takeaways/Exercises

- Regression problems pop up whenever we want to predict a numerical value. Common examples include predicting prices (of homes, stocks, etc.), predicting the length of stay (for patients in the hospital), forecasting demand (for retail sales), among numerous others. Not every prediction problem is one of classical regression.
- In traditional linear regression, the parameters of a linear function are chosen to minimize squared loss on the training set.
- How does linear regression compare to other complex models like decision trees or neural networks in terms of interpretability and performance? For linear Regression, it is highly interpretable and efficient for linear relationships, but limited in capturing complex patterns. While decision Trees are more flexible in modeling relationships with moderate interpretability, but sometimes overfit. lastly neural networks are vert powerful for complex datasets, but lack interpretability and require big amount of computational resources.

3.2. Object-Oriented Design for Implementation

```
import time
import numpy as np
import torch
```

```
from torch import nn
from d2l import torch as d2l


def add_to_class(Class):
    def wrapper(obj):
        setattr(Class, obj.__name__, obj)
    return wrapper


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()
```

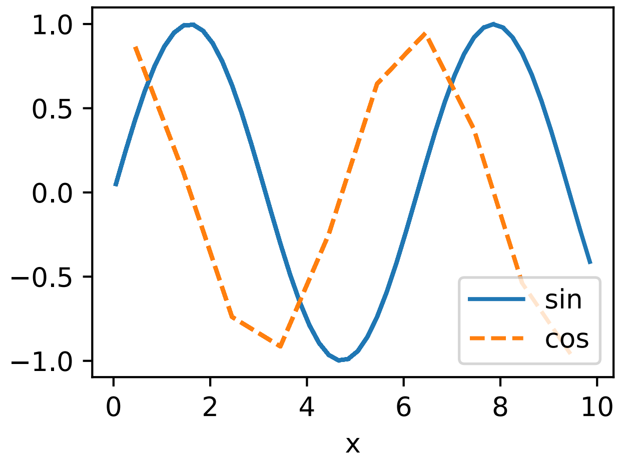
```
Class attribute "b" is 1
```

```
class HyperParameters:
    def save_hyperparameters(self, ignore=[]):
        raise NotImplemented


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):
    def __init__(self, xlabel=None, ylabel=None, xlim=None,
                 ylim=None, xscale='linear', yscale='linear',
                 ls=['-', '--', '-.', ':'], colors=['C0', 'C1', 'C2', 'C3'],
                 fig=None, axes=None, figsize=(3.5, 2.5), display=True):
        self.save_hyperparameters()

    def draw(self, x, y, label, every_n=1):
        raise NotImplemented
```

```
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)
```



```
class Module(nn.Module, d2l.HyperParameters):
    def __init__(self, plot_train_per_epoch=2, plot_valid_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'), 'Neural network is defined'
        return self.net(X)

    def plot(self, key, value, train):
        assert hasattr(self, 'trainer'), 'Trainer is not inited'
        self.board.xlabel = 'epoch'
        if train:
            x = self.trainer.train_batch_idx / \
                self.trainer.num_train_batches
            n = self.trainer.num_train_batches / \
                self.plot_train_per_epoch
        else:
            x = self.trainer.epoch + 1
            n = self.trainer.num_val_batches / \
                self.plot_valid_per_epoch
        self.board.draw(x, value.to(d2l.cpu()).detach().numpy(),
                        ('train_' if train else 'val_') + key,
                        every_n=int(n))

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

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

    def configure_optimizers(self):
        raise NotImplementedError
```

```
class DataModule(d2l.HyperParameters):
    def __init__(self, root='../data', num_workers=4):
        self.save_hyperparameters()

    def get_dataloader(self, train):
        raise NotImplementedError

    def train_dataloader(self):
        return self.get_dataloader(train=True)

    def val_dataloader(self):
        return self.get_dataloader(train=False)


class Trainer(d2l.HyperParameters):
    def __init__(self, max_epochs, num_gpus=0, gradient_clip_val=0):
        self.save_hyperparameters()
        assert num_gpus == 0, 'No GPU support yet'

    def prepare_data(self, data):
        self.train_dataloader = data.train_dataloader()
        self.val_dataloader = data.val_dataloader()
        self.num_train_batches = len(self.train_dataloader)
        self.num_val_batches = (len(self.val_dataloader)
                                if self.val_dataloader is not None else 0)

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

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

    def fit_epoch(self):
        raise NotImplementedError
```

Discussions/Takeaways/Exercises

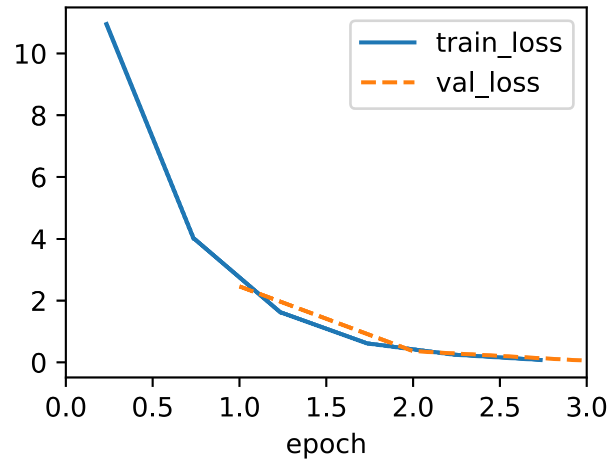
Given the utility of the `add_to_class` function in adding methods to classes, what are the advantages and disadvantages of using such a utility in the design of deep learning? It enhances modularity and flexibility in deep learning frameworks by allowing developers to define methods in separate code blocks, which promotes a cleaner and more organized codebase. This design enables easy extension of existing classes without modifying their core structure, thereby reducing the risk of introducing bugs and making it simpler for teams to collaborate on different functionalities. Additionally, the dynamic nature of this utility allows developers to quickly iterate on models and functionalities without the need to create new classes.With these advantages, it also presents challenges related to readability, debugging, and the learning curve for new developers. Striking a balance between flexibility and maintainability is crucial for ensuring that the benefits outweigh the drawbacks in deep learning framework design. Proper documentation and clear guidelines on how to use such utilities can help mitigate some of the potential pitfalls associated with dynamic method addition.

3.4. Linear Regression Implementation from Scratch

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

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


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


class SGD(d2l.HyperParameters):
    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_()


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


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

@d2l.add_to_class(d2l.Trainer)
def fit_epoch(self):
    self.model.train()
    for batch in self.train_dataloader:
        loss = self.model.training_step(self.prepare_batch(batch))
        self.optim.zero_grad()
        with torch.no_grad():
            loss.backward()
            if self.gradient_clip_val > 0:
                self.clip_gradients(self.gradient_clip_val, self.model)
            self.optim.step()
        self.train_batch_idx += 1
    if self.val_dataloader is None:
        return
    self.model.eval()
    for batch in self.val_dataloader:
        with torch.no_grad():
            self.model.validation_step(self.prepare_batch(batch))
        self.val_batch_idx += 1
```

```
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}')
```

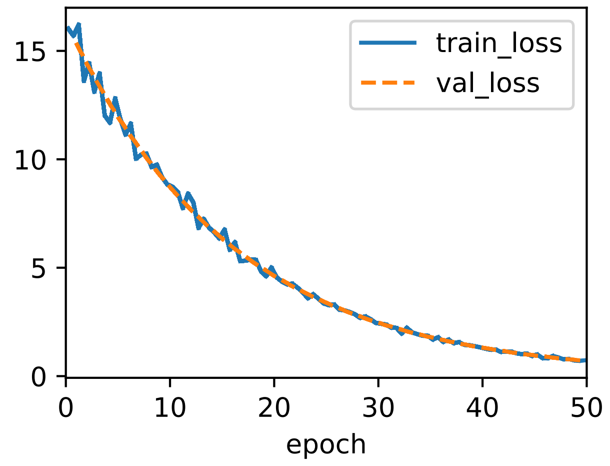
```
error in estimating w: tensor([ 0.1606, -0.1804])
error in estimating b: tensor([0.2307])
```

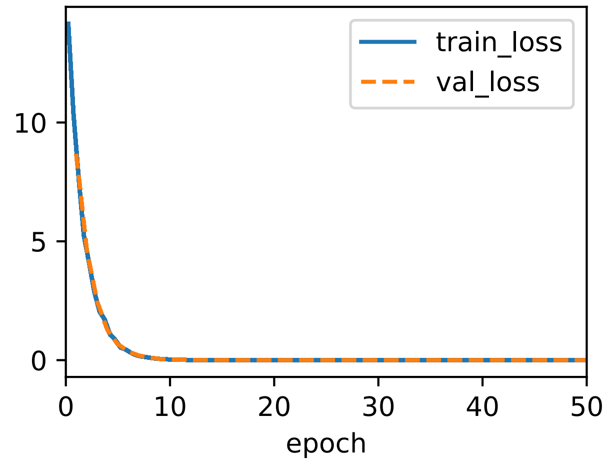
Discussions/Takeaways/Exercises

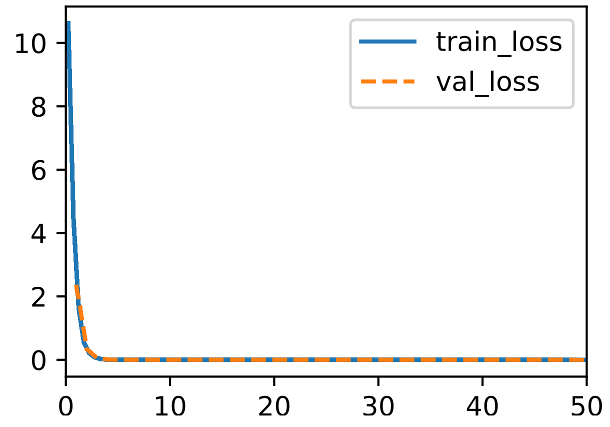
Build a data loader, a model, a loss function, an optimization procedure, and a visualization and monitoring tool.

1. What would happen if we were to initialize the weights to zero. Would the algorithm still work? What if we initialized the parameters with variance 1000 rather than 0.01 ? If we initialize the weights to zero, the algorithm will not work properly for neural networks with more than one layer). Because every weight update will be the same for all input features, leading to no learning and a symmetry problem. The gradients for all weights would be identical, resulting in no distinct adjustments to the parameters. If we initialize the weights with a high variance, the initial predictions will be inaccurate, resulting in a loss.
2. Experiment using different learning rates to find out how quickly the loss function value drops.

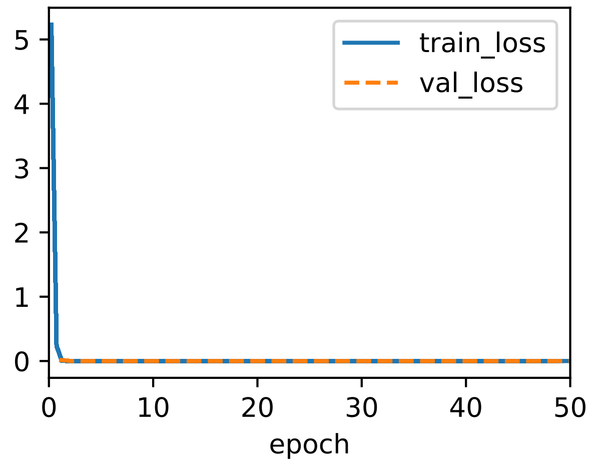
```
learning_rates = [0.001, 0.01, 0.03, 0.1]
for lr in learning_rates:
    model = LinearRegressionScratch(2, lr=lr)
    trainer = d2l.Trainer(max_epochs=50)
    trainer.fit(model, data)
```


train_loss
val_loss
epoch


train_loss
val_loss
epoch


train_loss
val_loss

epoch



How does the choice of learning rate affect the training process of a linear regression model, and what strategies can be employed to select an appropriate learning rate? The learning rate significantly impacts the training process of a linear regression model. Using strategies like learning rate schedules, using a learning rate finder, using adaptive learning rate methods, and performing hyperparameter searches can help in identifying an effective learning rate, enhancing model performance.

4.1. Softmax Regression

Discussions/Takeaways/Exercises

1. Softmax Regression Overview Purpose: Softmax regression is a classification technique, shifting focus from regression's "how much?" questions to classification's "which category?" questions. Examples: spam detection in emails, customer sign-up predictions, and multi-label classifications like news articles covering various topics.

2. Classification Problems:

- Involves hard assignments (dog, cat) and soft assignments (probabilities of categories).
- Multi-label classification: More than one category can be true for a single instance.

3. Label Representation:

- One-hot encoding: Represents categories in binary vectors (e.g., cat = (1, 0, 0), chicken = (0, 1, 0), dog = (0, 0, 1)).
- Suitable for categorical data without natural ordering.

4. Linear Models for Classification:

- Multiple outputs required for classification, one for each class.
- Uses a single-layer neural network with weights and biases for predictions.

5. The Softmax Function

- Functionality: Converts raw output scores (logits) into probabilities.
- Outputs sum to 1, ensuring valid probability distribution.
- Maintains order of logits, allowing for straightforward classification.

Explore how different regularization techniques (e.g., L1, L2 regularization) can affect the weights used in softmax regression. How might this impact the interpretability and performance of the model? Regularization techniques such as L1 and L2 significantly influence the weights in softmax regression, impacting both model interpretability and performance. L1 regularization promotes sparsity and enhances interpretability by selecting important features, while L2 regularization leads to smaller weights, promoting stability and robustness. The choice of regularization should depend on the model’s goals, whether prioritizing interpretability or overall predictive performance.

4.2. The Image Classification Dataset

```
%matplotlib inline
import time
import torch
import torchvision
```

```
from torchvision import transforms
from d2l import torch as d2l

d2l.use_svg_display()


class FashionMNIST(d2l.DataModule):
    def __init__(self, batch_size=64, resize=(28, 28)):
        super().__init__()
        self.save_hyperparameters()
        trans = transforms.Compose([transforms.Resize(resize),
                                    transforms.ToTensor()])
        self.train = torchvision.datasets.FashionMNIST(
            root=self.root, train=True, transform=trans, download=True)
        self.val = torchvision.datasets.FashionMNIST(
            root=self.root, train=False, transform=trans, download=True)


data = FashionMNIST(resize=(32, 32))
len(data.train), len(data.val)
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz to ../data/Fashion
100%|██████████| 26421880/26421880 [00:01<00:00, 13398100.52it/s]
Extracting ../data/FashionMNIST/raw/train-images-idx3-ubyte.gz to ../data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz to ../data/Fashion
100%|██████████| 29515/29515 [00:00<00:00, 231800.01it/s]
Extracting ../data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to ../data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to ../data/FashionM
100%|██████████| 4422102/4422102 [00:01<00:00, 4212503.46it/s]
Extracting ../data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to ../data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to ../data/FashionM
100%|██████████| 5148/5148 [00:00<00:00, 12898612.30it/s]
Extracting ../data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to ../data/FashionMNIST/raw
```

```
(60000, 10000)
```

```python
data.train[0][0].shape
```

```
torch.Size([1, 32, 32])
```

```python
@d2l.add_to_class(FashionMNIST)
def text_labels(self, indices):
    """Return text labels."""
    labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
              'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
    return [labels[int(i)] for i in indices]
```

```python
@d2l.add_to_class(FashionMNIST)
def get_dataloader(self, train):
    data = self.train if train else self.val
    return torch.utils.data.DataLoader(data, self.batch_size, shuffle=train,
                                       num_workers=self.num_workers)
```

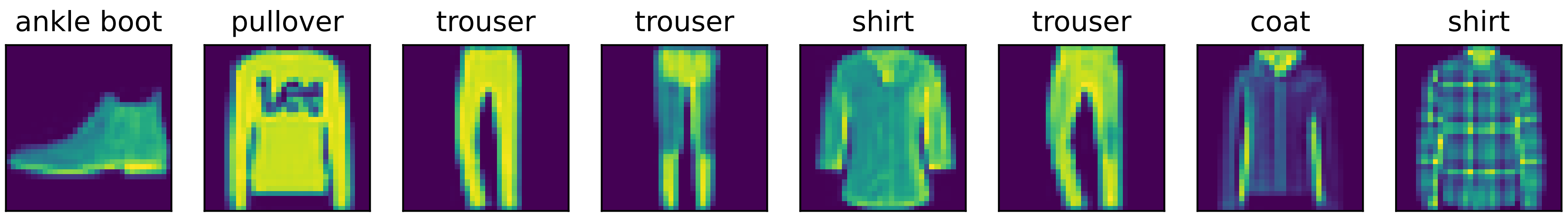
```python
X, y = next(iter(data.train_dataloader()))
print(X.shape, X.dtype, y.shape, y.dtype)
```

```
/usr/local/lib/python3.10/dist-packages/torch/utils/data/dataloader.py:557: UserWarning: This DataLoader will create
  warnings.warn(_create_warning_msg(
torch.Size([64, 1, 32, 32]) torch.float32 torch.Size([64]) torch.int64
```

```python
tic = time.time()
for X, y in data.train_dataloader():
    continue
f'{time.time() - tic:.2f} sec'
```

```
'17.47 sec'
```

```
def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):
    raise NotImplementedError


@d2l.add_to_class(FashionMNIST)
def visualize(self, batch, nrows=1, ncols=8, labels=[]):
    X, y = batch
    if not labels:
        labels = self.text_labels(y)
    d2l.show_images(X.squeeze(1), nrows, ncols, titles=labels)
batch = next(iter(data.val_dataloader()))
data.visualize(batch)
```



Discussions/Takeaways/Exercises

1. Does reducing the batch_size (for instance, to 1) affect the reading performance? Yes, reducing the batch_size to 1 affects the reading performance. With a batch size of 1, the data iterator reads images one at a time, which increases the overhead related to image loading and processing, which leads to slower overall performance due to the increased number of iterations required to process the entire dataset.
2. The data iterator performance is important. Do you think the current implementation is fast enough? The current implementation may not be fast enough, if it spends a significant amount of time reading data rather than processing it. To improve the data loading performance, we could ensure that data is available in memory before it is needed for processing, or caching frequently accessed data or using an in-memory dataset can significantly reduce loading times.
3. How does the choice of batch size impact the performance of training deep learning models, and what considerations should be taken into account when selecting an appropriate batch size for a specific dataset? The choice of batch size is a very important hyperparameter

in training deep learning models. Smaller batch sizes allow for more frequent updates to model weights, potentially speeding up convergence. However, they can introduce noise in gradient estimation, leading to less stable convergence and requiring more iterations to complete an epoch. Larger batch sizes provide a more accurate gradient estimate and stable convergence but demand more memory and computational resources, possibly resulting in poorer generalization. When selecting an appropriate batch size, we shouldconsider memory limitations, model complexity, data size, as larger sizes can speed up training on extensive datasets by reducing the number of updates per epoch.

4.3. The Base Classification Model

```
class Classifier(d2l.Module):
    def validation_step(self, batch):
        Y_hat = self(*batch[:-1])
        self.plot('loss', self.loss(Y_hat, batch[-1]), train=False)
        self.plot('acc', self.accuracy(Y_hat, batch[-1]), train=False)


@d2l.add_to_class(d2l.Module)
def configure_optimizers(self):
    return torch.optim.SGD(self.parameters(), lr=self.lr)


@d2l.add_to_class(Classifier)
def accuracy(self, Y_hat, Y, averaged=True):
    Y_hat = Y_hat.reshape((-1, Y_hat.shape[-1]))
    preds = Y_hat.argmax(axis=1).type(Y.dtype)
    compare = (preds == Y.reshape(-1)).type(torch.float32)
    return compare.mean() if averaged else compare
```

Discussions/Takeaways/Exercises

- Classification is a sufficiently common problem that it warrants its own convenience functions.

- The ACCURACY of the classifier - is the most important thing in classification
- How do the choice of loss functions and performance metrics in classification tasks influence model training and evaluation? The choice of loss functions and performance metrics significantly impacts how classification models are trained and evaluated. Loss functions like Cross-Entropy guide optimization by measuring the gap between predictions and actual results, affecting training speed and stability. Different fields prioritize various metrics: for example, medical applications focus on minimizing false negatives, while marketing might address class imbalances.

4.4. Softmax Regression Implementation from Scratch

```
X = torch.tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
X.sum(0, keepdims=True), X.sum(1, keepdims=True)
```

```
(tensor([[5., 7., 9.]]),
 tensor([[ 6.],
         [15.]]))
```

```
def softmax(X):
    X_exp = torch.exp(X)
    partition = X_exp.sum(1, keepdims=True)
    return X_exp / partition
```

```
X = torch.rand((2, 5))
X_prob = softmax(X)
X_prob, X_prob.sum(1)
```

```
(tensor([[0.2514, 0.1656, 0.2591, 0.1477, 0.1761],
         [0.2425, 0.1813, 0.2848, 0.1511, 0.1403]]),
 tensor([1.0000, 1.0000]))
```

```
class SoftmaxRegressionScratch(d2l.Classifier):
    def __init__(self, num_inputs, num_outputs, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.W = torch.normal(0, sigma,
                              size=(num_inputs, num_outputs),
                              requires_grad=True)
        self.b = torch.zeros(num_outputs, requires_grad=True)

    def parameters(self):
        return [self.W, self.b]
```

```
@d2l.add_to_class(SoftmaxRegressionScratch)
def forward(self, X):
    X = X.reshape((-1, self.W.shape[0]))
    return softmax(torch.matmul(X, self.W) + self.b)
```

```
y = torch.tensor([0, 2])
y_hat = torch.tensor([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])
y_hat[[0, 1], y]
```

```
tensor([0.1000, 0.5000])
```

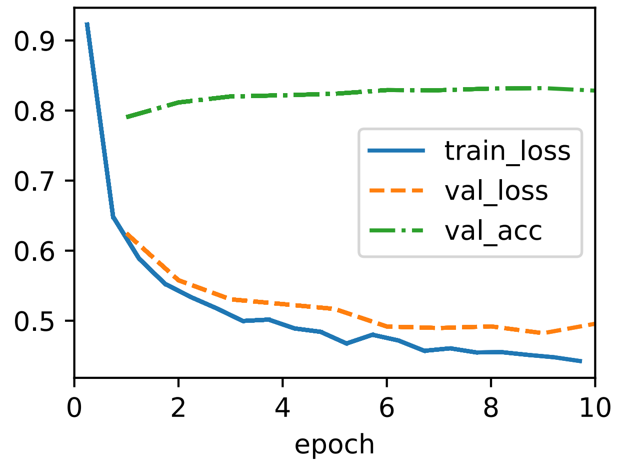
```
def cross_entropy(y_hat, y):
    return -torch.log(y_hat[list(range(len(y_hat))), y]).mean()

cross_entropy(y_hat, y)
```

```
tensor(1.4979)
```

```
@d2l.add_to_class(SoftmaxRegressionScratch)
def loss(self, y_hat, y):
    return cross_entropy(y_hat, y)
```

```
data = d2l.FashionMNIST(batch_size=256)
model = SoftmaxRegressionScratch(num_inputs=784, num_outputs=10, lr=0.1)
```

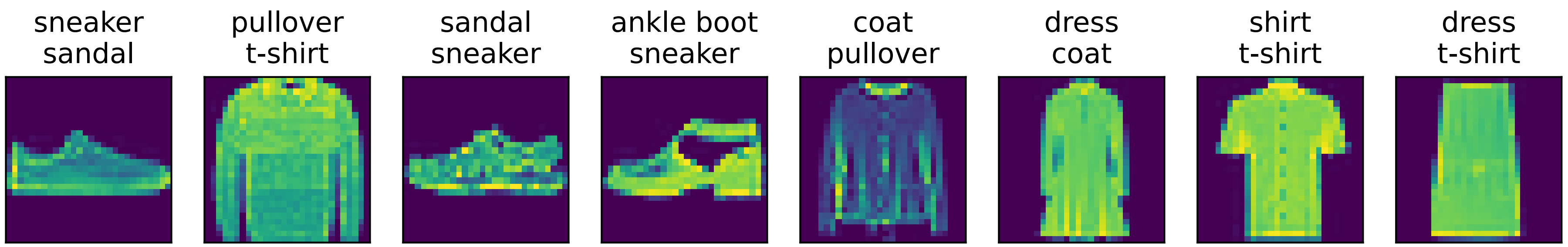
```
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```



```
X, y = next(iter(data.val_dataloader()))
preds = model(X).argmax(axis=1)
preds.shape
```

```
torch.Size([256])
```

```
wrong = preds.type(y.dtype) != y
X, y, preds = X[wrong], y[wrong], preds[wrong]
labels = [a+'\n'+b for a,
          b in zip(data.text_labels(y), data.text_labels(preds))]
data.visualize([X, y], labels=labels)
```

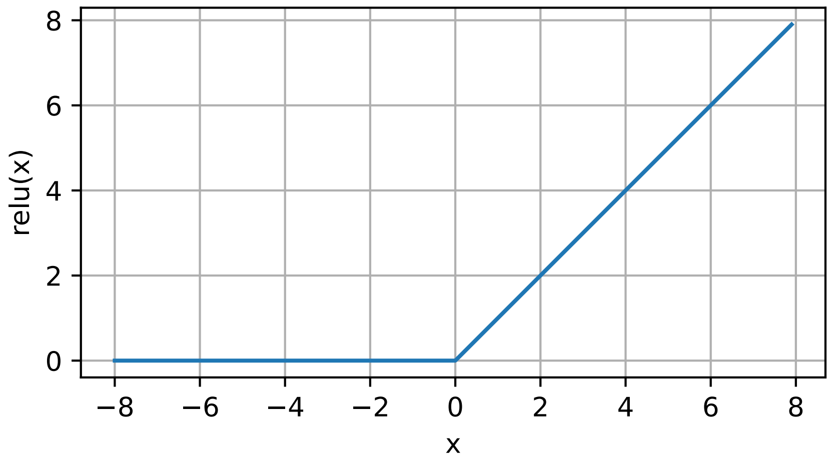


Discussions/Takeaways/Exercises

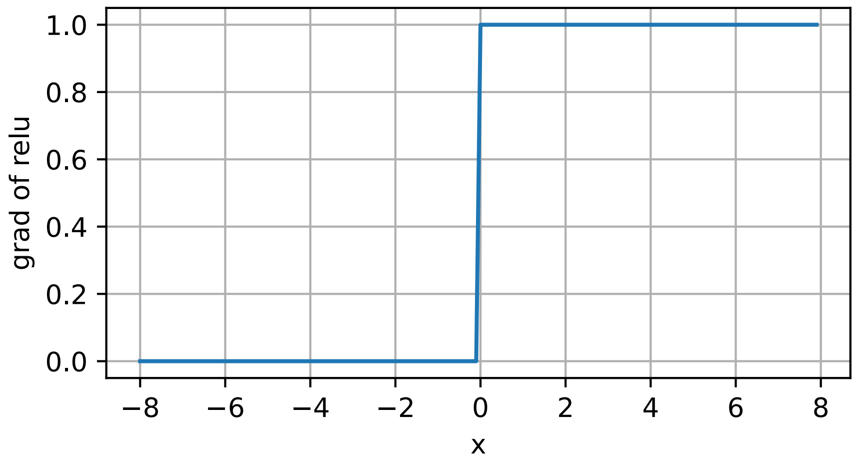
How do numerical stability issues affect the implementation of the softmax function in deep learning models, and what could be done to address these issues? Numerical stability issues can affect softmax function implementation in deep learning models, particularly when dealing with large or small input values. The exponential function used in softmax can result in very large outputs for high input values, leading to overflow, and very small outputs for low input values,leading to underflow, causing inaccuracies. To address these issues the following could be done: First, apply stability by shifting as it involves subtracting the maximum value from the input vector before applying the exponential function to prevent overflow, ensuring that the highest value is shifted to zero. Second, clipping can be done. It limits input values to a certain range, limiting extreme values to avoid overflow and underflow. Also, utilizing built-in functions from libraries like PyTorch or TensorFlow ensures that these stability issues are handled internally, making the softmax function more reliable.

5.1. Multilayer Perceptrons

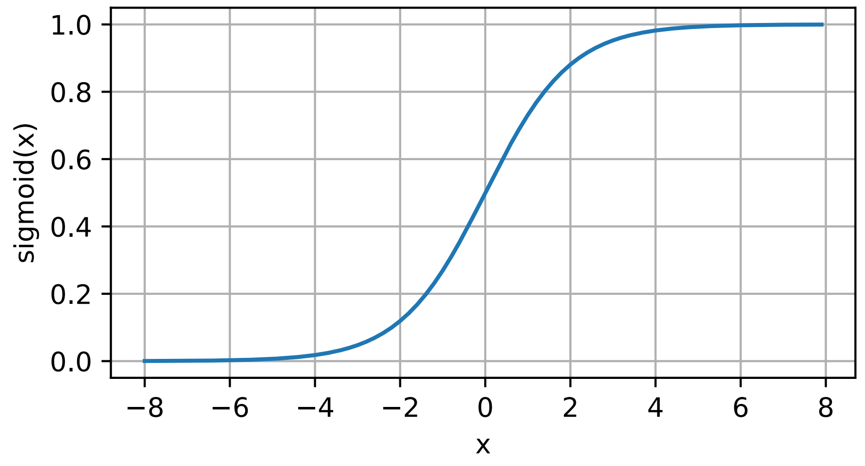
```
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.relu(x)
d2l.plot(x.detach(), y.detach(), 'x', 'relu(x)', figsize=(5, 2.5))
```



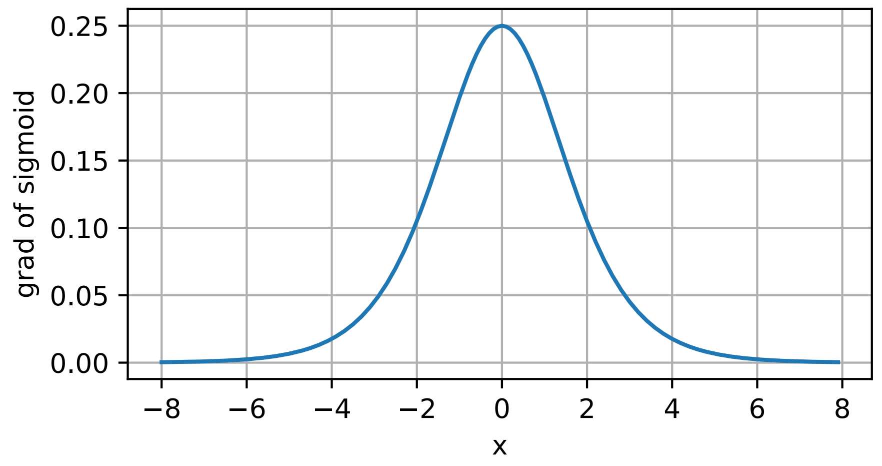
```
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of relu', figsize=(5, 2.5))
```



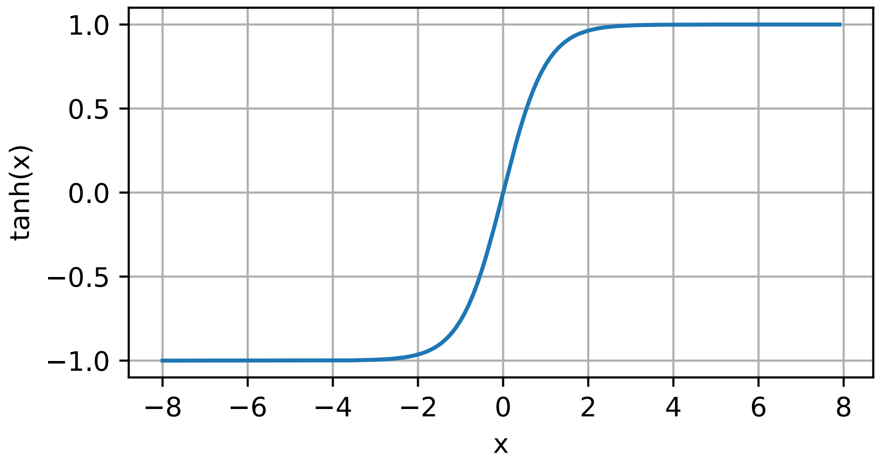
```
y = torch.sigmoid(x)
d2l.plot(x.detach(), y.detach(), 'x', 'sigmoid(x)', figsize=(5, 2.5))
```



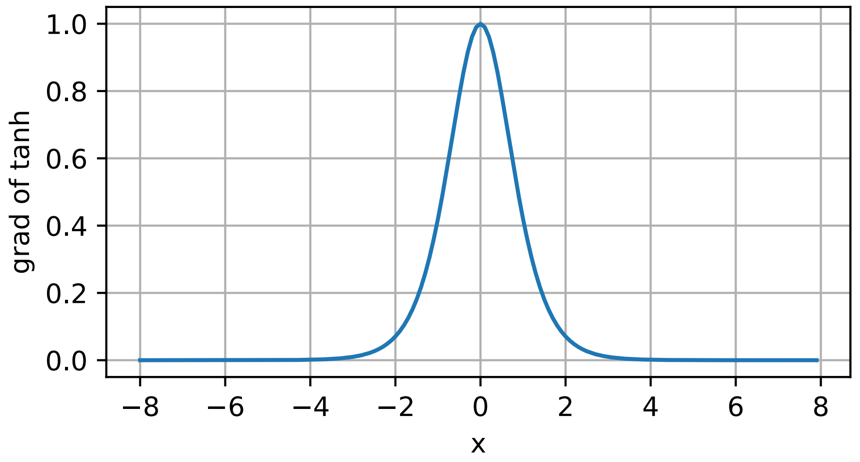
```
x.grad.data.zero_()
y.backward(torch.ones_like(x),retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of sigmoid', figsize=(5, 2.5))
```



```
y = torch.tanh(x)
d2l.plot(x.detach(), y.detach(), 'x', 'tanh(x)', figsize=(5, 2.5))
```



```
x.grad.data.zero_()
y.backward(torch.ones_like(x),retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of tanh', figsize=(5, 2.5))
```



Discussions/Takeaways/Exercises

- MLP - a way to overcome the limitations of linear models

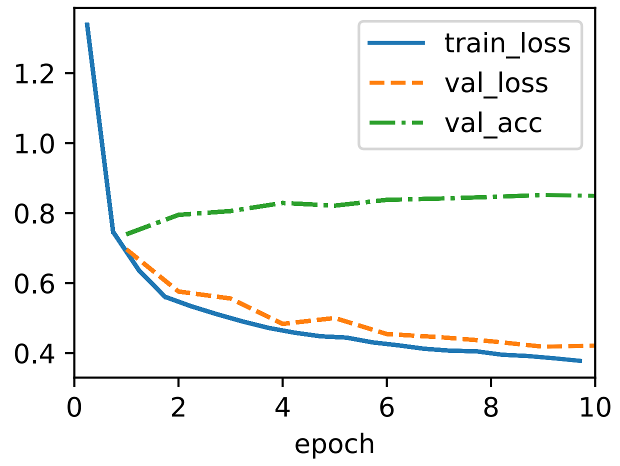
- Linearity implies the weaker assumption of monotonicity
- Incorporate hiddent layers
- Affine Transformations and Nonlinearities
- Activation Functions:

1. ReLU (Rectified Linear Unit): ReLU(x)=max(0,x). Helps mitigate vanishing gradients and provides simplicity in computation.
2. Sigmoid Function: Squashes inputs to the range (0, 1), often used for binary classification.
3. Tanh Function: Maps inputs to the range (-1, 1) and is point symmetric around the origin.

- A single hidden layer can approximate any continuous function given enough neurons, but learning that function effectively is the challenge.
- Deeper networks often provide better function approximations than wider networks.
- Assume that we have a nonlinearity that applies to one minibatch at a time, such as batch normalization. What kinds of problems do you expect this to cause? Applying a nonlinearity to one minibatch at a time can lead to inconsistencies during training and inference. For instance, if the normalization is based on the statistics of the current minibatch, it might not generalize well to unseen data, causing the model to behave erratically. This can also obstruc convergence since the model is optimizing based on statistics that change frequently, resulting in noisy gradients.
- How do the activation functions in MLPs affect the model's ability to learn complex patterns in data? Activation functions introduce non-linearity into the model, enabling it to learn complex patterns in data. By transforming the output of neurons, these functions allow MLPs to represent complex relationships that linear functions cannot.

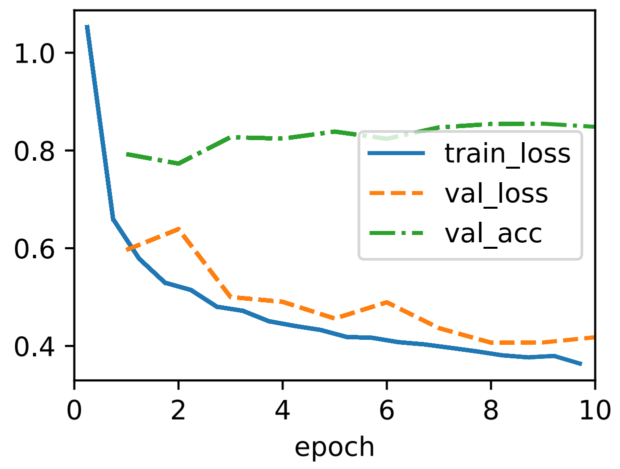
5.2. Implementation of Multilayer Perceptrons

```
class MLPScratch(d2l.Classifier):
    def __init__(self, num_inputs, num_outputs, num_hiddens, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.W1 = nn.Parameter(torch.randn(num_inputs, num_hiddens) * sigma)
        self.b1 = nn.Parameter(torch.zeros(num_hiddens))
        self.W2 = nn.Parameter(torch.randn(num_hiddens, num_outputs) * sigma)
        self.b2 = nn.Parameter(torch.zeros(num_outputs))


def relu(X):
    a = torch.zeros_like(X)
    return torch.max(X, a)


@d2l.add_to_class(MLPScratch)
def forward(self, X):
    X = X.reshape((-1, self.num_inputs))
    H = relu(torch.matmul(X, self.W1) + self.b1)
    return torch.matmul(H, self.W2) + self.b2


model = MLPScratch(num_inputs=784, num_outputs=10, num_hiddens=256, lr=0.1)
data = d2l.FashionMNIST(batch_size=256)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```



```
class MLP(d2l.Classifier):
    def __init__(self, num_outputs, num_hiddens, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(nn.Flatten(),
                                 nn.LazyLinear(num_hiddens),
                                 nn.ReLU(),
                                 nn.LazyLinear(num_outputs))
```

```
model = MLP(num_outputs=10, num_hiddens=256, lr=0.1)
trainer.fit(model, data)
```



Discussions/Takeaways/Exercises

How do different hyperparameters, such as the number of hidden units and learning rate, impact the performance of MLPs in classifying datasets like Fashion-MNIST? Different hyperparameters, such as the number of hidden units and learning rate, significantly influence the performance of MLPs in classifying datasets like Fashion-MNIST. Increasing the number of hidden units generally enhances the model's capacity to learn complex patterns, improving accuracy up to a certain point, beyond which overfitting may occur. Conversely, a learning rate that is too high can lead to divergence, while a rate that is too low may result in slow convergence, preventing the model from reaching optimal performance. Therefore, finding the right combination of these hyperparameters is very important, as it impacts classification accuracy.

5.3. Forward Propagation, Backward Propagation, and Computational Graphs

Discussions/Takeaways/Exercises

- Forward Propagation: Involves calculating and storing intermediate variables from the input layer to the output layer, enabling the neural network to produce outputs based on input data.
- Backpropagation: A method for calculating gradients of the network's parameters by traversing the network in reverse order using the chain rule, enabling updates to model parameters during training.

- Computational Graph: A graphical representation of the operations and dependencies in a neural network, facilitating understanding of how data flows and how gradients are computed.
- Weight Decay: Involves ℓ2 regularization to prevent overfitting by adding a regularization term to the loss function, which is calculated during forward propagation.
- Interdependence: Forward and backward propagation are interconnected; the output from forward propagation is required for backpropagation, and gradients from backpropagation influence future forward passes.
- Memory Usage: Training requires more memory than prediction due to the storage of intermediate variables during forward propagation, which are used for backpropagation.
- Gradient Calculation: Gradients of the loss function are computed sequentially for each layer and stored for updating model parameters, necessitating the retention of intermediate values until backpropagation is complete.
- Batch Size Impact: The size of intermediate values is proportional to the number of layers and the batch size, making deeper networks and larger batches more prone to out-of-memory errors.
- Efficient Computation: Backpropagation reuses stored intermediate values from forward propagation, reducing redundant calculations and improving efficiency during training.
- Training Process: Alternates between forward propagation and backpropagation, updating model parameters based on gradients derived from the objective function.
- Assume that the inputs $\mathbf{X}$ to some scalar function f are $n \times m$ matrices. What is the dimensionality of the gradient of f with respect to $\mathbf{X}$? The gradient of a scalar function f f with respect to X (an n×m matrix) has the same dimensionality as X.
- How does the implementation of forward and backward propagation techniques in neural networks impact the overall training efficiency and model performance? The implementation of forward and backward propagation techniques significantly influences the training efficiency and performance of neural networks by establishing a systematic approach for computing both outputs and gradients. Forward propagation computes the output layer by sequentially applying transformations from the input to the output, storing intermediate variables to make the gradient calculations more efficieent during backpropagation. This allows for effective gradient updates while minimizing computational redundancy; however, it also results in higher memory usage since intermediate values must be stored until backpropagation completion. Memory consumption and computational complexity must be balanced, as deeper networks with large batch sizes can lead to out-of-memory errors.