EE 519: Deep Learning Theory & Fundamentals **Spring 2023**

Homework 1
Due: April 21, 2023, 11:59PM PT

Student Name: Andy Franck *Instructor Name: John Lipor*

For this assignment, you will begin coding your own object-oriented deep learning library called *MyTorch*. This library will have the below file structure, allowing you to call functions in a manner similar to PyTorch (see included code). Your job is to complete the respective functions and test your deep learning library.

Note: This is a pure `numpy` implementation, so you may not import any additional libraries.

```
hw01files
└── mytorch
    ├── nn
    │   ├── activation.py
    │   ├── linear.py
    │   └── loss.py
    └── optim
        └── sgd.py
└── models
    └── mlp.py
```

Code: Please attach your code to the end of your final document. You do not need to link your code for each problem within gradescope, only the outputs of the respective jupyter notebooks.

Problem 1 `linear.py` (1 pt, 2 pts, 4 pts)

Your first task is to complete the function `linear.py`, which initializes a single linear layer with the given input and output sizes, and performs the forward and backward operations. Recall that the forward pass of a linear layer performs the operation

$$O = AW + 1_N b^T, \tag{1}$$

where $A \in \mathbb{R}^{N \times d}$ is the input data matrix of N examples, $W \in \mathbb{R}^{d \times h}$ is the weight matrix, $b \in \mathbb{R}^h$ is the bias term, and $1_N \in \mathbb{R}^N$ denotes the vector of all ones.

To perform backpropagation, we require the gradient of the loss with respect to the output O, which will be implemented in a later problem. We then wish to output the gradients with respect to the parameters W and b as well as the inputs A. Assuming we have access to $\partial L/\partial O$, these gradients are defined by the equations

$$\begin{aligned} \frac{\partial L}{\partial A} &= \frac{\partial L}{\partial O}\left(\frac{\partial O}{\partial A}\right)^T \\ \left(\frac{\partial L}{\partial W}\right)^T &= \left(\frac{\partial L}{\partial O}\right)^T \frac{\partial O}{\partial W} \\ \frac{\partial L}{\partial b} &= \left(\frac{\partial L}{\partial O}\right)^T \frac{\partial O}{\partial b}. \end{aligned} \tag{2}$$

(a) Implement the `__init__` function, initializing W using a normal distribution with variance 0.01 and b to be all zeros.

(b) Implement the `forward` method as defined by (1). This method receives the input data matrix A and returns the output data matrix O.

(c) Implement the `backward` method as defined by (2). Note that you need to determine the gradients of the output with respect to the various inputs yourself. This method receives $\partial L/\partial O$ as input, returns $\partial L/\partial A$, and stores $\partial L/\partial W$ and $\partial L/\partial b$ to the `linear` object.

Turn in the output of the notebook `LinearTester.ipynb`. Note that you can easily save the output in pdf form and attach to a LaTeXdocument using the `include` command.

Answer:

In [ ]:
```
%load_ext autoreload
%autoreload 2

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

import mytorch
from mytorch import nn as mynn
```

In [ ]:
```
# Load data for testing
data = np.load('testerData.npz')
W, b, X, Y, dLdZ = [data[fname] for fname in data.files]

[N, num_inputs] = X.shape
num_outputs = Y.shape[1]

# converted torch versions
Xt = torch.tensor(X).float()
Wt = torch.tensor(W).float()
bt = torch.tensor(b).float()
Yt = torch.tensor(Y).float()
```

In [ ]:
```
# initialize model and fix weights to true values
my_net = mynn.Linear(num_inputs, num_outputs)
my_net.W = W
my_net.b = b.flatten()

# initialize torch model, loss, optimizer
net = nn.Linear(num_inputs, num_outputs)
net.weight = nn.Parameter(Wt.T)
net.bias = nn.Parameter(bt[:, 0])
criterion = nn.MSELoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.1, momentum=0.0)
```

Compare `forward()`

In [ ]:
```
true_out = X @ W + np.outer(np.ones(N), b)
my_out = my_net.forward(X)
torch_out = net(Xt)

print('True:\n', true_out, '\n')
print('MyTorch:\n', my_out, '\n')
print('PyTorch:\n', torch_out.data, '\n')

print('Difference:', np.linalg.norm(my_out - torch_out.data.numpy()))
```

```
True:
 [[2.45419505 3.85377348 3.86239248 4.09861525 3.42552912]
 [2.37093331 3.22945671 2.88333967 3.03220271 2.72105395]
 [1.90472749 2.88840363 2.93526692 3.4361838  3.17490315]
 [1.45392748 2.79782519 2.16759199 2.56624407 2.53054982]
 [2.04588663 3.40446758 2.76237953 3.22387254 2.86430505]]

MyTorch:
 [[2.45419505 3.85377348 3.86239248 4.09861525 3.42552912]
 [2.37093331 3.22945671 2.88333967 3.03220271 2.72105395]
 [1.90472749 2.88840363 2.93526692 3.4361838  3.17490315]
 [1.45392748 2.79782519 2.16759199 2.56624407 2.53054982]
 [2.04588663 3.40446758 2.76237953 3.22387254 2.86430505]]

PyTorch:
 tensor([[2.4542, 3.8538, 3.8624, 4.0986, 3.4255],
        [2.3709, 3.2295, 2.8833, 3.0322, 2.7211],
        [1.9047, 2.8884, 2.9353, 3.4362, 3.1749],
        [1.4539, 2.7978, 2.1676, 2.5662, 2.5305],
        [2.0459, 3.4045, 2.7624, 3.2239, 2.8643]])

Difference: 7.96027202151818e-07
```

Compare `backward` and gradients

In [ ]:

```python
my_net.backward(dLdZ)
my_dLdW = my_net.dLdW
my_dLdb = my_net.dLdb

optimizer.zero_grad()
torch_loss_fn = nn.MSELoss()
torch_loss = torch_loss_fn(torch_out, Yt)
torch_loss.backward(retain_graph=True)
torch_dLdW = net.weight.grad.data
torch_dLdb = net.bias.grad.data

print('MyTorch dLdW:\n', my_dLdW, '\n')
print('PyTorch dLdW:\n', torch_dLdW.T, '\n')
print('MyTorch dLdb:\n', my_dLdb, '\n')
print('PyTorch dLdb:\n', torch_dLdb, '\n')

print('Difference in dLdW:', np.linalg.norm(my_dLdW.T - torch_dLdW.data.numpy()))
print('Difference in dLdb:', np.linalg.norm(my_dLdb.flatten() - torch_dLdb.data.num
```

```
MyTorch dLdW:
 [[ 0.12012567  0.00295367  0.04191126 -0.07267667  0.02791693]
 [ 0.03518954 -0.0200141   0.03406723 -0.02431096  0.02070968]
 [ 0.14748257 -0.01532951  0.08382155 -0.01930616 -0.0203474 ]
 [ 0.06681007 -0.03891314  0.08080953  0.01504673  0.00045012]
 [ 0.11256331 -0.01973837  0.01680606 -0.00138297  0.00995075]
 [ 0.01978376 -0.02740623  0.02936517  0.03265669 -0.00413681]
 [ 0.13485436 -0.00176794  0.03313958 -0.00091217 -0.01160413]
 [ 0.02693809 -0.02455704  0.0399628   0.00525783  0.00168999]
 [ 0.14058303 -0.02630857  0.0610937  -0.01097525  0.00104313]
 [ 0.06542919 -0.01061266 -0.00446366  0.03434501 -0.00295966]]

PyTorch dLdW:
 tensor([[ 0.1201,  0.0030,  0.0419, -0.0727,  0.0279],
        [ 0.0352, -0.0200,  0.0341, -0.0243,  0.0207],
        [ 0.1475, -0.0153,  0.0838, -0.0193, -0.0203],
        [ 0.0668, -0.0389,  0.0808,  0.0150,  0.0005],
        [ 0.1126, -0.0197,  0.0168, -0.0014,  0.0100],
        [ 0.0198, -0.0274,  0.0294,  0.0327, -0.0041],
        [ 0.1349, -0.0018,  0.0331, -0.0009, -0.0116],
        [ 0.0269, -0.0246,  0.0400,  0.0053,  0.0017],
        [ 0.1406, -0.0263,  0.0611, -0.0110,  0.0010],
        [ 0.0654, -0.0106, -0.0045,  0.0343, -0.0030]])

MyTorch dLdb:
 [ 0.17004118 -0.02935796  0.06321571 -0.02123358  0.01974512]

PyTorch dLdb:
 tensor([ 0.1700, -0.0294,  0.0632, -0.0212,  0.0197])

Difference in dLdW: 1.1085912244326638e-07
Difference in dLdb: 6.319506077808191e-08
```

Compare a single optimization step

In [ ]:
```python
# my SGD step
my_optimizer = mytorch.optim.SGD(my_net, lr=0.1)
my_optimizer.step()
my_Wk = my_net.W
my_bk = my_net.b

# torch SGD step
optimizer.zero_grad()
torch_loss.backward(retain_graph=True)
optimizer.step()
torch_Wk = net.weight.data
torch_bk = net.bias.data

print('MyTorch Wk:\n', my_Wk, '\n')
print('PyTorch Wk:\n', torch_Wk.T, '\n')
print('MyTorch bk:\n', my_bk, '\n')
print('PyTorch bk:\n', torch_bk)
```

```
print('Difference in Wk:', np.linalg.norm(my_Wk - torch_Wk.data.numpy().T))
print('Difference in bk:', np.linalg.norm(my_bk.flatten() - torch_bk.data.numpy()))
```

```
MyTorch Wk:
 [[0.00225624 0.65454369 0.22786553 0.20375799 0.41525752]
 [0.06995093 0.09965917 0.54460619 0.32926168 0.65353759]
 [0.69258879 0.6152178  0.48305614 0.45607477 0.16900205]
 [0.05864521 0.35032098 0.24489474 0.8639804  0.83109932]
 [0.7373715  0.07190249 0.15495887 0.52788239 0.4534458 ]
 [0.14922424 0.40519783 0.92280482 0.5548514  0.28792973]
 [0.09215936 0.45045506 0.78606601 0.93621087 0.64919466]
 [0.28578957 0.23344379 0.51530289 0.79663504 0.54019491]
 [0.22328781 0.61469977 0.93714608 0.01456096 0.10773379]
 [0.06160243 0.89891648 0.83156413 0.43773651 0.74866049]]

PyTorch Wk:
 tensor([[0.0023, 0.6545, 0.2279, 0.2038, 0.4153],
        [0.0700, 0.0997, 0.5446, 0.3293, 0.6535],
        [0.6926, 0.6152, 0.4831, 0.4561, 0.1690],
        [0.0586, 0.3503, 0.2449, 0.8640, 0.8311],
        [0.7374, 0.0719, 0.1550, 0.5279, 0.4534],
        [0.1492, 0.4052, 0.9228, 0.5549, 0.2879],
        [0.0922, 0.4505, 0.7861, 0.9362, 0.6492],
        [0.2858, 0.2334, 0.5153, 0.7966, 0.5402],
        [0.2233, 0.6147, 0.9371, 0.0146, 0.1077],
        [0.0616, 0.8989, 0.8316, 0.4377, 0.7487]])

MyTorch bk:
 [[0.61248868 0.96033306 0.09868591 0.80567634 0.61423769]]

PyTorch bk:
 tensor([0.6125, 0.9603, 0.0987, 0.8057, 0.6142])
Difference in Wk: 1.037580056856067e-07
Difference in bk: 2.613949504884645e-08
```

In [ ]:

Problem 2 `loss.py` (2 pts each)

Your next task is to implement both the mean squared error (MSE) and cross entropy losses as their own classes in `loss.py`. Defining these classes allows us to treat losses as objects, which allows for storing and passing gradients.

(a) Implement the `forward` method for the `MSELoss` class. This method takes as inputs the output matrix $O \in \mathbb{R}^{N\times q}$ and returns the MSE with respect to the true targets $Y \in \mathbb{R}^{N\times q}$. Note that the MSE is normalized by dividing by Nq rather than just N.

(b) Implement the `backward` method for the `MSELoss` class. This method takes no input arguments and uses the stored variables O and Y to compute and return $\partial L/\partial O$.

(c) Implement the `forward` method for the `CrossEntropyLoss` class. This method takes as inputs the output matrix $O \in \mathbb{R}^{N\times q}$ and returns the MSE with respect to the true targets $Y \in \mathbb{R}^{N\times q}$. Note that the cross entropy loss is normalized by dividing by N only.

(d) Implement the `backward` method for the `CrossEntropyLoss` class. This method takes no input arguments and uses the stored variables O and Y to compute and return $\partial L/\partial O$.

Turn in the output of the notebook `LossTester.ipynb`.

Answer:

In [ ]:
```
%load_ext autoreload
%autoreload 2

import numpy as np
from mytorch import nn as mynn
from mytorch.optim import SGD
from torch import nn
import torch
from d2l import torch as d2l
```

MSE Testing

In [ ]:
```
# set up synthetic data
N = 10
num_inputs = 7
num_outputs = 3

# numpy/our versions
W = np.random.rand(num_inputs, num_outputs)
b = np.random.rand(num_outputs, 1)
X = np.random.randn(N, num_inputs)
Y = X @ W + np.outer(np.ones(N), b) + 0.5 * np.random.randn(N, num_outputs)

# converted torch versions
Xt = torch.tensor(X).float()
Wt = torch.tensor(W).float()
bt = torch.tensor(b).float()
Yt = torch.tensor(Y).float()
```

In [ ]:
```
# initialize model and fix weights to true values
my_net = mynn.Linear(num_inputs, num_outputs)
my_net.W = W
my_net.b = b

# initialize torch model, loss, optimizer
net = nn.Linear(num_inputs, num_outputs)
net.weight = nn.Parameter(Wt.T)
net.bias = nn.Parameter(bt[:, 0])
torch_out = net(Xt)
optimizer = torch.optim.SGD(net.parameters(), lr=0.1, momentum=0.0)
```

Test `forward()`

In [ ]:
```
# torch loss function
torch_mse_fn = nn.MSELoss()
torch_mse = torch_mse_fn(torch_out, Yt)

# mytorch loss function
my_mse_fn = mynn.MSELoss()
my_mse = my_mse_fn.forward(torch_out.detach().numpy(), Y)
```

```
print('Torch MSE:', torch_mse.data)
print('My MSE:', my_mse, '\n')
```

```
Torch MSE: tensor(0.2180)
My MSE: 0.21803364618155888
```

Test `backward()`

In [ ]:

```
# MSE
optimizer.zero_grad()
torch_out = net(Xt)
torch_mse = torch_mse_fn(torch_out, Yt)
torch_mse.backward(retain_graph=True)
torch_dLdW = net.weight.grad.data
torch_dLdb = net.bias.grad.data

dLdZ = my_mse_fn.backward()
my_net.forward(X)
my_net.backward(dLdZ)
my_dLdW = my_net.dLdW
my_dLdb = my_net.dLdb

print('MyTorch dLdW:\n', my_dLdW, '\n')
print('PyTorch dLdW:\n', torch_dLdW.T, '\n')
print('MyTorch dLdb:\n', my_dLdb, '\n')
print('PyTorch dLdb:\n', torch_dLdb, '\n')

print('Difference in dLdW:', np.linalg.norm(my_dLdW.T - torch_dLdW.data.numpy()))
print('Difference in dLdb:', np.linalg.norm(my_dLdb.flatten() - torch_dLdb.data.num
```

```
MyTorch dLdW:
 [[ 0.15629022 -0.01670823 -0.01395194]
 [ 0.02062998 -0.02817953 -0.14405865]
 [-0.06217844 -0.01217057  0.05367357]
 [ 0.09557618  0.01814454 -0.02136826]
 [ 0.03588844 -0.06888509 -0.01345359]
 [ 0.0222143  -0.0438693  -0.08048498]
 [ 0.12853376 -0.09517782  0.00418429]]

PyTorch dLdW:
 tensor([[ 0.1563, -0.0167, -0.0140],
        [ 0.0206, -0.0282, -0.1441],
        [-0.0622, -0.0122,  0.0537],
        [ 0.0956,  0.0181, -0.0214],
        [ 0.0359, -0.0689, -0.0135],
        [ 0.0222, -0.0439, -0.0805],
        [ 0.1285, -0.0952,  0.0042]])

MyTorch dLdb:
 [-0.03687998  0.13018967 -0.13604935]

PyTorch dLdb:
 tensor([-0.0369,  0.1302, -0.1360])

Difference in dLdW: 7.343191299264743e-08
Difference in dLdb: 1.4674534402403805e-08
```

CE Testing

In [ ]:

```
# set up synthetic data
N = 10
num_inputs = 7
num_outputs = 3

# numpy/our versions
W = np.random.rand(num_inputs, num_outputs)
b = np.random.rand(num_outputs, 1)
# generate random one-hot matrix
x = np.eye(num_outputs)
x[np.random.choice(x.shape[0], size=N)]
Y = np.eye(num_outputs)[np.random.choice(num_outputs, N)]

# converted torch versions
Xt = torch.tensor(X).float()
Wt = torch.tensor(W).float()
bt = torch.tensor(b).float()
Yt = torch.tensor(Y).float()
```

In [ ]:

```
# initialize model and fix weights to true values
my_net = mynn.Linear(num_inputs, num_outputs)
my_net.W = W
my_net.b = b

# initialize torch model, loss, optimizer
```

```
net = nn.Linear(num_inputs, num_outputs)
net.weight = nn.Parameter(Wt.T)
net.bias = nn.Parameter(bt[:, 0])
torch_out = net(Xt)
optimizer = torch.optim.SGD(net.parameters(), lr=0.1, momentum=0.0)
```

Test `forward()`

In [ ]:
```
# torch loss functions
torch_ce_fn = nn.CrossEntropyLoss()
torch_ce = torch_ce_fn(torch_out, Yt)

# mytorch loss functions
my_ce_fn = mynn.CrossEntropyLoss()
my_ce = my_ce_fn.forward(torch_out.detach().numpy(), Y)

print('Torch CE:', torch_ce.data)
print('My CE:', my_ce, '\n')
```

```
Torch CE: tensor(1.6076)
My CE: 1.4981077154179694
```

Test `backward()`

In [ ]:
```
optimizer.zero_grad()
torch_out = net(Xt)
torch_ce = torch_ce_fn(torch_out, Yt)
torch_ce.backward(retain_graph=True)
torch_dLdW = net.weight.grad.data
torch_dLdb = net.bias.grad.data

dLdZ = my_ce_fn.backward()
my_net.forward(X)
my_net.backward(dLdZ)
my_dLdW = my_net.dLdW
my_dLdb = my_net.dLdb

print('MyTorch dLdW:\n', my_dLdW, '\n')
print('PyTorch dLdW:\n', torch_dLdW.T, '\n')
print('MyTorch dLdb:\n', my_dLdb, '\n')
print('PyTorch dLdb:\n', torch_dLdb, '\n')

print('Difference in dLdW:', np.linalg.norm(my_dLdW.T - torch_dLdW.data.numpy()))
print('Difference in dLdb:', np.linalg.norm(my_dLdb.flatten() - torch_dLdb.data.num
```

```
MyTorch dLdW:
 [[-2.42025283 -0.74964799 -4.02664113]
 [ 1.271734   -2.08355641 -1.50988379]
 [-3.78258475  0.56260216 -0.22064712]
 [-0.12173315  0.01676124 -2.289436  ]
 [ 0.62239384 -0.81948011  0.28667337]
 [ 2.05611344  0.6249941   1.40096314]
 [ 0.54078871  0.71594616 -1.41667321]]

PyTorch dLdW:
 tensor([[ 0.1990, -0.1686, -0.0303],
        [-0.2816, -0.2838,  0.5654],
        [-0.0604,  0.1042, -0.0438],
        [ 0.0676,  0.1027, -0.1702],
        [ 0.0580, -0.1197,  0.0617],
        [-0.0905, -0.0410,  0.1315],
        [-0.0730, -0.3267,  0.3996]])

MyTorch dLdb:
 [-0.68258679 -1.88425163  1.19849675]

PyTorch dLdb:
 tensor([ 0.0650,  0.0972, -0.1622])

Difference in dLdW: 8.002632024295615
Difference in dLdb: 2.517237004940137
```

Problem 3 `sgd.py` (2 pts)

Now that we have the ability to pass gradients, we can use stochastic gradient descent (SGD) to optimize the parameters of our linear layer. Your task is to implement the `step` method, which updates both the weight matrix W and bias vector b. Note that the provided implementation allows for multiple layers, but the update equation is the same for either case.
Turn in the output of the notebook `SGDTester.ipynb`.

Answer:

In [ ]:
```
%load_ext autoreload
%autoreload 2

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

import mytorch
from mytorch import nn as mynn
```

In [ ]:
```
# load data for testing
data = np.load('testerData.npz')
W, b, X, Y, dLdZ = [data[fname] for fname in data.files]

[N, num_inputs] = X.shape
num_outputs = Y.shape[1]

# converted torch versions
Xt = torch.tensor(X).float()
Wt = torch.tensor(W).float()
bt = torch.tensor(b).float()
Yt = torch.tensor(Y).float()
```

In [ ]:
```
# initialize model and fix weights to true values
my_net = mynn.Linear(num_inputs, num_outputs)
my_net.W = W
my_net.b = b.flatten()

# initialize torch model, loss, optimizer
net = nn.Linear(num_inputs, num_outputs)
net.weight = nn.Parameter(Wt.T)
net.bias = nn.Parameter(bt[:, 0])
criterion = nn.MSELoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.1, momentum=0.0)
```

In [ ]:
```
# get gradients for both networks
my_out = my_net.forward(X)
my_net.backward(dLdZ)
my_dLdW = my_net.dLdW
my_dLdb = my_net.dLdb

torch_out = net(Xt)
optimizer.zero_grad()
torch_loss_fn = nn.MSELoss()
torch_loss = torch_loss_fn(torch_out, Yt)
torch_loss.backward(retain_graph=True)
```

Compare a single optimization step

In [ ]:
```
# my SGD step
my_optimizer = mytorch.optim.SGD(my_net, lr=0.1)
```

```
my_optimizer.step()
my_Wk = my_net.W
my_bk = my_net.b

# torch SGD step
optimizer.zero_grad()
torch_loss.backward(retain_graph=True)
optimizer.step()
torch_Wk = net.weight.data
torch_bk = net.bias.data

print('MyTorch Wk:\n', my_Wk, '\n')
print('PyTorch Wk:\n', torch_Wk.T, '\n')
print('MyTorch bk:\n', my_bk, '\n')
print('PyTorch bk:\n', torch_bk, '\n')

print('Difference in Wk:', np.linalg.norm(my_Wk - torch_Wk.data.numpy().T))
print('Difference in bk:', np.linalg.norm(my_bk.flatten() - torch_bk.data.numpy()))
```

```
MyTorch Wk:
 [[0.00225624 0.65454369 0.22786553 0.20375799 0.41525752]
 [0.06995093 0.09965917 0.54460619 0.32926168 0.65353759]
 [0.69258879 0.6152178  0.48305614 0.45607477 0.16900205]
 [0.05864521 0.35032098 0.24489474 0.8639804  0.83109932]
 [0.7373715  0.07190249 0.15495887 0.52788239 0.4534458 ]
 [0.14922424 0.40519783 0.92280482 0.5548514  0.28792973]
 [0.09215936 0.45045506 0.78606601 0.93621087 0.64919466]
 [0.28578957 0.23344379 0.51530289 0.79663504 0.54019491]
 [0.22328781 0.61469977 0.93714608 0.01456096 0.10773379]
 [0.06160243 0.89891648 0.83156413 0.43773651 0.74866049]]

PyTorch Wk:
 tensor([[0.0023, 0.6545, 0.2279, 0.2038, 0.4153],
        [0.0700, 0.0997, 0.5446, 0.3293, 0.6535],
        [0.6926, 0.6152, 0.4831, 0.4561, 0.1690],
        [0.0586, 0.3503, 0.2449, 0.8640, 0.8311],
        [0.7374, 0.0719, 0.1550, 0.5279, 0.4534],
        [0.1492, 0.4052, 0.9228, 0.5549, 0.2879],
        [0.0922, 0.4505, 0.7861, 0.9362, 0.6492],
        [0.2858, 0.2334, 0.5153, 0.7966, 0.5402],
        [0.2233, 0.6147, 0.9371, 0.0146, 0.1077],
        [0.0616, 0.8989, 0.8316, 0.4377, 0.7487]])

MyTorch bk:
 [[0.61248868 0.96033306 0.09868591 0.80567634 0.61423769]]

PyTorch bk:
 tensor([0.6125, 0.9603, 0.0987, 0.8057, 0.6142])

Difference in Wk: 1.037580056856067e-07
Difference in bk: 2.613949504884645e-08
```

Problem 4 `activation.py` (2 pts each)

At this point, we have implemented a single-layer neural network that can be used for either classification or regression. You can (and probably should) test your network using the notebook `TrainingTester.ipynb` to see how it performs on these tasks. To allow our network to learn nonlinear functions, we need to utilize activation functions, which are implemented in `activation.py`.

Consider a multi-layer perceptrion (MLP) with a single hidden layer. Let $H \in \mathbb{R}^{N \times h}$ be the output of hidden variables after the first linear layer. For an activation function $\phi : \mathbb{R} \to \mathbb{R}$, let $A \in \mathbb{R}^{N \times h}$ be the output after applying ϕ element-wise to H.

(a) Implement the sigmoid activation function. Complete the `forward` method, which takes the hidden variables H as input and outputs the the activation function applied element-wise to H. Next, complete the `backward` method, which utilizes the stored variable H to compute the element-wise gradient of the activation function with respect to H. Note that the gradient $\partial A / \partial H$ should have the same dimensions as H.

(b) Implement the tanh activation function, completing both the `forward` and `backward` methods.

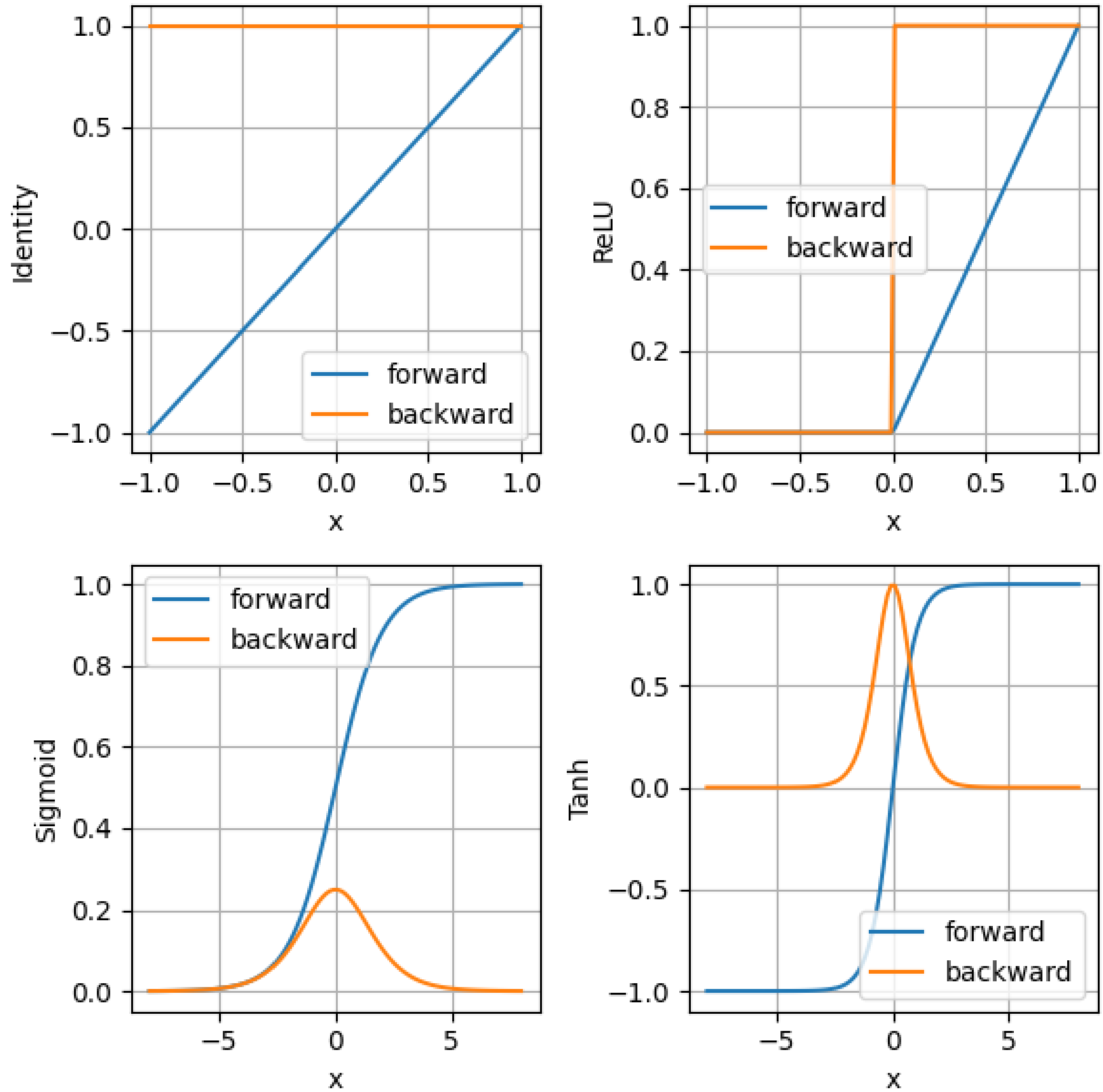
(c) Implement the ReLU activation function, completing both the `forward` and `backward` methods.

Turn in the output of the notebook `ActivationTester.ipynb`.

Answer:

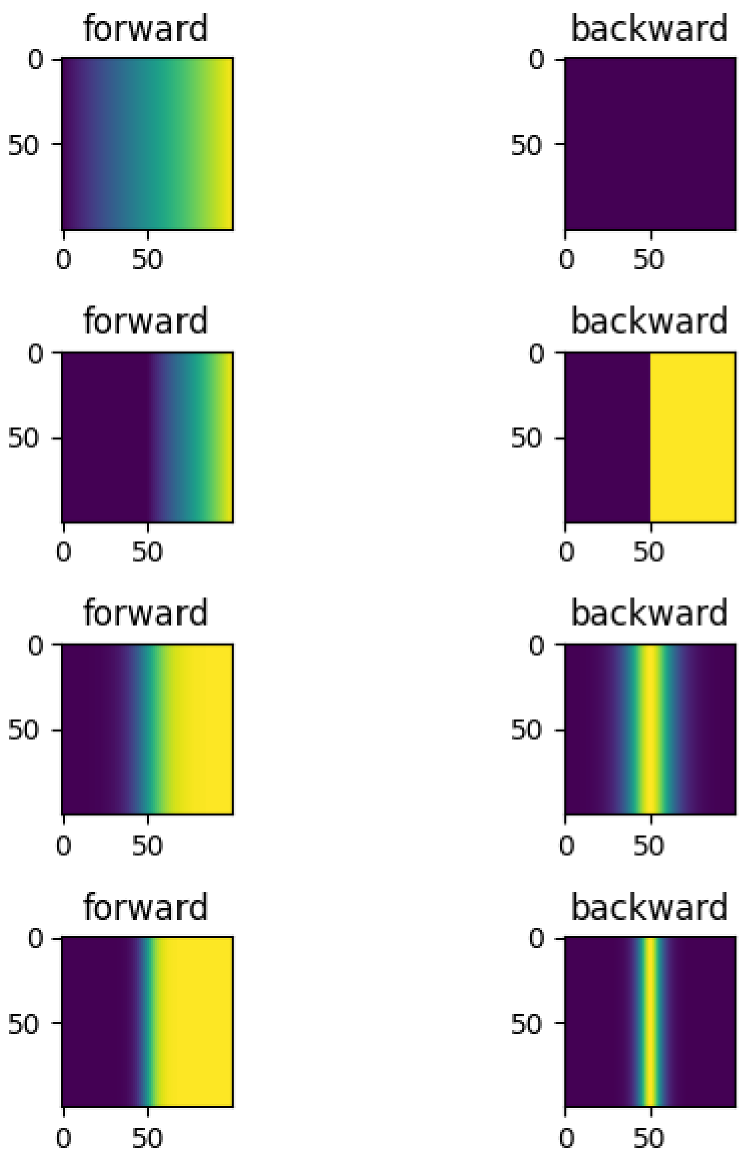
In [ ]:
```
%load_ext autoreload
%autoreload 2

import numpy as np
import matplotlib.pyplot as plt
from torch import nn
import torch
from d2l import torch as d2l

from mytorch import nn as mynn
from mytorch.optim import SGD
```

In [ ]:
```
# test activation forward/backward in one dimension
fig, ax = plt.subplots(2, 2, figsize=(6, 6))

X = np.linspace(-1, 1, 100)
act = mynn.Identity()
ax[0, 0].plot(X, act.forward(X), label='forward')
ax[0, 0].plot(X, act.backward(), label='backward')
ax[0, 0].grid()
ax[0, 0].set_xlabel('x')
ax[0, 0].set_ylabel('Identity')
ax[0, 0].legend()

act = mynn.ReLU()
ax[0, 1].plot(X, act.forward(X), label='forward')
ax[0, 1].plot(X, act.backward(), label='backward')
ax[0, 1].grid()
ax[0, 1].set_xlabel('x')
ax[0, 1].set_ylabel('ReLU')
ax[0, 1].legend()

X = np.linspace(-8, 8, 100)
act = mynn.Sigmoid()
ax[1, 0].plot(X, act.forward(X), label='forward')
ax[1, 0].plot(X, act.backward(), label='backward')
ax[1, 0].grid()
ax[1, 0].set_xlabel('x')
ax[1, 0].set_ylabel('Sigmoid')
ax[1, 0].legend()

act = mynn.Tanh()
ax[1, 1].plot(X, act.forward(X), label='forward')
ax[1, 1].plot(X, act.backward(), label='backward')
ax[1, 1].grid()
ax[1, 1].set_xlabel('x')
ax[1, 1].set_ylabel('Tanh')
ax[1, 1].legend()

fig.tight_layout()
```




```
In [ ]: # test activation forward/backward in two dimensions
fig, ax = plt.subplots(4, 2, figsize=(6, 6))

X = np.outer(np.ones(100), np.linspace(-1, 1, 100))
act = mynn.Identity()
ax[0, 0].imshow(act.forward(X))
ax[0, 1].imshow(act.backward())
ax[0, 0].set_title('forward')
ax[0, 1].set_title('backward')

act = mynn.ReLU()
ax[1, 0].imshow(act.forward(X))
ax[1, 1].imshow(act.backward())
ax[1, 0].set_title('forward')
ax[1, 1].set_title('backward')

X = np.outer(np.ones(100), np.linspace(-8, 8, 100))
act = mynn.Sigmoid()
ax[2, 0].imshow(act.forward(X))
ax[2, 1].imshow(act.backward())
```

```
ax[2, 0].set_title('forward')
ax[2, 1].set_title('backward')

act = mynn.Tanh()
ax[3, 0].imshow(act.forward(X))
ax[3, 1].imshow(act.backward())
ax[3, 0].set_title('forward')
ax[3, 1].set_title('backward')

fig.tight_layout()
```



In [ ]:

Problem 5 `mlp.py` (4 pts each)

Having put into place the tools for optimizing a linear layer, selecting the appropriate cost function, and adding nonlinearities, we are now ready to implement a MLP with an arbitrary number of layers. In this problem, we will complete the `mlp.py` file, implementing MLPs with 0, 1, and 4 hidden layers.

(a) Complete the class `MLP0` to implement an MLP with 0 hidden layers. The class structure is as follows.

- `__init__`: Defines the list of layers in `self.layers` and activation functions in `self.f`. For the MLP0, we have one linear layer and a single Identity activation function.
- `forward`: Performs a forward pass, first through the linear layer, then through the activation function. The parameters follow the equations

$$H^{(1)} = XW^{(1)} + 1_N(b^{(1)})^T$$
$$A^{(1)} = \phi\left(H^{(1)}\right) \text{ (element wise)}$$

- `backward`: Performs a backward pass through the network. This method takes the gradient $\partial L/\partial A^{(1)}$ as input, then computes $\partial A^{(1)}/\partial H^{(1)}$ via the `backward` method for the corresponding activation function. It then computes

$$\frac{\partial L}{\partial H^{(1)}} = \frac{\partial L}{\partial A^{(1)}} \odot \frac{\partial A^{(1)}}{\partial H^{(1)}}.$$

  Noting that $H^{(1)}$ is the output of our single linear layer, it uses $\partial L/\partial H^{(1)}$ to compute the gradients of the loss with respect to $W^{(1)}$, $b^{(1)}$, and the inputs X. These gradients can be used with SGD or passed to additional layers, as in parts (b) and (c) of this problem.

(b) Complete the class `MLP1` to implement an MLP with 1 hidden layer. In this case, our forward equations become

$$H^{(1)} = XW^{(1)} + 1_N(b^{(1)})^T$$
$$A^{(1)} = \phi\left(H^{(1)}\right) \text{ (element wise)}$$
$$H^{(2)} = A^{(1)}W^{(2)} + 1_N(b^{(2)})^T$$
$$A^{(2)} = \phi\left(H^{(2)}\right) \text{ (element wise).}$$

(c) Follow the above sequence to complete the class `MLP4`.

Turn in the output of the notebook `MLPTester.ipynb`.

Answer:

In [ ]:
```
%load_ext autoreload
%autoreload 2

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

import mytorch
from mytorch import nn as mynn
from models import MLP0, MLP1
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

In [ ]:
```
# set up synthetic data
N = 10
num_inputs = 7
num_outputs = 2

# numpy/our versions
W = np.random.rand(num_inputs, num_outputs)
b = np.random.rand(num_outputs)
X = np.random.randn(N, num_inputs)
Y = X @ W + np.outer(np.ones(N), b) + 0.5 * np.random.randn(N, num_outputs)

# converted torch versions
Xt = torch.tensor(X).float()
Wt = torch.tensor(W).float()
bt = torch.tensor(b).float()
Yt = torch.tensor(Y).float()
```

MLP0

Test `forward()`

In [ ]:
```
# initialize model and fix weights to true values
mlp0 = MLP0(num_inputs, num_outputs)
mlp0.layers[0].W = W
mlp0.layers[0].b = b

# initialize torch model, loss, optimizer
net = nn.Sequential(nn.Linear(num_inputs, num_outputs))
net[0].weight = nn.Parameter(Wt.T)
net[0].bias = nn.Parameter(bt)
optimizer = torch.optim.SGD(net.parameters(), lr=1, momentum=0.0)

my_out = mlp0.forward(X)
torch_out = net(Xt)

print('MyTorch:\n', my_out, '\n')
```

```
print('PyTorch:\n', torch_out.data, '\n')
print('Difference:', np.linalg.norm(my_out - torch_out.data.numpy()))
```

```
MyTorch:
 [[ 3.86002782  2.99438936]
 [-0.79696783 -0.2794177 ]
 [-0.16700786  0.08508616]
 [-1.68222523 -0.6309244 ]
 [-0.82226997 -0.4766995 ]
 [ 3.64463101  1.33491015]
 [-2.59587695 -0.72561625]
 [-1.96711607 -1.04104274]
 [-1.24519469 -1.12006223]
 [ 0.83728668  1.70837926]]

PyTorch:
 tensor([[ 3.8600,  2.9944],
        [-0.7970, -0.2794],
        [-0.1670,  0.0851],
        [-1.6822, -0.6309],
        [-0.8223, -0.4767],
        [ 3.6446,  1.3349],
        [-2.5959, -0.7256],
        [-1.9671, -1.0410],
        [-1.2452, -1.1201],
        [ 0.8373,  1.7084]])

Difference: 3.7775427099892897e-07
```

Test `backward()`

In [ ]:

```
my_mse_fn = mynn.MSELoss()
my_mse = my_mse_fn.forward(my_out, Y)
dLdZ = my_mse_fn.backward()
mlp0.backward(dLdZ)
my_dLdW = mlp0.layers[0].dLdW
my_dLdb = mlp0.layers[0].dLdb

optimizer.zero_grad()
torch_loss_fn = nn.MSELoss()
torch_loss = torch_loss_fn(torch_out, Yt)
torch_loss.backward(retain_graph=True)
torch_dLdW = net[0].weight.grad.data
torch_dLdb = net[0].bias.grad.data

print('MyTorch dLdW:\n', my_dLdW, '\n')
print('PyTorch dLdW:\n', torch_dLdW.T, '\n')
print('MyTorch dLdb:\n', my_dLdb, '\n')
print('PyTorch dLdb:\n', torch_dLdb, '\n')

print('Difference in dLdW:', np.linalg.norm(my_dLdW.T - torch_dLdW.numpy()))
print('Difference in dLdb:', np.linalg.norm(my_dLdb.flatten() - torch_dLdb.n
```

```
MyTorch dLdW:
 [[-0.21218814  0.05111207]
 [-0.08868613  0.02084339]
 [ 0.078828    0.17021322]
 [ 0.06374441 -0.16389866]
 [ 0.17499219 -0.0323467 ]
 [ 0.21739307 -0.07383894]
 [-0.08884962 -0.10607492]]

PyTorch dLdW:
 tensor([[-0.2122,  0.0511],
        [-0.0887,  0.0208],
        [ 0.0788,  0.1702],
        [ 0.0637, -0.1639],
        [ 0.1750, -0.0323],
        [ 0.2174, -0.0738],
        [-0.0888, -0.1061]])

MyTorch dLdb:
 [-0.34788953  0.21548112]

PyTorch dLdb:
 tensor([-0.3479,  0.2155])

Difference in dLdW: 1.5759888413168285e-07
Difference in dLdb: 1.6025198587496917e-08
```

Test a single optimization step

In [ ]:
```
# my SGD step
my_optimizer = mytorch.optim.SGD(mlp0, lr=1)
my_optimizer.step()
my_Wk = mlp0.layers[0].W
my_bk = mlp0.layers[0].b

# torch SGD step
optimizer.zero_grad()
torch_loss.backward(retain_graph=True)
optimizer.step()
torch_Wk = net[0].weight.data
torch_bk = net[0].bias.data

print('MyTorch Wk:\n', my_Wk, '\n')
print('PyTorch Wk:\n', torch_Wk.T, '\n')
print('MyTorch bk:\n', my_bk, '\n')
print('PyTorch bk:\n', torch_bk)

print('Difference in Wk:', np.linalg.norm(my_Wk.T - torch_Wk.numpy()))
print('Difference in bk:', np.linalg.norm(my_bk.flatten() - torch_bk.numpy()
```

```
MyTorch Wk:
 [[0.64096616 0.2778433 ]
 [0.61522442 0.40386958]
 [0.29952571 0.16312084]
 [0.88400252 0.59914513]
 [0.28828544 0.71377332]
 [0.58806773 0.79375926]
 [0.69368298 0.18085436]]

PyTorch Wk:
 tensor([[0.6410, 0.2778],
        [0.6152, 0.4039],
        [0.2995, 0.1631],
        [0.8840, 0.5991],
        [0.2883, 0.7138],
        [0.5881, 0.7938],
        [0.6937, 0.1809]])

MyTorch bk:
 [[0.69026393 0.39344826]]

PyTorch bk:
 tensor([0.6903, 0.3934])
Difference in Wk: 1.8491735404934512e-07
Difference in bk: 6.96111754757384e-08
```

MLP1

Test `forward()`

In [ ]:

```
num_hiddens=3

# initialize torch model, loss, optimizer
net = nn.Sequential(nn.Linear(num_inputs, num_hiddens),
                    nn.ReLU(),
                    nn.Linear(num_hiddens, num_outputs),
                    nn.Identity())
optimizer = torch.optim.SGD(net.parameters(), lr=0.1, momentum=0.0)

# initialize my network using torch W, b for each layer
W0 = net[0].weight.detach().numpy().T
b0 = net[0].bias.detach().numpy().T
W1 = net[2].weight.detach().numpy().T
b1 = net[2].bias.detach().numpy().T

mlp1 = MLP1(num_inputs, num_outputs, num_hiddens)
mlp1.layers[0].W = W0
mlp1.layers[0].b = b0
mlp1.layers[1].W = W1
mlp1.layers[1].b = b1

my_out = mlp1.forward(X)
```

```
torch_out = net(Xt)

print('MyTorch:\n', my_out, '\n')
print('PyTorch:\n', torch_out.data, '\n')
print('Difference:', np.linalg.norm(my_out - torch_out.data.numpy()))
```

```
MyTorch:
 [[-0.43635382  0.38668833]
 [-0.54950023  0.3628647 ]
 [-0.54950023  0.3628647 ]
 [-0.51382037  0.21551731]
 [-0.50974625  0.37123513]
 [-0.54950023  0.3628647 ]
 [-0.38876143  0.29533438]
 [-0.56818172  0.27561022]
 [-0.52001079  0.3601708 ]
 [-0.61013367  0.07966791]]

PyTorch:
 tensor([[-0.4364,  0.3867],
        [-0.5495,  0.3629],
        [-0.5495,  0.3629],
        [-0.5138,  0.2155],
        [-0.5097,  0.3712],
        [-0.5495,  0.3629],
        [-0.3888,  0.2953],
        [-0.5682,  0.2756],
        [-0.5200,  0.3602],
        [-0.6101,  0.0797]])

Difference: 7.751443391709816e-08
```

Test `backward()`

In [ ]:
```
my_mse_fn = mynn.MSELoss()
my_mse = my_mse_fn.forward(my_out, Y)
dLdZ = my_mse_fn.backward()
mlp1.backward(dLdZ)
my_dLdW0 = mlp1.layers[0].dLdW.T
my_dLdb0 = mlp1.layers[0].dLdb
my_dLdW1 = mlp1.layers[1].dLdW.T
my_dLdb1 = mlp1.layers[1].dLdb

optimizer.zero_grad()
torch_loss_fn = nn.MSELoss()
torch_loss = torch_loss_fn(torch_out, Yt)
torch_loss.backward(retain_graph=True)
torch_dLdW0 = net[0].weight.grad.data
torch_dLdb0 = net[0].bias.grad.data
torch_dLdW1 = net[2].weight.grad.data
torch_dLdb1 = net[2].bias.grad.data

print('Difference in dLdW0:', np.linalg.norm(my_dLdW0 - torch_dLdW0.data.num
print('Difference in dLdb0:', np.linalg.norm(my_dLdb0.flatten() - torch_dLdb
```

```
print('Difference in dLdW1:', np.linalg.norm(my_dLdW1 - torch_dLdW1.data.num
print('Difference in dLdb1:', np.linalg.norm(my_dLdb1.flatten() - torch_dLdb
```

```
Difference in dLdW0: 5.073911425469867e-08
Difference in dLdb0: 4.622361428754544e-09
Difference in dLdW1: 3.821583479869518e-08
Difference in dLdb1: 2.3672265170171044e-08
```

Test a single optimization step

In [ ]:

```
# my SGD step
my_optimizer = mytorch.optim.SGD(mlp1, lr=1)
my_optimizer.step()
my_Wk0 = mlp1.layers[0].W
my_bk0 = mlp1.layers[0].b
my_Wk1 = mlp1.layers[1].W
my_bk1 = mlp1.layers[1].b

# torch SGD step
optimizer.zero_grad()
torch_loss.backward(retain_graph=True)
optimizer.step()
torch_Wk0 = net[0].weight.data
torch_bk0 = net[0].bias.data
torch_Wk1 = net[2].weight.data
torch_bk1 = net[2].bias.data

print('Difference in Wk0:', np.linalg.norm(my_Wk0 - torch_Wk0.numpy().T))
print('Difference in bk0:', np.linalg.norm(my_bk0.flatten() - torch_bk0.nump
print('Difference in Wk1:', np.linalg.norm(my_Wk1 - torch_Wk1.numpy().T))
print('Difference in bk1:', np.linalg.norm(my_bk1.flatten() - torch_bk1.nump
```

```
Difference in Wk0: 0.4484992422899716
Difference in bk0: 0.11698040828986507
Difference in Wk1: 0.0828025291773284
Difference in bk1: 0.7600495461767015
```

Problem 6 `TrainingTester.ipynb` (0 pts)

Congratulations! If you've reached this point, you have coded a pure-`numpy`, object-oriented implementation of an MLP with arbitrary number of layers and activation functions, capable exciting tasks such as regressing synthetic data and classifying pictures of clothing! In `TrainingTester.ipynb`, I have implemented the framework to allow for training neural networks with MyTorch using tools from the D2L library. Have a look at the code in this notebook to observe what changes were required to utilize our library, as well as how to perform the same tasks using PyTorch.

Turn in nothing.

All code:

Linear Code:

```python
class Linear:

    def __init__(self, num_inputs, num_outputs):
        """
        Initialize the weights to be zero-mean Gaussian with
        variance 0.01 and biases to zero.

        :param num_inputs: Number of inputs to layer.
        :param num_outputs: Number of outputs after layer.
        """

        #intialize W using a normal distribution with variance 0.01 and mean 0
        self.W = np.random.normal(0, 0.01, (num_inputs, num_outputs))

        #initialize b to be a vector of zeros
        self.b = np.zeros((num_outputs, 1))



    def forward(self, A):
        """
        Forward operation of linear layer. Performs
        operation O = AW + b. Stores input to object.

        :param A: Input data matrix with rows as examples.
        :return O: Output data matrix after affine transformation.
        """
        self.A = A
        self.N = A.shape[0]
        ones = np.ones((1,self.N))
        # self.b = np.atleast_2d(self.b)
        self.b = np.reshape(np.atleast_2d(self.b), (1, -1))
        pt1 = ones.T @ np.atleast_2d(self.b)
        pt2 = self.A @ self.W
        O = pt1 + pt2
        return O

    def backward(self, dLdO):
        """
        Backpropagation operation for variables in linear
        layer. Stores derivatives dLdW, dLdb and returns dLdA.

        :param dLdO: Derivative of loss with respect to output.
        Obtained from backward operation on loss object.
        :returns dLdA: Derivative of loss with respect to input.
        """
        dOdW = self.A
        dOdb = np.ones((self.N, 1))
        dOdA = self.W
        dLdW = dLdO.T @ dOdW
        dLdb = dLdO.T @ dOdb
        dLdA = dLdO @ dOdA.T
        self.dLdW = dLdW.T
        self.dLdb = dLdb.flatten()
        return dLdA
```

Loss Code:

```python
class MSELoss:

    def forward(self, O, Y):
        """
        Compute MSE loss between outputs O and true targets Y.

        :param O: Output predictions.
        :param Y: True targets.
        :return L: Mean squared error, normalized by total number
        of elements in O.
        """
        self.O = O
        self.Y = Y
        self.N = O.shape[0]
        # Compute mean squared error, then normalize by number of elements in O.
        #L = None was original
        L = np.sum((O - Y)**2) / (self.N * O.shape[1])
        return L

    def backward(self):
        """
        Compute gradient dLdO for MSE loss.

        :return dLdO: Gradient of loss with respect to output O.
        """
        O = self.O
        Y = self.Y
        dLdO = 2 * (O - Y) / (self.N * O.shape[1])
        return dLdO

class CrossEntropyLoss:

    def forward(self, O, Y):
        """
        Compute cross entropy loss between outputs O and true targets Y
        as well as softmax probabilities for outputs O.
        Note: Does not match PyTorch unless Y is a one-hot label matrix.

        :param O: Output predictions.
        :param Y: True targets.
        :return L: Cross entropy loss, normalized by number of examples.
        """
        self.O = O
        self.Y = Y
        self.N = O.shape[0]
        #first do softmax of predicted and targets
        O = np.exp(O) / np.sum(np.exp(O), axis=1, keepdims=True)
        Y = np.exp(Y) / np.sum(np.exp(Y), axis=1, keepdims=True)
        #compute MSE with rspect to true targets. Normalize by dividing by N only.
        L = (-np.sum(Y * np.log(O))) / self.N
        return L

    def backward(self):
        """
        Compute gradient dLdO for cross entropy loss.

        :return dLdO: Gradient of loss with respect to output O.
        """
        dLdO = -self.Y / self.O
        return dLdO
```

SGD Code:

```
class SGD:

    def __init__(self, model, lr=0.1):
        """
        Initialize SGD object.

        :param model: Neural network object from mytorch.nn.
        :param lr: Learning rate.
        """
        self.model = model
        self.lr = lr
        # for use with MLP, which has multiple layers
        if hasattr(model, "layers"):
            self.l = model.layers
            self.L = len(model.layers)

    def step(self):
        """
        Perform a single SGD step.
        """
        if hasattr(self.model, "layers"):
            for i in range(self.L):
                dLdW = self.l[i].dLdW
                dLdb = self.l[i].dLdb
                #perform step for W and b
                self.l[i].W = self.l[i].W - self.lr * dLdW
                self.l[i].b = self.l[i].b - self.lr * dLdb

        else:
            dLdW = self.model.dLdW
            dLdb = self.model.dLdb
            #perform step for W and b (same as prior, but for single layer)
            self.model.W = self.model.W - self.lr * dLdW
            self.model.b = self.model.b - self.lr * dLdb

    def zero_grad(self):
        """
        Dummy function for use with d2l library.
        """
        return
```

Activation Code:

```python
class Identity:

    def forward(self, H):
        """
        Compute identity activation function.

        :param H: Output from hidden or final layer.
        :return A: Output after applying activation function.
        """
        self.H = H
        return self.H

    def backward(self):
        """
        Compute derivative of identity activation function.

        :return dAdH: Element-wise derivative with respect to
        input to activation function H.
        """
        dAdH = np.ones(self.H.shape, dtype="f")
        return dAdH


class Sigmoid:

    def forward(self, H):
        """
        Compute sigmoid activation function.

        :param H: Output from hidden or final layer.
        :return A: Output after applying activation function.
        """
        self.H = H
        def sig(x):
            return 1/(1 + np.exp(-x))
        # sigfunc = np.vectorize(sig)
        newH = sig(self.H)
        return newH

    def backward(self):
        """
        Compute derivative of identity activation function.

        :return dAdH: Element-wise derivative with respect to
        input to activation function H.
        """
        def sig(x):
            f = 1/(1 + np.exp(-x))
            return f * (1 - f)
        # sigfunc = np.vectorize(sig)
        newH = sig(self.H)
        return newH
```

```python
class Tanh:

    def forward(self, H):
        """
        Compute tanh activation function.

        :param H: Output from hidden or final layer.
        :return A: Output after applying activation function.
        """
        self.H = H
        def tanh(x):
            t = (np.exp(x) - np.exp(-x)) / (np.exp(x) + np.exp(-x))
            return t
        # tanhfunc = np.vectorize(tanh)
        newH = tanh(self.H)
        return newH

    def backward(self):
        """
        Compute derivative of identity activation function.

        :return dAdH: Element-wise derivative with respect to
        input to activation function H.
        """
        def tanh(x):
            t = (np.exp(x) - np.exp(-x)) / (np.exp(x) + np.exp(-x))
            return 1 - t**2

        # tanhfunc = np.vectorize(tanh)
        newH = tanh(self.H)
        return newH

class ReLU:

    def forward(self, H):
        """
        Compute tanh activation function.

        :param H: Output from hidden or final layer.
        :return A: Output after applying activation function.
        """
        self.H = H
        def relu(x):
            return max(0.0, x)
        relufunc = np.vectorize(relu)
        newH = relufunc(self.H)
        return newH

    def backward(self):
        """
        Compute derivative of identity activation function.

        :return dAdH: Element-wise derivative with respect to
        input to activation function H.
        """
        def relu(x):
            if x > 0:
                return 1
            else:
                return 0
        relufunc = np.vectorize(relu)
        newH = relufunc(self.H)
        return newH
```

MLP Code:

```python
class MLP0:

    def __init__(self, num_inputs, num_outputs):
        """
        Initialize MLP object with a single linear layer
        followed by an identity activation function.

        :param num_inputs: Number of inputs to layer.
        :param num_outputs: Number of outputs after layer.
        """
        #create self.layers which is a single linear layer
        self.layers = [Linear(num_inputs, num_outputs)]
        #set self.f to be of size num_outputs
        self.f = [Identity()]

    def forward(self, X):
        """
        Forward operation of MLP with zero hidden layers.

        :param X: Input data matrix with rows as examples.
        :return A1: Output data matrix after affine transformation
        and activation function.
        """
        #perform forward pass through the network. Compute H1
        H1 = self.layers[0].forward(X)
        #then compute A1, which is rho(H1) elementwise
        A1 = self.f[0].forward(H1)

        return A1

    def backward(self, dLdA1):
        """
        Backpropagation operation for MLP with zero hidden layers.
        Performs backpropagation on appropriate layers to obtain
        gradient with respect to the input X.
        Does not return anything.

        :param dLdA1: Derivative of loss with respect to output A1.
        Obtained from backward operation on loss object.
        """
        #perform backward pass through the network. Compute dA1dH1. Note backward() takes no arguments
        dA1dH1 = self.f[0].backward()
        #then compute dLdH1
        dLdH1 = dLdA1 * dA1dH1
        #then compute dLdX
        dLdX = self.layers[0].backward(dLdH1)
```

```python
class MLP1:

    def __init__(self, num_inputs, num_outputs, num_hiddens):
        """
        Initialize MLP object with a single hidden layer
        followed by a ReLU activation function. Use and Identity
        activation function at the output.

        :param num_inputs: Number of inputs to model.
        :param num_outputs: Number of outputs from model.
        :param num_hiddens: Size of hidden layer.
        """
        self.layers = [Linear(num_inputs, num_hiddens), Linear(num_hiddens, num_outputs)]
        self.f = [ReLU(), Identity()]

    def forward(self, X):
        """
        Forward operation of MLP with one hidden layer.

        :param X: Input data matrix with rows as examples.
        :return A2: Output data matrix.
        """
        #perform forward pass through the network. Compute H1
        H1 = self.layers[0].forward(X)
        #then compute A1, which is rho(H1) elementwise
        A1 = self.f[0].forward(H1)
        #then compute H2
        H2 = self.layers[1].forward(A1)
        #then compute A2, which is rho(H2) elementwise
        A2 = self.f[1].forward(H2)
        return A2

    def backward(self, dLdA2):
        """
        Backpropagation operation for MLP with one hidden layer.
        Performs backpropagation on appropriate layers to obtain
        gradient with respect to the input X.
        Does not return anything.

        :param dLdA2: Derivative of loss with respect to output A2.
        Obtained from backward operation on loss object.
        """
        dA2dH2 = self.f[1].backward() #(this is W^(l+a) * dL/dH^(l+1))
        dLdH2 = dLdA2 * dA2dH2 #(this is * dL/dH^(l+1))
        #then compute dLdA1
        dLdA1 = self.layers[1].backward(dLdH2)
        #then compute dA1dH1
        dA1dH1 = self.f[0].backward()
        #then compute dLdH1
        dLdH1 = dLdA1 * dA1dH1
        #then compute dLdX
        dLdX = self.layers[0].backward(dLdH1)
```

```python
class MLP4:

    def __init__(self, num_inputs, num_outputs, num_hiddens):
        """
        Initialize 4 hidden layers and an output layer of shape below:
        Layer1 (num_inputs, num_hiddens),
        Layer2 (num_hiddens, num_hiddens),
        Layer3 (num_hiddens, num_hiddens),
        Layer4 (num_hiddens, num_hiddens),
        Output Layer (num_hiddens, num_outputs)
        Follow all hidden layers with a ReLU activation function.

        :param num_inputs: Number of inputs to model.
        :param num_outputs: Number of outputs from model.
        :param num_hiddens: Size of hidden layer.
        """

        self.layers = [Linear(num_inputs, num_hiddens), Linear(num_hiddens, num_hiddens), Linear(num_hiddens, num_hiddens), Linear(num_hiddens, num_hiddens), Linear
        (num_hiddens, num_outputs)]
        self.f = [ReLU(), ReLU(), ReLU(), ReLU(), Identity()]

    def forward(self, X):
        """
        Forward operation of MLP with four hidden layers.

        :param X: Input data matrix with rows as examples.
        :return A: Output data matrix.
        """
        L = len(self.layers)
        A = X
        for i in range(L):
            H = self.layers[i].forward(X)
            # H = A * self.W[i] + self.b[i]
            A = self.f[i].forward(H)
        return A

    def backward(self, dLdA):
        """
        Backpropagation operation for MLP with four hidden layers.
        Performs backpropagation on appropriate layers to obtain
        gradient with respect to the input X.
        Does not return anything.

        :param dLdA: Derivative of loss with respect to output A.
        Obtained from backward operation on loss object.
        """
        L = len(self.layers)
        for i in reversed(range(L)):
            dAdH = self.f[i].backward(dLdA)
            dLdH = dLdA * dAdH
            dLdA = self.layers[i].backward(dLdH)
```