Georgia Institute of Technology

ECE 4252/8803: Fundamentals of Machine Learning (FunML)

Spring 2025

Homework Assignment # 5

Due: March 14, 2025 @8PM

Please read the following instructions carefully.

- The entire homework assignment is to be completed on this `ipython` notebook. It is designed to be used with `Google Colab`, but you may use other tools (e.g., Jupyter Lab) as well.
- Make sure that you execute all cells in a way so their output is printed beneath the corresponding cell. Thus, after successfully executing all cells properly, the resulting notebook has all the questions and your answers.
- Make sure you delete any scratch cells before you export this document as a PDF. Do not change the order of the questions and do not remove any part of the questions. Edit at the indicated places only.
- Print a PDF copy of the notebook with all its outputs printed. **Submit PDF on `Gradescope`.**
  Then, zip both **PDF** and **IPYNB** in a single **ZIP** file and submit it on `Canvas` under Assignments.
- Rename the PDF, IPYNB and ZIP file according to the format:
  LastName_FirstName_ECE_4252_8803_F24_assignment_5.zip
  LastName_FirstName_ECE_4252_8803_F24_assignment_5.pdf
  LastName_FirstName_ECE_4252_8803_F24_assignment_5.ipynb
- When submitting PDF on Gradescope, make sure to match each question to the corresponding pages. **Incorrect page assignment may lead to reduction of points.**
- It is encouraged for you to discuss homework problems amongst each other, but any copying is strictly prohibited and will be subject to Georgia Tech Honor Code.
- Late homework is not accepted unless arranged otherwise and in advance.
- Comment on your codes.
- Refer to the tutorial and the supplementary/reading materials that are posted on `Canvas` for the first lecture to help you with this assignment.
- **IMPORTANT:** Start your solution with a **BOLD RED** text that includes the words *solution* and the part of the problem you are working on. For example, start your solution for Part (c) of Problem 2 by having the first line as:
  Solution to Problem 2 Part (c). Failing to do so may result in a *20% penalty* of the total grade.

Assignment Objectives:

- Introduction to the use of `PyTorch` basic functions and libraries
- Learn deep model trainining and evaluation
- Learn to use pre-trained deep models for classifying your own images
- Advance in your `PyTorch` knowledge and experience

Recommended Readings

This assignment uses the popular Python-based deep learning framework, `Pytorch`. You are highly encouraged to refer to the following resources as references. The key functions you will be using relate to neural network and defining a dataloader pipeline.

- Introduction to `Pytorch` tensors and autograd
- Defining a neural network architecture
- Setting up a dataloader

The following web pages offer a summarized overview that is worth reading before starting the work in this assignment.

- Overview of `Pytorch`
- How to use a dataloader in `Pytorch`?

Guide for Exporting Ipython Notebook to PDF:

Remeber to convert your homework into PDF format before submitting it.
Here is a video summarizing how to export ipython Notebook into PDF.

- **[Method 1: Print to PDF]**
  After you run every cell and get their outputs, you can use **[File] -> [Print Preview]** , and then use your browser's print function. Choose **[Save as PDF]** to export this Ipython Notebook to PDF for submission.
  Note: Sometimes figures or texts are splited into different pages. Try to tweak the layout by adding empty lines to avoid this effect as much as you can.
- **[Method 2: GoFullPage Chrome Extension]**
  Install the extension and generate PDF file of the Ipython Notebook in the browser.

Note: Since we have embedded images in HW1, it's recommended to generate PDF using the first method. Also, Georgia Tech provides a student discount for Adobe Acrobat subscription. Further information can be found here.

Problem 1: Introduction to PyTorch (10 points)

`PyTorch` is a Python-based scientific computing library for deep learning. `PyTorch` executes deep learning computations over `Tensor` object, which is a specialized data structure that behaves similarly as `Numpy` array but can run on GPUs. It also provides a powerful automatic differentiation engine that computes the gradients during the back-propagation. To enable GPU and PyTorch on AI Makerspace (PACE Cluster), please use the follow configs when launching Jupyter Notebook server (a screenshot is attached as `Cluster_Configs.png` in homework folder).

1. Choose `Pillow 10.2.0 + scikit-learn 1.4.0 + PyTorch 2.1.0 as Anaconda Module.`
2. Choose `NVIDIA GPU (first avail) as Node Type.`

This problem introduces tensor initialization, basic operations and automatic differentiation that you can carry out with `PyTorch` and the standard library `torch`. You may find the Link useful.

Suppose that you are given the following:

$$A = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}, \mathbf{v}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \text{and } \mathbf{v}_2 = \begin{bmatrix} -2 \\ -1 \end{bmatrix}, V = \begin{bmatrix} | & | \\ \mathbf{v}_1 & \mathbf{v}_2 \\ | & | \end{bmatrix}.$$

You are provided with the template code below. Do not change the parts indicated. Your task below is to:
(a) Create the vectors $\mathbf{v}_1, \mathbf{v}_2$, matrices A and V above as `tensor` objects. $\mathbf{v}_1$ and $\mathbf{v}_2$ should have shape `torch.Size([2, 1])`, and A and V should have shape `torch.Size([2, 2])`. *Note: Data shape is important in `numpy` and `torch`. Although (2, 1) and (2,) both have 2 elements in the data, some operations are only valid in one of the forms.*
(b) Write code to create the Hadamard product of A and V
(c) Write code to create the Matrix product AV.
(d) Write your own code to compute the **square** of L2 norm $\|\mathbf{v}_1 - \mathbf{v}_2\|_2^2$. In this part, do not use the `torch.norm` function.
(e) Given $y = \|\mathbf{v}_1 - \mathbf{v}_2\|_2^2$, you will implement automatic differentiation to compute gradients of y with respect to $\mathbf{v}_1$ and $\mathbf{v}_2$. First, re-create two tensors $\mathbf{v}_1$ and $\mathbf{v}_2$ with requires_grad=True. This signals to autograd that every operation on the tensors should be tracked. Thus, the gradients with respect to the tensors can be automatically computed using chain rule. Then, write the code to perform automatic differentiation of y using function torch.autograd.backward. In the printing functions below, `v1.grad` and `v2.grad` automatically compute $\frac{\partial y}{\partial \mathbf{v}_1}$ and $\frac{\partial y}{\partial \mathbf{v}_2}$, respectively. Compare them with the printed manual gradients calculation $\frac{\partial y}{\partial \mathbf{v}_1} = 2(\mathbf{v}_1 - \mathbf{v}_2)$ and $\frac{\partial y}{\partial \mathbf{v}_2} = -2(\mathbf{v}_1 - \mathbf{v}_2)$, do they match?

Problem 1 (a)-(e) Solution

```
## Problem 1

# Import Pytorch Libraries
import torch

## part (a) Create matrices
v1 = torch.tensor([[1], [2]])
v2 = torch.tensor([[-2], [-1]])
A = torch.tensor([[1, -1], [0, 1]]) # define A
```

```
V = torch.tensor([[1, -2], [2, -1]]) # define V

## part (b) Compute the hadamard product
hadamard_AV = A * V   #  hadamard product of A and V

## part (c) Compute the matrix product
Matrix_AV = A @ V   #  matrix product of A and V

## part (d) Compute the L2 norm
square_l2_v = torch.sum((v1-v2)**2)   #  the L2 norm of v1 - v2

## part (e)
# Create tensors and keep track of operations on them
v1 = torch.tensor([[1.0], [2.0]], requires_grad=True)
v2 = torch.tensor([[-2.0], [-1.0]], requires_grad=True)

y = (torch.norm((v1 - v2).float(),2))**2
# Perform automatic differentiation of y
y.backward()

#-----------------Don't change anything below-----------------------#
print('v1: \n', v1)
print('v2: \n', v2)
print('A: \n', A)
print('V: \n', V)
print('\nHadamard Product of A and V: \n', hadamard_AV)
print('\nMatrix Product of A and V: \n', Matrix_AV)
print("L2 norm with \'torch.norm\': %4f" %(torch.norm((v1 - v2).float(),2))**2,' | L2 norm with your function: %4f'  %square_l2_v)
print("The gradient w.r.t. v1 calculated  by automatic differentiation: \n" , v1.grad)
print("The gradient w.r.t. v1 calculated  manually: \n", 2*(v1-v2))
print("The gradient w.r.t. v2 calculated  by automatic differentiation: \n" , v2.grad)
print("The gradient w.r.t. v2 calculated  manually: \n", -2*(v1-v2))
```

```
v1:
 tensor([[1.],
        [2.]], requires_grad=True)
v2:
 tensor([[-2.],
        [-1.]], requires_grad=True)
A:
 tensor([[ 1, -1],
        [ 0,  1]])
V:
 tensor([[ 1, -2],
        [ 2, -1]])

Hadamard Product of A and V:
 tensor([[ 1,  2],
        [ 0, -1]])

Matrix Product of A and V:
 tensor([[-1, -1],
        [ 2, -1]])
L2 norm with 'torch.norm': 17.999998  | L2 norm with your function: 18.000000
The gradient w.r.t. v1 calculated  by automatic differentiation:
 tensor([[6.],
        [6.]])
The gradient w.r.t. v1 calculated  manually:
 tensor([[6.],
        [6.]], grad_fn=<MulBackward0>)
The gradient w.r.t. v2 calculated  by automatic differentiation:
 tensor([[-6.],
        [-6.]])
The gradient w.r.t. v2 calculated  manually:
 tensor([[-6.],
        [-6.]], grad_fn=<MulBackward0>)
```

Problem 2 : Computational Graph and Backpropagation (9 points)

In Problem 1 (e), we have seen the autograd feature in `Pytorch` in a very simple function. Here, we will learn how to plot a computational graph and calculate backpropagation on a more complex function.

Problem 2 (a) Plot Computational Graph

Computation Graph is a directional graph that defines all the arithmetic calculations in variables within the model. In `Pytorch`, the autograd feature uses dynamic computational graph mechanism to keep track of the gradient of every variable with `requires_grad=True` in the

computational graph. Here is an overview about `Pyotrch` autograd feature. For example, the computational graph of

$$g(x_1) = w_1 x_1 + b_1$$

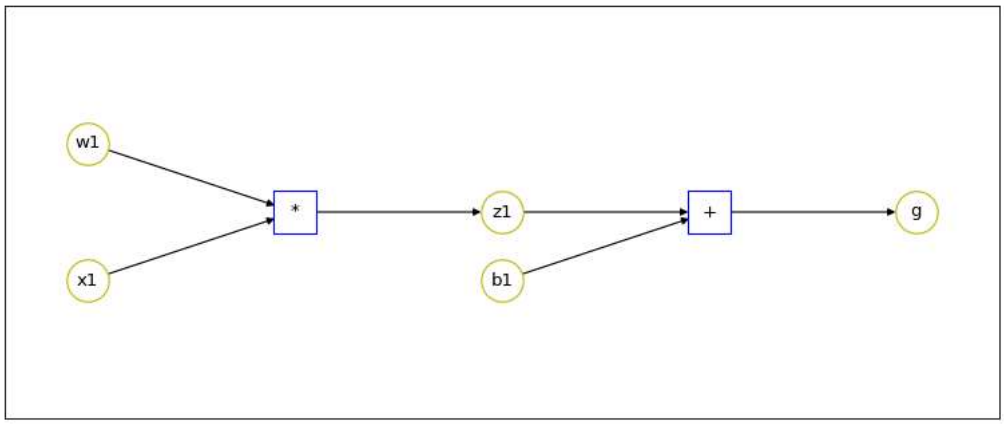
is given in attached file `Computational_Graph_Example.png` in homework folder.

In this question, plot the computaional graph of

$$f(w, x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$

You can plot it by hand or use `networkx` and `matplotlib` libraries. Use circles for variables and rectangles for arithmetic operations. Intermediate states should be labled as z_n.

```
## Example Code of Computational Graph for g(x1)=w1x1+b1
import networkx as nx
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure
figure(figsize=(12, 5), dpi=80)
class Vert:

    # default constructor
    def __init__(self, name, edges):
        self.name = name
        self.edges = edges

nodes = []
nodes.append(Vert('w1', ['mul1']))
nodes.append(Vert('x1', ['mul1']))
nodes.append(Vert('mul1', ['z1']))
nodes.append(Vert('z1', ['add1']))
nodes.append(Vert('b1', ['add1']))
nodes.append(Vert('add1', ['g']))
nodes.append(Vert('g', []))

G = nx.DiGraph()

for v  in nodes:
    G.add_node(v.name)
    for e in v.edges:
        G.add_edge(v.name, e)

positions = {'w1':(-100, 10),
             'x1':(-100, -10),
             'mul1':(-50, 0),
             'z1': (0, 0),
             'b1': (0, -10),
             'add1':(50, 0),
             'g':(100, 0)}

labels = {'w1':'w1',
          'x1':'x1',
          'mul1':'*',
          'z1':'z1',
          'b1':'b1',
          'add1':'+',
          'g':'g'}

nx.draw_networkx_nodes(G, pos=positions, node_shape='o', nodelist=['w1', 'x1', 'z1', 'b1', 'g'], node_size=800, node_color='w', edgecolors='
nx.draw_networkx_nodes(G, pos=positions, node_shape='s', nodelist=['mul1', 'add1'], node_size=800, node_color='w', edgecolors='b')
nx.draw_networkx_labels(G, pos=positions, labels=labels)
nx.draw_networkx_edges(G, pos=positions, arrowsize=10, node_size=800)

plt.xlim(-120, 120)
plt.ylim(-30, 30)

plt.show()
```

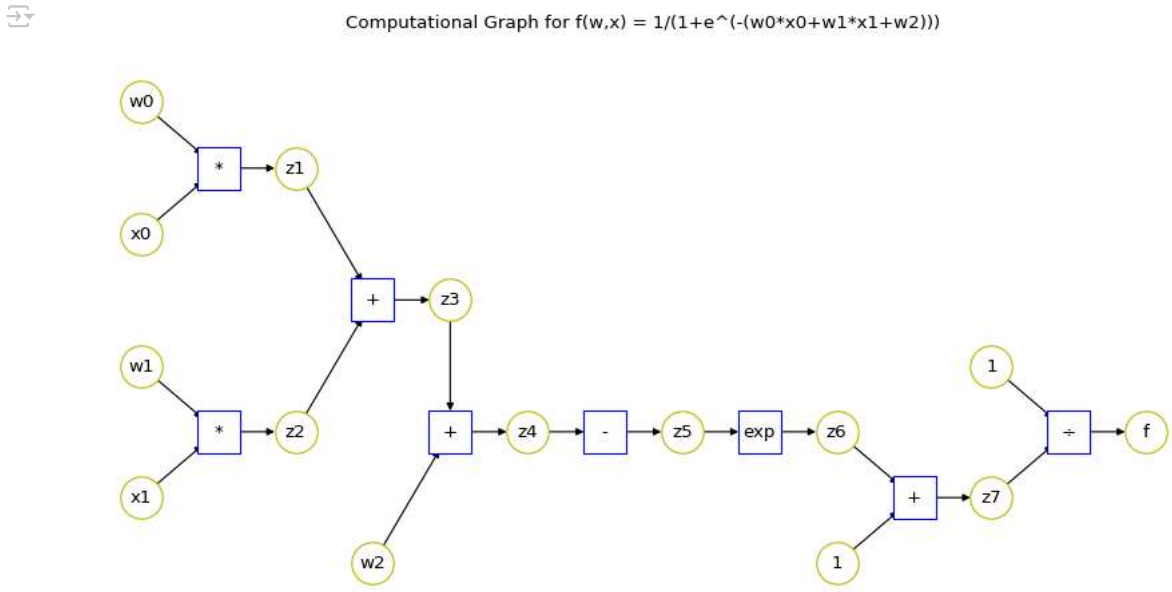


Problem 2 (a) Solution

```
import networkx as nx
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure

figure(figsize=(14, 8), dpi=80)

class Vert:
    # default constructor
    def __init__(self, name, edges):
        self.name = name
        self.edges = edges

nodes = []
# Input variables
nodes.append(Vert('w0', ['mul1']))
nodes.append(Vert('x0', ['mul1']))
nodes.append(Vert('w1', ['mul2']))
nodes.append(Vert('x1', ['mul2']))
nodes.append(Vert('w2', ['add2']))

# Multiplication operations
nodes.append(Vert('mul1', ['z1']))
nodes.append(Vert('mul2', ['z2']))

# First intermediate result
nodes.append(Vert('z1', ['add1']))
nodes.append(Vert('z2', ['add1']))

# First addition
nodes.append(Vert('add1', ['z3']))

# Second intermediate result
nodes.append(Vert('z3', ['add2']))

# Second addition
nodes.append(Vert('add2', ['z4']))

# Third intermediate result (linear combination)
nodes.append(Vert('z4', ['neg']))

# Negation
nodes.append(Vert('neg', ['z5']))

# Fourth intermediate result (negated)
nodes.append(Vert('z5', ['exp']))

# Exponential
nodes.append(Vert('exp', ['z6']))

# Fifth intermediate result (exponential)
nodes.append(Vert('z6', ['add3']))

# Constant 1
```

```
    nodes.append(Vert('const1', ['add3']))

    # Addition with 1
    nodes.append(Vert('add3', ['z7']))

    # Sixth intermediate result (1 + exp)
    nodes.append(Vert('z7', ['div']))

    # Constant 1 for numerator
    nodes.append(Vert('const2', ['div']))

    # Division
    nodes.append(Vert('div', ['f']))

    # Final output
    nodes.append(Vert('f', []))

    # Create the graph
    G = nx.DiGraph()

    for v in nodes:
        G.add_node(v.name)
        for e in v.edges:
            G.add_edge(v.name, e)

    # Define positions for better visualization
    positions = {
        'w0': (-200, 80),
        'x0': (-200, 40),
        'mul1': (-150, 60),
        'z1': (-100, 60),

        'w1': (-200, 0),
        'x1': (-200, -40),
        'mul2': (-150, -20),
        'z2': (-100, -20),

        'add1': (-50, 20),
        'z3': (0, 20),

        'w2': (-50, -60),
        'add2': (0, -20),
        'z4': (50, -20),

        'neg': (100, -20),
        'z5': (150, -20),

        'exp': (200, -20),
        'z6': (250, -20),

        'const1': (250, -60),
        'add3': (300, -40),
        'z7': (350, -40),

        'const2': (350, 0),
        'div': (400, -20),
        'f': (450, -20)
    }

    # Define labels for better readability
    labels = {
        'w0': 'w0',
        'x0': 'x0',
        'w1': 'w1',
        'x1': 'x1',
        'w2': 'w2',
        'mul1': '*',
        'mul2': '*',
        'z1': 'z1',
        'z2': 'z2',
        'add1': '+',
        'z3': 'z3',
        'add2': '+',
        'z4': 'z4',
        'neg': '-',
        'z5': 'z5',
        'exp': 'exp',
```

```
    'z6': 'z6',
    'const1': '1',
    'add3': '+',
    'z7': 'z7',
    'const2': '1',
    'div': '÷',
    'f': 'f'
}

# Draw variable nodes (circles)
variable_nodes = ['w0', 'x0', 'w1', 'x1', 'w2', 'z1', 'z2', 'z3', 'z4', 'z5', 'z6', 'z7', 'const1', 'const2', 'f']
nx.draw_networkx_nodes(G, pos=positions, node_shape='o', nodelist=variable_nodes,
                       node_size=800, node_color='w', edgecolors='y')

# Draw operation nodes (rectangles)
operation_nodes = ['mul1', 'mul2', 'add1', 'add2', 'neg', 'exp', 'add3', 'div']
nx.draw_networkx_nodes(G, pos=positions, node_shape='s', nodelist=operation_nodes,
                       node_size=800, node_color='w', edgecolors='b')

# Add labels and edges
nx.draw_networkx_labels(G, pos=positions, labels=labels)
nx.draw_networkx_edges(G, pos=positions, arrowsize=10, node_size=800)

plt.xlim(-250, 500)
plt.ylim(-100, 100)
plt.title('Computational Graph for f(w,x) = 1/(1+e^(-(w0*x0+w1*x1+w2)))')
plt.axis('off')
plt.show()
```



Problem 2 (b) Forward pass

In this question, the computaional graph is

$$f(x_1, x_2, w_1, w_2) = e^{-(w_1 x_1 + w_2 x_2)}$$

Assume we have $w_1 = 0.1, x_1 = 0.3, w_2 = -0.2, x_2 = 0.4$ and we have $f^* = 1$ which is the desired value of f. And we define error $e = f - f^*$. Calculate the value of f and e, and add f^* and e in the computationl graph.
(all numbers round to at least 3 decimals.)

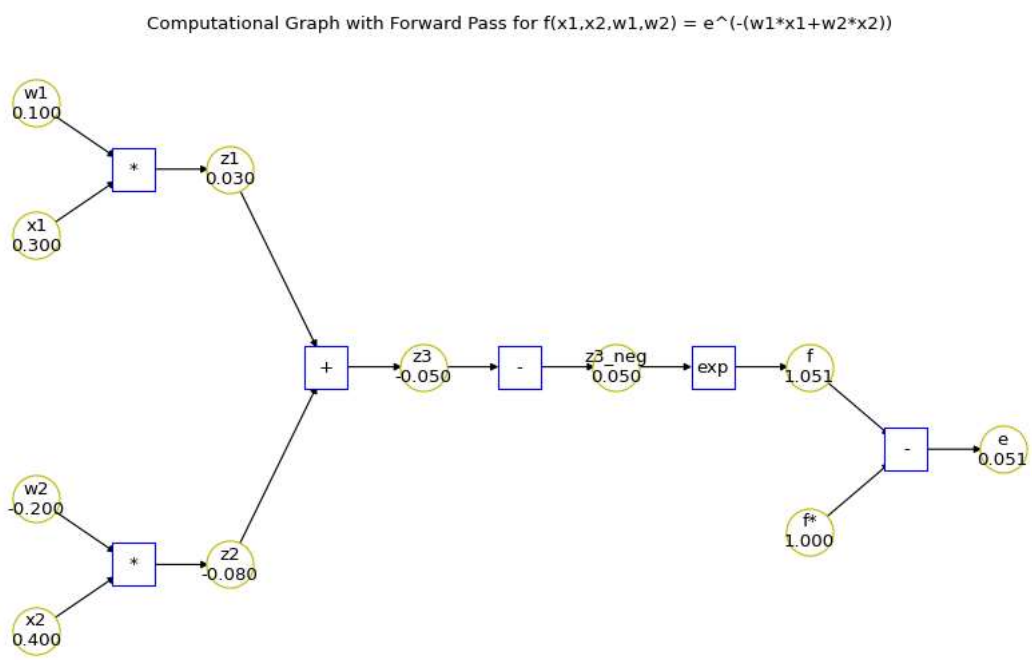
Plot the graph as well

Problem 2 (b) Solution

[Put your answer in this cell]

```
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.pyplot import figure

figure(figsize=(14, 8), dpi=80)

# Given values
w1 = 0.1
x1 = 0.3
w2 = -0.2
x2 = 0.4
f_star = 1.0  # desired output

# Calculate forward pass
z1 = w1 * x1  # = 0.1 * 0.3 = 0.03
z2 = w2 * x2  # = -0.2 * 0.4 = -0.08
z3 = z1 + z2  # = 0.03 + (-0.08) = -0.05
z4 = np.exp(-z3)  # = e^(-(-0.05)) = e^0.05 ≈ 1.051
f = z4  # = 1.051
e = f - f_star  # = 1.051 - 1 = 0.051

class Vert:
    # default constructor
    def __init__(self, name, edges, value=None):
        self.name = name
        self.edges = edges
        self.value = value

nodes = []
# Input variables with their values
nodes.append(Vert('w1', ['mul1'], w1))
nodes.append(Vert('x1', ['mul1'], x1))
nodes.append(Vert('w2', ['mul2'], w2))
nodes.append(Vert('x2', ['mul2'], x2))

# Operations and intermediate results
nodes.append(Vert('mul1', ['z1']))
nodes.append(Vert('z1', ['add1'], z1))
nodes.append(Vert('mul2', ['z2']))
nodes.append(Vert('z2', ['add1'], z2))

nodes.append(Vert('add1', ['z3']))
nodes.append(Vert('z3', ['neg'], z3))

nodes.append(Vert('neg', ['z3_neg']))
nodes.append(Vert('z3_neg', ['exp'], -z3))

nodes.append(Vert('exp', ['f']))
nodes.append(Vert('f', ['sub'], f))

# Target value and error computation
nodes.append(Vert('f*', ['sub'], f_star))
nodes.append(Vert('sub', ['e']))
nodes.append(Vert('e', [], e))

# Create the graph
G = nx.DiGraph()

for v in nodes:
    G.add_node(v.name)
    for edge in v.edges:
        G.add_edge(v.name, edge)

# Define positions for better visualization
positions = {
    'w1': (-200, 80),
    'x1': (-200, 40),
    'mul1': (-150, 60),
    'z1': (-100, 60),

    'w2': (-200, -40),
    'x2': (-200, -80),
    'mul2': (-150, -60),
    'z2': (-100, -60),
```

```
        'add1': (-50, 0),
        'z3': (0, 0),

        'neg': (50, 0),
        'z3_neg': (100, 0),

        'exp': (150, 0),
        'f': (200, 0),

        'f*': (200, -50),
        'sub': (250, -25),
        'e': (300, -25)
    }

    # Create labels that include the calculated values
    labels = {}
    for node in nodes:
        if node.value is not None:
            labels[node.name] = f"{node.name}\n{node.value:.3f}"
        else:
            if node.name in ['mul1', 'mul2']:
                labels[node.name] = '*'
            elif node.name == 'add1':
                labels[node.name] = '+'
            elif node.name == 'neg':
                labels[node.name] = '-'
            elif node.name == 'exp':
                labels[node.name] = 'exp'
            elif node.name == 'sub':
                labels[node.name] = '-'
            else:
                labels[node.name] = node.name

    # Draw variable nodes (circles)
    variable_nodes = ['w1', 'x1', 'w2', 'x2', 'z1', 'z2', 'z3', 'z3_neg', 'f', 'f*', 'e']
    nx.draw_networkx_nodes(G, pos=positions, node_shape='o', nodelist=variable_nodes,
                           node_size=1000, node_color='w', edgecolors='y')

    # Draw operation nodes (rectangles)
    operation_nodes = ['mul1', 'mul2', 'add1', 'neg', 'exp', 'sub']
    nx.draw_networkx_nodes(G, pos=positions, node_shape='s', nodelist=operation_nodes,
                           node_size=800, node_color='w', edgecolors='b')

    # Add labels and edges
    nx.draw_networkx_labels(G, pos=positions, labels=labels)
    nx.draw_networkx_edges(G, pos=positions, arrowsize=10, node_size=800)

    plt.xlim(-250, 350)
    plt.ylim(-100, 100)
    plt.title('Computational Graph with Forward Pass for f(x1,x2,w1,w2) = e^(-(w1*x1+w2*x2))')
    plt.axis('off')
    plt.show()

    print(f"Forward pass calculation:")
    print(f"z1 = w1 * x1 = {w1} * {x1} = {z1:.3f}")
    print(f"z2 = w2 * x2 = {w2} * {x2} = {z2:.3f}")
    print(f"z3 = z1 + z2 = {z1} + ({z2}) = {z3:.3f}")
    print(f"f = e^(-z3) = e^(-({z3})) = e^({-z3:.3f}) = {f:.3f}")
    print(f"e = f - f* = {f:.3f} - {f_star} = {e:.3f}")
```




```
Forward pass calculation:
z1 = w1 * x1 = 0.1 * 0.3 = 0.030
z2 = w2 * x2 = -0.2 * 0.4 = -0.080
z3 = z1 + z2 = 0.03 + (-0.08000000000000002) = -0.050
f = e^(-z3) = e^(-(-0.05000000000000002)) = e^(0.050) = 1.051
e = f - f* = 1.051 - 1.0 = 0.051
```

Problem 2 (c) Backpropagation

In Lecture 13, we learned the idea of backpropagation. Following the concepts in Lecture 13, we can calculate every gradient of the variable in the computational graph in Problem 2 (b) with chain rule. According to your computational graph, calculate the value of

(i) $\frac{\partial e}{\partial f}$

(ii) $\frac{\partial e}{\partial w_1}$

(iii) $\frac{\partial e}{\partial w_2}$

Problem 2 (c) Solution

(i) $\frac{\partial e}{\partial f} = 1$

(ii) $\frac{\partial e}{\partial w_1} = \frac{\partial e}{\partial f} \cdot \frac{\partial f}{\partial w_1} = -x_1 \cdot f = -(0.3) \cdot 1 = -0.3$

(iii) $\frac{\partial e}{\partial w_2} = \frac{\partial e}{\partial f} \cdot \frac{\partial f}{\partial w_2} = -x_2 \cdot f = -(0.4) \cdot 1 = -0.4$

Problem 2 (d) Verify with Pytorch

Now, verify $\frac{\partial e}{\partial w_1}$ and $\frac{\partial e}{\partial w_2}$ you calculate in part (c) using `Pytorch` in the cell below.

```
import torch

w1 = torch.tensor([0.1], requires_grad=True)
x1 = torch.tensor([0.3])

w2 = torch.tensor([-0.2], requires_grad=True)
x2 = torch.tensor([0.4])

f_star = torch.tensor([1.0])
#----------------Don't change anything above----------------------#

z1 = w1 * x1
z2 = w2 * x2
z3 = z1 + z2
f = torch.exp(-z3)
```

```
e = f - f_star
e.backward()
dedw1 = w1.grad
dedw2 = w2.grad

#-----------------Don't change anything below-----------------------#
print('de/dw1:')
print(dedw1)
print('de/dw2:')
print(dedw2)
```

```
de/dw1:
tensor([-0.3154])
de/dw2:
tensor([-0.4205])
```

Problem 3 : Jumpstart to CNN training with `Digits` (20 points)

You may remember from assignment 4 that you were asked to perform image segmentation on the `Digits` dataset in `sklearn`. The `Digits` dataset contains images of handwritten digits $0, 1, \ldots, 9$, each of size 8×8 pixels. We are now going to perform image classification with CNNs. In the process, you are going to learn how to set up a standard training and testing pipeline in `Pytorch`.

Problem 3 (a) Load Data from `sklearn`

Execute the code cell below to load the digits dataset into the workspace and divide it into training and testing sets. Use the `load_digits()` and `train_test_split` functions provided in `sklearn` for this purpose.

Problem 3 (a) Solution

```
## Problem 3 (a)

# imports
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
import numpy as np
import matplotlib.pyplot as plt
from torch.utils.data import Dataset, DataLoader
import torch
import torch.nn as nn

# load data
X, y = load_digits().data, load_digits().target

#-----------------Don't change anything below-----------------------#

# create train test splits
num_train = 500
num_test = X.shape[0] - num_train
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=num_test, train_size=num_train, random_state=4803)
```

Problem 3 (b) Implement Custom `Pytorch` Dataset

Training neural networks in `Pytorch` requires one to instantiate objects called *Dataloaders*, the purpose of which is to extract raw data and present it in the appropriate form to the neural networks. This requires one to set up a custom *Dataset* class and afterwards define its `__getitem__()` and `__len__()` functions. The dataset class is then input to the `torch.utils.data.DataLoader` object to create a data pipeline. This is where one may define various attributes like the batch size, if one wants the batches to be presented in a shuffled order, and so on. The `Pytorch` documentation page here provides an excellent overview of this process. You are provided the template for the *Digits* dataset class and its various functions below. Use the function documentations and helps to guide you to complete the dataset class definition.

Problem 3 (b) Solution

```
## Problem 3 (b)

# set up custom dataset class
```

```
class Digits(Dataset):
  def __init__(self, X, y):
    """Function stores the data and label arrays returned by load_digits function.

    Parameters
    ----------
    X : array_like, shape(Num_samples, Num_of_features)
      numpy array containing the data matrix containing digits training examples
      and features.

    y : array_like, shape(num_samples)
      numpy array containing labels from 0,1,...,9 for each training sample in X
    """

    self.X = X
    self.y = y

  def __getitem__(self, index):
    """function extracts a single example from X and the label given its index.

    Parameters
    ----------
    index : int
      index of a single example to be extracted from X

    Returns
    -------
    input : torch.tensor, shape(1, 8, 8), type torch.float
      indexed example from X reshaped into a single channel grayscale image of
      size 8 x 8 and float datatype.

    target : int, dtype torch.longtensor
      label for input returned as an integer of torch.longtensor datatype
     """
    input = torch.tensor(self.X[index], dtype=torch.float).view(1, 8, 8)
    target = torch.tensor(self.y[index], dtype=torch.long)

    return input, target

  def __len__(self):
    return self.X.shape[0]


#-----------------Don't change anything below-----------------------#

train_batch_size = 100
test_batch_size = 1

train_dataset = Digits(X_train, y_train)
trainloader = DataLoader(train_dataset, batch_size=train_batch_size, shuffle=True)

test_dataset = Digits(X_test, y_test)
testloader = DataLoader(test_dataset, batch_size=test_batch_size, shuffle=True)
```

Problem 3 (c) Define a CNN

The next stage in the process involves defining a neural network class inheriting from the `torch.nn.Module` parent class. The user is required to initialize the network and then define its `forward()` function. The `Pytorch` documentation page here gives an excellent example of this. For our purposes, we are going to use a simple 3-layer network to classify the images presented. The details of the layers are given below:

- 2D convolution layer, kernel size = 3, input channels = 1 , output channels = 16, padding = 1
- ReLU
- 2D convolution layer, kernel size = 3, input channels = 16 , output channels = 32, padding = 1
- ReLU
- Maxpool, reduces input dimensions by half i.e., from 8×8 to 4×4
- Linear layer, maps the 16 pixels in each 4×4 image to a 10-element vector, corresponding to the number of possible output labels.

Complete the class definition for the neural network given below as per the instructions above.

Problem 3 (c) Solution

```
## Problem 3 (c)

# define network
class MySimpleCNN(torch.nn.Module):
  def __init__(self):
    """Inititalizes the various layers in the network"""
    super(MySimpleCNN, self).__init__()
    self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3, padding=1)
    self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, padding=1)
    self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
    self.fc = nn.Linear(in_features=32*4*4, out_features=10)

  def forward(self, x):
    """Processes the input from the dataloaders to return predicted output
    probability vectors for each example in the batch.

    Parameters
    ----------
    x : torch.tensor, shape(batch_size, 1, 8, 8), dtype torch.float
      output from dataloade containing batch_size number of 8 x 8 images as
      torch tensors

    Returns
    -------
    out : torch.tensor, shape (batch_size,10), dtype= torch.float.
      batch_size number of 10-element vectors for each image in the input batch.
    """
    x = torch.relu(self.conv1(x))

    x = self.pool(torch.relu(self.conv2(x)))

    x = x.view(-1, 32 * 4 * 4)

    out = self.fc(x)

    return out
```

Problem 3 (d) Train the CNN

We now move to training the CNN. This requires defining objects for the loss function (Cross entropy in this case), the optimizer (any one of those you learnt in class), passing the network parameters to the optimizer object, and defining the learning rate. In the training loop, one is required to iterate through the training loader, predict the output vector (unnormalized) probabilities, compute the loss, backpropagate, and perform the gradient descent step. Complete the code below to execute the training step. A successfully training process would show the loss decreasing from a high starting value.

Problem 3 (d) Solution

```
## Problem 3 (d)

# perform training
lr = 1e-3
epochs = 100

# initliaze the network
net = MySimpleCNN()

loss_function = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net.parameters(), lr=lr)

for epoch in range(epochs):

  net.train() # training mode

  for iteration, (x, y) in enumerate(trainloader):

    optimizer.zero_grad()

    out = net(x)
    loss = loss_function(out, y)

    loss.backward()
```

```
    optimizer.step()

    print('Epoch : {} | Training Loss : {:0.4f}'.format(epoch, loss.item()))
```

```
Epoch : 88 | Training Loss : 0.0006
Epoch : 88 | Training Loss : 0.0013
Epoch : 88 | Training Loss : 0.0008
Epoch : 89 | Training Loss : 0.0014
Epoch : 89 | Training Loss : 0.0008
Epoch : 89 | Training Loss : 0.0009
Epoch : 89 | Training Loss : 0.0006
Epoch : 89 | Training Loss : 0.0009
Epoch : 90 | Training Loss : 0.0009
Epoch : 90 | Training Loss : 0.0011
Epoch : 90 | Training Loss : 0.0009
Epoch : 90 | Training Loss : 0.0008
Epoch : 90 | Training Loss : 0.0008
Epoch : 91 | Training Loss : 0.0011
Epoch : 91 | Training Loss : 0.0010
Epoch : 91 | Training Loss : 0.0006
Epoch : 91 | Training Loss : 0.0007
Epoch : 91 | Training Loss : 0.0010
Epoch : 92 | Training Loss : 0.0011
Epoch : 92 | Training Loss : 0.0008
Epoch : 92 | Training Loss : 0.0008
Epoch : 92 | Training Loss : 0.0009
Epoch : 92 | Training Loss : 0.0006
Epoch : 93 | Training Loss : 0.0006
Epoch : 93 | Training Loss : 0.0009
Epoch : 93 | Training Loss : 0.0010
Epoch : 93 | Training Loss : 0.0008
Epoch : 93 | Training Loss : 0.0008
Epoch : 94 | Training Loss : 0.0008
Epoch : 94 | Training Loss : 0.0008
Epoch : 94 | Training Loss : 0.0007
Epoch : 94 | Training Loss : 0.0009
Epoch : 94 | Training Loss : 0.0008
Epoch : 95 | Training Loss : 0.0010
Epoch : 95 | Training Loss : 0.0007
Epoch : 95 | Training Loss : 0.0011
Epoch : 95 | Training Loss : 0.0005
Epoch : 95 | Training Loss : 0.0007
Epoch : 96 | Training Loss : 0.0007
Epoch : 96 | Training Loss : 0.0007
Epoch : 96 | Training Loss : 0.0007
Epoch : 96 | Training Loss : 0.0010
Epoch : 96 | Training Loss : 0.0007
Epoch : 97 | Training Loss : 0.0008
Epoch : 97 | Training Loss : 0.0005
Epoch : 97 | Training Loss : 0.0007
Epoch : 97 | Training Loss : 0.0009
Epoch : 97 | Training Loss : 0.0009
Epoch : 98 | Training Loss : 0.0010
Epoch : 98 | Training Loss : 0.0007
Epoch : 98 | Training Loss : 0.0009
Epoch : 98 | Training Loss : 0.0007
Epoch : 98 | Training Loss : 0.0004
Epoch : 99 | Training Loss : 0.0006
Epoch : 99 | Training Loss : 0.0010
Epoch : 99 | Training Loss : 0.0008
Epoch : 99 | Training Loss : 0.0006
Epoch : 99 | Training Loss : 0.0006
```

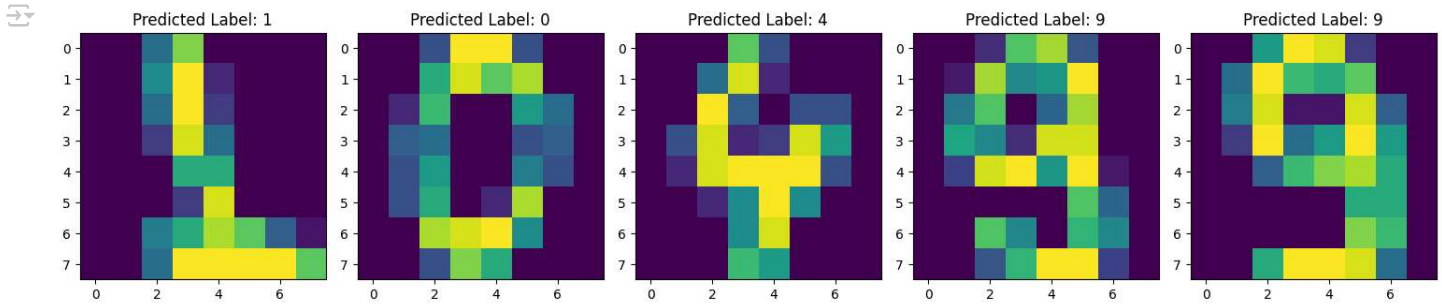
Problem 3 (e) Evaluation Test Result

Finally, we move onto test evaluation using the trained CNN. After training the CNN, the trained parametes will be saved in the CNN object which is our `net` variable here. You simply execute the cell below to visualize randomly sampled images in the test set and the network predictions on those images. Execute the cell multiple times. Are you getting consistently good predictions for all images? Explain some of the reasons why the network is able to learn to classify so well on the Digits dataset with reference to the particular characteristics of the images in the dataset.

Problem 3 (e) Solution

The results are consistently good predictions for all images. There are times when it gets confused and similar numbers are mixed up togehter. It is able to classify so well because the network breaks down the image into important features which are used to figure out how to distinct each feature from other features.

```
## Problem 3 (e)

# perform inference and visualize predictions
net.eval() # testing mode

num_images = 5
fig, axes = plt.subplots(1, num_images, figsize=(15,8))

for i, ax in zip(range(num_images), axes.flatten()):
  x, _ = next(iter(testloader))
  out = net(x)
  ax.imshow(x.detach().cpu().numpy().squeeze())
  ax.set_title('Predicted Label: {}'.format(out.argmax().item()))

plt.tight_layout()
plt.show()
```



Problem 4: Image Classification Using Pre-trained Models (28 points)

You have learned the pipeline of how to train a CNN on a specific dataset. In practice, we do not need to train an entire model from scratch, because it is relatively rare to have a dataset of sufficient size. Instead, it is common to use a pre-trained deep model that is trained on a very large dataset such as `ImageNet` to address your task of interest. In this exercise, you will learn how to directly use pre-trained deep architectures to classify your own images. You can choose photos that you already took or find photos from the web.

Problem 4 (a) Upload Images

Create a folder named `my_images` using the code below (`!mkdir my_images`) to store all your uploaded images. Then, upload your own images (at least 12 images) to the created `my_images` folder onto your working directory.

Problem 4 (a) Solution

```
## Problem 4 (a)

# create a folder named `my_images`
!mkdir my_images
```

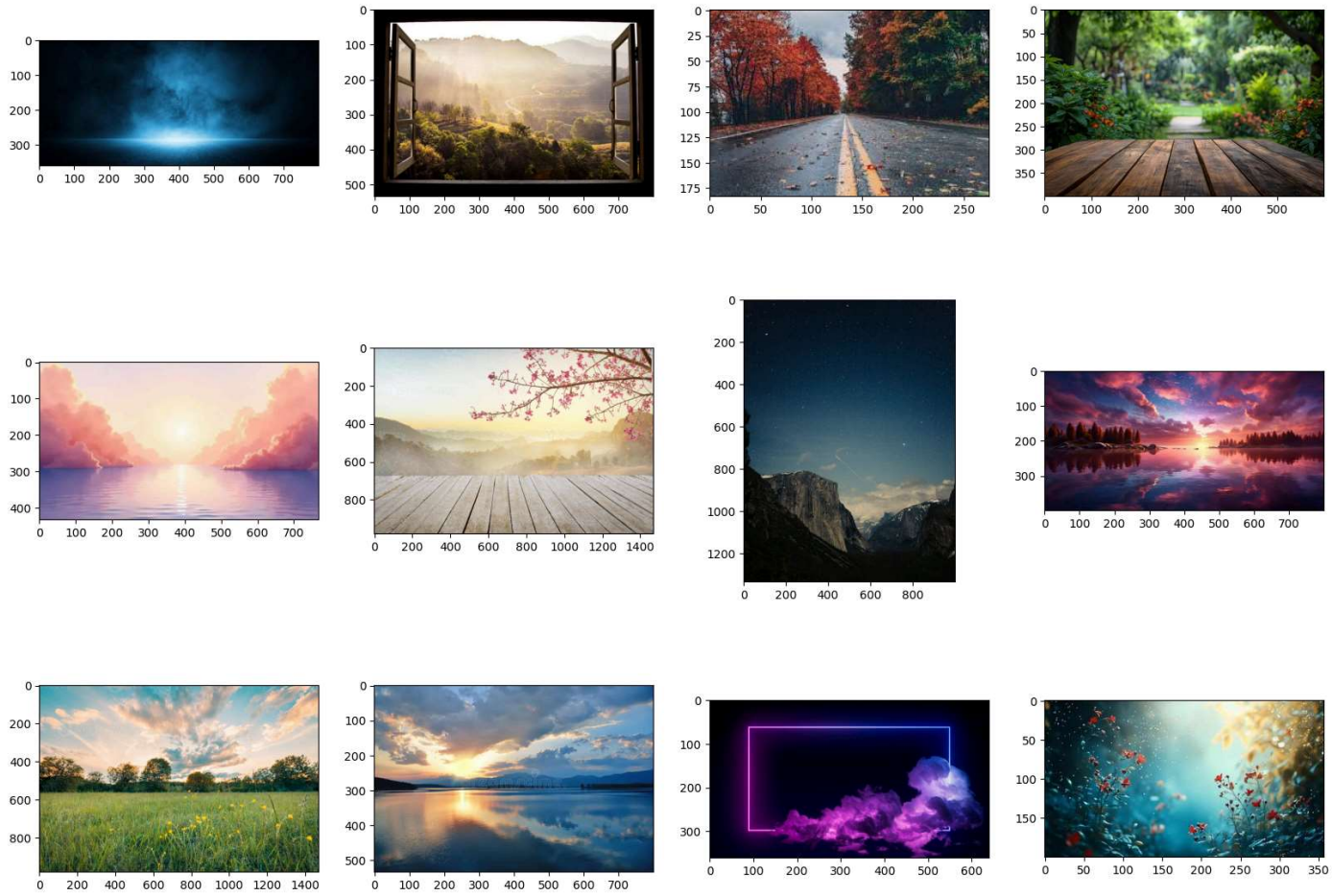
```
mkdir: cannot create directory ‘my_images’: File exists
```

Problem 4 (b) Visualize Images

Visualize your uploaded images by simply executing the code below.

Problem 4 (b) Solution

```
## Problem 4 (b)

from PIL import Image
import matplotlib.pyplot as plt
from glob import glob
import os
import numpy as np

def imread(img_dir):
  # read the images into a list of `PIL.Image` objects
  images = []
  for f in glob(os.path.join(img_dir, "*")):
    images.append(Image.open(f).convert('RGB'))

  return images

def vis_img_label(image_list, label_list=None):
  # visualize the images w/ labels
  Tot = len(image_list)
  Cols = 4
  Rows = Tot // Cols
  Rows += (Tot % Cols)>0
  if label_list is None:
    label_list = [""]*Tot
  # Create a Position index
  Position = range(1,Tot + 1)
  fig = plt.figure(figsize=(Cols*5, Rows*5))
  for i in range(Tot):
    image = image_list[i]
    # add every single subplot to the figure
    ax = fig.add_subplot(Rows,Cols,Position[i])
    ax.imshow(np.asarray(image))
    ax.set_title(label_list[i])


## Load your uploaded images
img_dir = "my_images" ##TODO: Modify this to be your image folder if needed
image_list = imread(img_dir)
## visualize your uploaded images
vis_img_label(image_list)
```



Problem 4 (c) Define Custom Dataset

In this part, you will learn how to prepare your image data for classification by constructing a customized PyTorch dataloader. The customized dataset is provided below as a template class `myDataset`. The class `myDataset` overrides the following methods:

- `__len__` that returns the size of `myDataset` (the length of the `img_list`)
- `__getitem__` to support the indexing such that `myDataset[i]` can be used to get the i-th sample of `img_list`.

Your tasks are:

- Write your code to return the size of `myDataset` in the method `__len__`.
- Write your code to index the i-th sample of `img_list` in the method `__getitem__`.

You may find more details about torch.utils.data.Dataset for constructing a customized PyTorch dataset.

Problem 4 (c) Solution

```
## Problem 4 (c)

from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
```

```
# customized pytorch dataset
class myDataset(Dataset):
  def __init__(self, img_list, data_transform=None):
    self.img_list = img_list   # the list of all uploaded Images
    self.length = len(img_list)
    self.data_transform = data_transform

  def __getitem__(self, index):
    """function extracts a single example from img_list given its index.

    Parameters
    ----------
    index : int
      index of a single example to be extracted from img_list

    Returns
    -------
    img : torch.tensor, shape(3, 224, 224), type torch.float
      indexed example from img_list reshaped into a RGB channel image of
      size 224 x 224 and float datatype.
    """

    img = self.img_list[index]

    if self.data_transform is not None:
        img =self.data_transform(img)  # apply data transformations
    assert img.shape == (3, 224, 224)

    return img

  def __len__(self):
    """
    Returns
    -------
    length : int
      length of img_list.
    """
    length = self.length

    return length

#-----------------Don't change anything below-----------------------#

data_transform = transforms.Compose([transforms.Resize((224, 224)),
                                     transforms.ToTensor(),
                                     transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                          std=[0.229, 0.224, 0.225])])

my_dataset = myDataset(image_list, data_transform)
my_dataloader = DataLoader(my_dataset, batch_size=64, shuffle=False, num_workers=2)

#----verify the customized dataset `myDataset`----#
print('The size of the dataset: ', len(my_dataset))
print('The dimension of the first image sample after transformations: ', my_dataset[0].shape)
```

```
The size of the dataset:  12
The dimension of the first image sample after transformations:  torch.Size([3, 224, 224])
```

Problem 4 (d) Load AlexNet

You have completed the data preparation by constructing a customized dataloader. Now you will start to use a pre-trained model for classifying your uploaded images. The provided code for this part loads the pre-trained AlexNet using the `torchvision.models` module. The pre-trained model is constructed by passing `pretrained=True`. Execute the code below as is and observe the printed AlexNet architecture. Does it match the AlexNet architecture illustrated in the slide 16 of lecture 15?

Problem 4 (d) Solution

```
## Problem 4 (d)

import torchvision
```

```
#Load the pre-trained AlexNet
alexnet = torchvision.models.alexnet(pretrained=True)
print(alexnet)   # print the model architecture
```

```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

Yes, the AlexNet architectures match.

Problem 4 (e) Inference with AlexNet

In this part, you will perform model inference to classify your uploaded images by the pre-trained AlexNet. Write your code to compute the predicted softmax probabilities in the function `predict(model, dataloader)`. You may consider to use the function torch.nn.functional.softmax. Run the code cell to visualize the images with predicted labels. Do these predictions make sense?

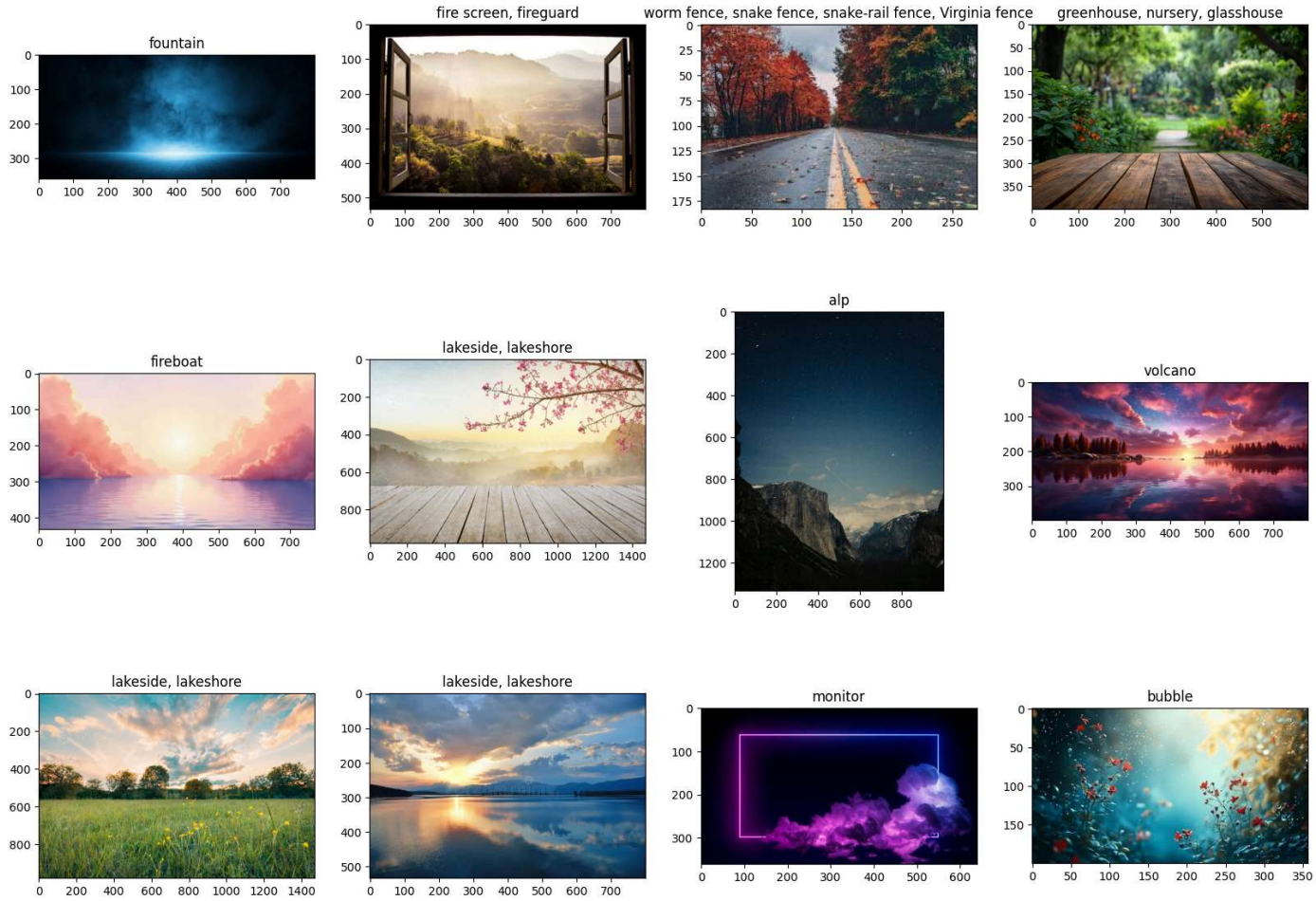
Problem 4 (e) Solution

Yes, most of the predictions make sense. Some of them are slightly wrong though.

```
## Problem 4 (e)

# importing Pytorch Libraries
import torch
import numpy as np
import matplotlib.pyplot as plt
import torch.nn.functional as F

##TODO: Upload label_dict.pth to your working directory.
###   label_dict.pth can be found in homework folder.
label_map = torch.load("label_dict.pth")

# ------------- Do NOT change anything above ------------- #

def predict(model, dataloader):
  pred_total = []

  device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
  model.to(device)

  with torch.no_grad():
    model.eval()  # switch to inference mode
    for batch_id, img_tensor in enumerate(dataloader):

      img_tensor = img_tensor.to(device)
      logits = model(img_tensor)
      output = F.softmax(logits, dim=1)
      pred_class_idx = output.argmax(dim=1)
```

```
        pred_total.append(pred_class_idx.data)

    pred_total = torch.cat(pred_total).cpu().numpy()
  return pred_total


predictions_alexnet = predict(alexnet, my_dataloader) # numpy array of predicted class labels
label_list_alexnet = [label_map[pred] for pred in predictions_alexnet]  # list of class names
print("------------- visualize the images with predicted class names by AlexNet  -------------")
vis_img_label(image_list, label_list_alexnet)
```

```
------------- visualize the images with predicted class names by AlexNet  -------------
```



Problem 4 (f) Vgg16 and Resnet18

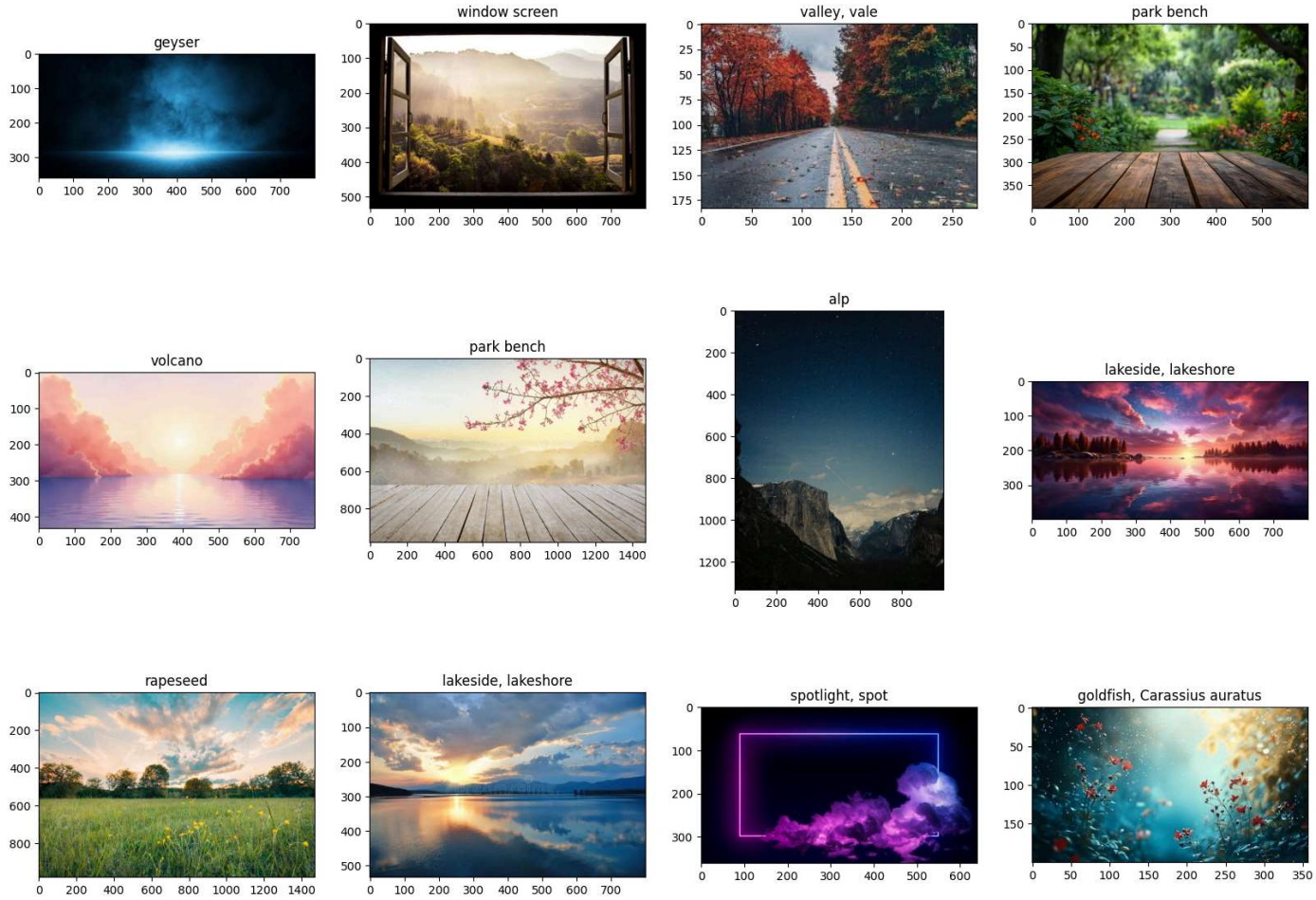
Now you will use other pre-trained model architectures including `vgg16` and `resnet18` for classifying your images. Simply execute the two code cells below for visualizing the images with predicted labels. Compare the predictions of different architectures. Do the predictions by a more complex model always make more sense?

Problem 4 (f) Solution

Typically, these predictions make more sense.

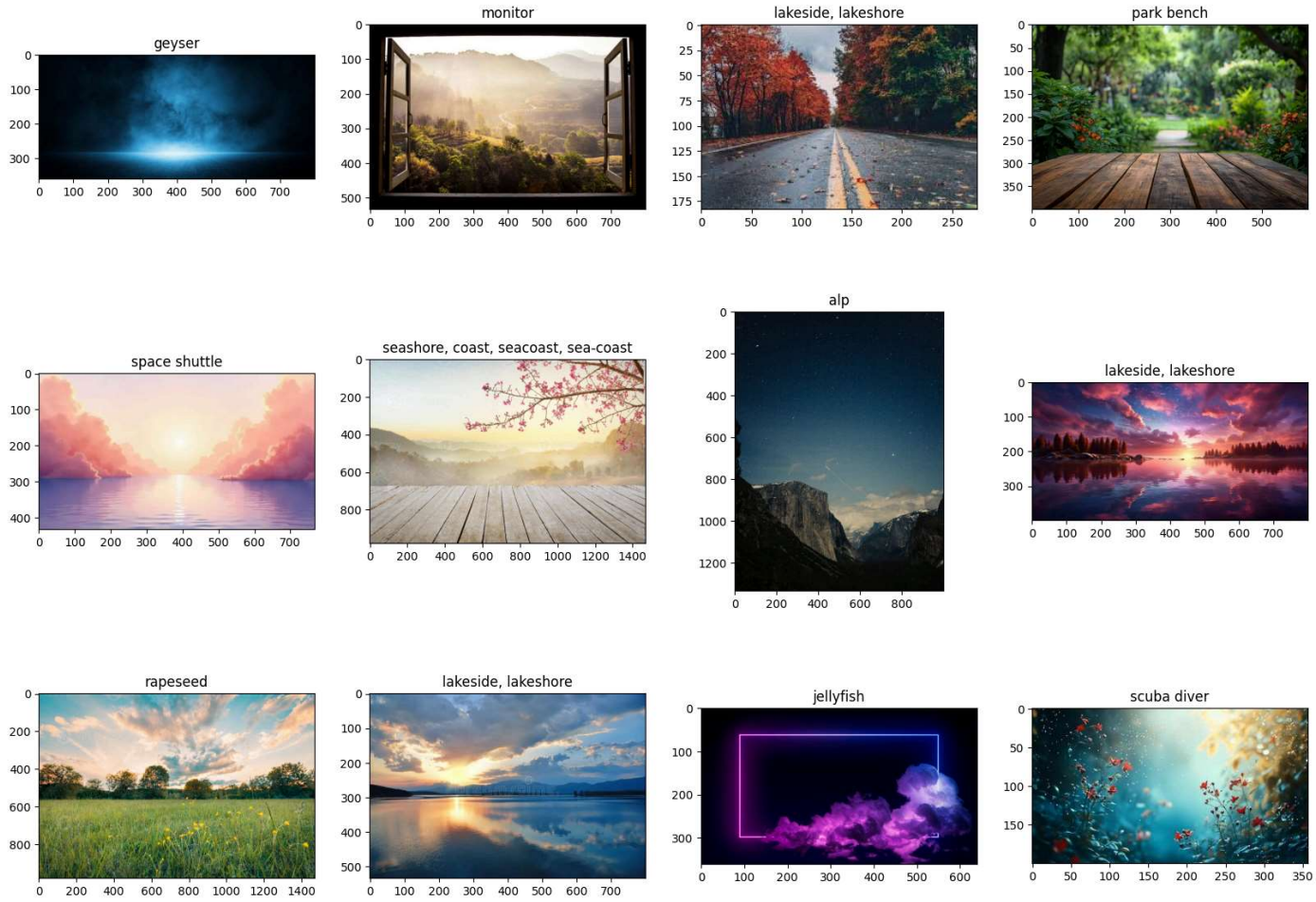
```
## Problem 4 (f) Vgg16

vgg16 = torchvision.models.vgg16(pretrained=True)

predictions_vgg16 = predict(vgg16, my_dataloader) # numpy array of predicted class labels
label_list_vgg16 = [label_map[pred] for pred in predictions_vgg16]  # list of class names
print("------------- visualize the images with predicted class names by VGG16  -------------")
vis_img_label(image_list, label_list_vgg16)
```

```
------------- visualize the images with predicted class names by VGG16  -------------
```




```
## Problem 4 (f) Resnet18

resnet18 = torchvision.models.resnet18(pretrained=True)

predictions_resnet18 = predict(resnet18, my_dataloader) # numpy array of predicted class labels
label_list_resnet18 = [label_map[pred] for pred in predictions_resnet18]  # list of class names
print("------------- visualize the images with predicted class names by ResNet18  -------------")
vis_img_label(image_list, label_list_resnet18)
```

```
------------- visualize the images with predicted class names by ResNet18  -------------
```



Problem 4 (g) Interpret Result

Observe the predictions of different architectures and provide your answers for the following questions:

- List some predictions that are closely related to your labels or what are presented in the corresponding images. For example, AlexNet predicts an image of a tiger as 'tiger' , or VGG16 predicts an image of an giraffe as 'gazelle'.
- Some of the predictions are not related but have reasonable explanations. For example, Resnet18 predicts a giraffe as 'honeycomb'. Such prediction is made because the fur pattern of a giraffe somehow looks alike the structure of honeycomb.
- The three architectures might have different predictions on the same images. For those cases, does a more complex model always generate predictions that are more related?

Problem 4 (g) Solution

Most images are predicted correctly across all models, such as identifying a koala as "koala bear," a lakeside scene as "lakeside," and an elephant as "tusker." However, some predictions show varying levels of accuracy. For example, a chipmunk is labeled as "wood rabbit" by AlexNet, "meerkat" by VGG16, and "fox squirrel" by ResNet18 - all similar rodents but not exactly correct, suggesting these models may lack training on that specific animal.

Interestingly, model complexity doesn't always guarantee better predictions. In one case, both AlexNet and VGG16 correctly identify a flower as a "daisy," while the more complex ResNet18 incorrectly labels it an "anemone." Conversely, the more complex models correctly classify a cliff scene as a "cliff" or "drop-off," while AlexNet misidentifies it as a volcano. This demonstrates that more complex architectures sometimes but not always produce more accurate predictions.

Problem 5: Multi Layer Perceptron (28 points)

A multilayer perceptron (MLP) is a fully connected class of feedforward artificial neural networks (ANN). An MLP consists of at least three layers of nodes: an input layer, a hidden layer, and an output layer. Except for the input nodes, each node is a neuron that uses a nonlinear activation function. In this exercise, you will construct the MLP from scratch.

One of the first steps when working with a new data set is preprocessing. However you come about your data, whether downloaded from an established dataset or created yourself, it's vital to carefully examine and prepare your data. Our data here is from the MNIST dataset of `28x28` pixel images of handwritten digits from `0-9`, which has an important place in the history of machine learning. Some common questions to ask yourself when confronted with a new data set when preparing analysis:

1. What range of values does your data take? Do values need to be rescaled?
2. What shape is your data? Should each sample be a matrix ($n \times m$), a vector ($1 \times n$), or even a multi-dimensional tensor ($n \times m \times p \times \ldots$)?
3. Are there fringe cases that need special attention?

Our data lives in the directory `sample_data` under the file names `mnist_train_small.csv` and `mnist_test.csv`, so the complete file path (from the main directory) are `sample_data/mnist_train_small.csv` and `data/mnist_test.csv`. With that in mind, let's define a helper function `load_data` that we can use to quickly in our data.

```
import time
import numpy as np
import matplotlib.pyplot as plt
import os

def load_data(dir_name):
    """
    Function for loading MNIST data stored in comma delimited files. Labels for
    each image are the first entry in each row.

    Parameters
    ----------
    dit_name : str
         Path to where data is contained

    Returns
    -------
    X : array_like
        A (N x p=784) matrix of samples
    Y : array_like
        A (N x 1) matrix of labels for each sample
    """
    data = list() # init a list called `data`
    with open(dir_name,"r") as f: # open the directory as a read ("r"), call it `f`
        for line in f: # iterate through each `line` in `f`
            split_line = np.array(line.split(',')) # split lines by `,` - cast the resultant list into an numpy array
            split_line = split_line.astype(np.float32) # make the numpy array of str into floats
            data.append(split_line) # collect the sample into the `data` list
    data = np.asarray(data) # convert the `data` list into a numpy array for easier indexing

    # As the first number in each sample is the label (0-9), extract that from the rest and return both (X,Y)
    return data[:,1:],data[:,0]

X_train,Y_train = load_data("sample_data/mnist_train_small.csv")
X_test,Y_test = load_data("sample_data/mnist_test.csv")
```

Problem 5 (a) One Hot Encoding

One hot encoding is a technique used to represent categorical variables as numerical values in a machine learning model. The advantages of using one hot encoding include:

- It allows the use of categorical variables in models that require numerical input.
- It can improve model performance by providing more information to the model about the categorical variable.

- It can help to avoid the problem of ordinality, which can occur when a categorical variable has a natural ordering (e.g. “small”, “medium”, “large”).

Familiarize yourself with one hot encoding and construct the one hot encodings for the target values of our data.

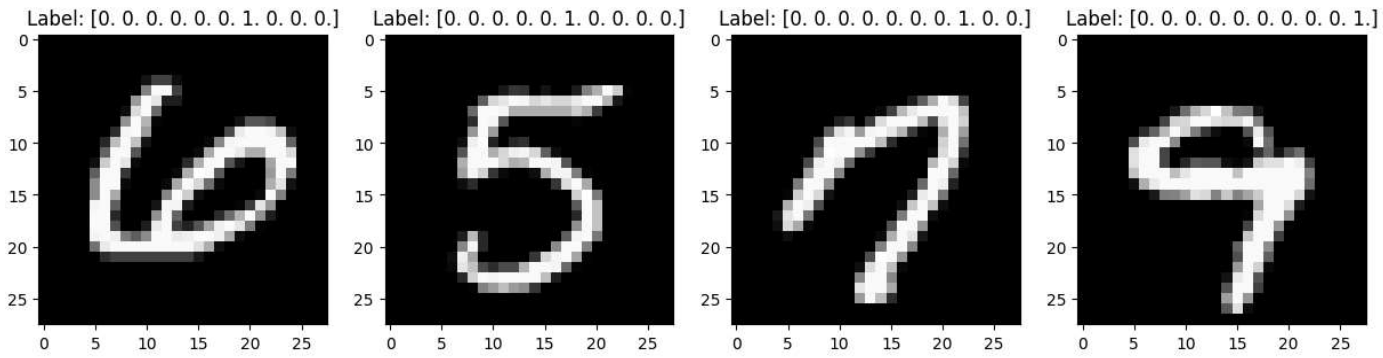
Problem 5 (a) Solution

```
# rescale data between 0 - 1.0
X_train = X_train/X_train.max()
X_test = X_test/X_test.max()

# one-hot encode train (y_train) and test (y_test) set labels
y_train = np.zeros((Y_train.shape[0], 10))
y_test = np.zeros((Y_test.shape[0], 10))

y_train[np.arange(Y_train.shape[0]), Y_train.astype(int)] = 1
y_test[np.arange(Y_test.shape[0]), Y_test.astype(int)] = 1

#-----------------Don't change anything below-----------------------#

print("Sanity check for the training samples")
print("The one-hot encoding for training sample 0 is ", y_train[0], ",where the actual integer value is ", Y_train[0])
print("The one-hot encoding for training sample 0 is ", y_train[10], ",where the actual integer value is ", Y_train[10])

print("Sanity check for the testing samples")
print("The one-hot encoding for testing sample 0 is ", y_test[0], ",where the actual integer value is ", Y_test[0])
print("The one-hot encoding for testing sample 0 is ", y_test[10], ",where the actual integer value is ", Y_test[10])
```

```
Sanity check for the training samples
The one-hot encoding for training sample 0 is  [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.] ,where the actual integer value is  6.0
The one-hot encoding for training sample 0 is  [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.] ,where the actual integer value is  2.0
Sanity check for the testing samples
The one-hot encoding for testing sample 0 is  [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.] ,where the actual integer value is  7.0
The one-hot encoding for testing sample 0 is  [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.] ,where the actual integer value is  0.0
```

Let's visualize some of the inputs and their associated labels. We can do this using the `imshow` function (making sure to resize the flattened size 784 representation into a 28 x 28 matrix for compatability with `imshow`).

```
num_images = 4
fig,axes = plt.subplots(1,num_images,figsize=(15,10))
for image,label,ax in zip(X_train[:num_images],y_train[:num_images],axes):
    ax.imshow(image.reshape(28,28),cmap='gray',vmin=0,vmax=1.0)
    ax.set_title(f"Label: {label}")
```



The Mathematical Foundation

Here some of the math underlying the multilayer perceptron (MLP) and the backpropagation algorithm are reviewed. The MLP and backpropagation are central to understanding deep learning as a whole. For more information please see the link.

Each input in a conventional perceptron is represented as a multidimensional vector.

$$\mathbf{x} = \{x_0 = 1, x_1, x_2, \ldots, x_n\}$$

They are multiplied by a set of weights;

$$\mathbf{w} = \{w_0, w_1, w_2, \ldots, w_n\}$$

Together, they produce the weighted sum;

$$z = \sum_{i=0}^{n} x_i w_i$$

Problem 5 (b) Forward Propagation, Sigmoid Function, and Softmax Function

Given a predefined neural network *architecture* (the *architecture* of a neural network refers to all the elements necessary to completely define the flow of data, which involve the number and size of hidden layers, which activation functions, the output size, etc.) the process of generating an output from an input is called a *forward* pass. As we shall see, for an MLP, the forward pass may be succinctly represented as a series of matrix multiplications.

Problem 5 (b-i) Sigmoid Function

Consider the MLP represented schematically above with sigmoid activations σ in the hidden layer. Each neuron in the hidden layers will be weighted sums of the inputs:

$$a_j = \sigma(\sum_{i=1}^{784} x_i * w_{i,j})$$

for $j = \{1, 2, \ldots, 15\}$ (notice, there is no bias term in this example). From here it clear to see that the weights $w_{i,j}$ may be compacted into a matrix $W \in \mathbb{R}^{784\times 15}$ where $W_{i,j} = w_{i,j}$, allowing for all the neurons in the hidden layer to be efficiently calculated using matrix multiplication: $\mathbf{a}^{(1)} = \sigma(\mathbf{z}^{(1)}) = \sigma(\mathbf{x}W)$. Where $\mathbf{a}^{(i)}, \mathbf{z}^{(i)} \in \mathbb{R}^{1\times 15}$, with the superscript (i) indicating the assocaited layer number, and $\mathbf{x} \in \mathbb{R}^{1\times 784}$ are both arranged as column vectors. Another, different, weight matrix is needed to transform the hidden layer to the output layer. Let's demarcate these two as $W^{(1)}$ for the matrix which transforms the inputs to the hidden layer $\mathbf{a}^{(1)}$ and $W^{(2)} \in \mathbb{R}^{15\times 10}$ for transforming the hidden layer to the output layer $\mathbf{z}^{(2)}$. Mathematically;

$$\mathbf{z}^{(2)} = \sigma(\mathbf{x}W^{(1)})W^{(2)}$$

Notice, we've yet to treat this output $\mathbf{z}^{(2)}$ with an activation function.

In this part you will implement the sigmoid function.

Problem 5 (b-i) Solution

```
def sigmoid(x):
    """
    Compute the sigmoid of `x`, calculated element-wise

    Parameters
    ----------
    x : float or array_like
        input

    Returns
    -------
    sigmoid(x) : float or array_like
        sigmoid applied to `x` element-wise
    """
#-----------------Don't change anything above-----------------------#
    return 1/(1+np.exp(-x))
#-----------------Don't change anything below-----------------------#

test_x = np.arange(-5, 5, 0.1).reshape(10, 10)
print(sigmoid(test_x))
```

```
[[0.00669285 0.00739154 0.00816257 0.0090133  0.0099518  0.01098694
  0.01212843 0.01338692 0.01477403 0.0163025 ]
 [0.01798621 0.01984031 0.02188127 0.02412702 0.02659699 0.02931223
  0.03229546 0.03557119 0.03916572 0.04310725]
 [0.04742587 0.05215356 0.05732418 0.06297336 0.06913842 0.07585818
  0.0831727  0.09112296 0.09975049 0.10909682]
 [0.11920292 0.13010847 0.14185106 0.15446527 0.16798161 0.18242552
  0.19781611 0.21416502 0.23147522 0.24973989]
 [0.26894142 0.2890505  0.31002552 0.33181223 0.35434369 0.37754067
  0.40131234 0.42555748 0.450166   0.47502081]
 [0.5        0.52497919 0.549834   0.57444252 0.59868766 0.62245933
  0.64565631 0.66818777 0.68997448 0.7109495 ]
 [0.73105858 0.75026011 0.76852478 0.78583498 0.80218389 0.81757448
  0.83201839 0.84553473 0.85814894 0.86989153]
```

```
[0.88079708 0.89090318 0.90024951 0.90887704 0.9168273  0.92414182
 0.93086158 0.93702664 0.94267582 0.94784644]
[0.95257413 0.95689275 0.96083428 0.96442881 0.96770454 0.97068777
 0.97340301 0.97587298 0.97811873 0.98015969]
[0.98201379 0.9836975  0.98522597 0.98661308 0.98787157 0.98901306
 0.9900482  0.9909867  0.99183743 0.99260846]]
```

Problem 5 (b-ii) Softmax Function

The last step of process for classification tasks is actually producing a prediction from these numbers in the output layer $\mathbf{z}^{(2)}$. Typically, in the case of multi-label classification, this is done using a *softmax* activation function which effectively converts the output neurons into probabilities for each label. This has the mathematical form;

$$\text{softmax}(\mathbf{z})_j = \frac{\exp(z_j)}{\sum_{k=0}^{K=9} \exp(z_k)}$$

The softmax activation function has the property of the outputs summing to 1 $\sum_{k=1}^{K=9} \text{softmax}(\mathbf{z}^{(2)})_k = 1$, allowing each output $\mathbf{a}_i^{(2)} = \text{softmax}(\mathbf{z}^{(2)})_i$ to be interpreted as the probability that the input is actually a digit `0-9`. Note, the output of a MLP does not need to have a softmax activation, for example in a regression setting a softmax activation would not make much sense. When evaluating the classification accuracy of the neural network, the input is typically classified according to the output label with the highest probability.

$$\mathbf{x}_{\text{predicted}} = \arg\max_j \left(\sigma(\mathbf{z}_j^{(2)})\right) = \arg\max_j \left(\frac{e^{z_j^{(2)}}}{\sum_{j=1}^{K} e^{z_j^{(2)}}}\right) = \arg\max_j \left(\sigma(\mathbf{x}W^{(1)})W^{(2)})_j\right)$$

In this part, you will implement the softmax function.

Problem 5 (b-ii) Solution

```
def softmax(x):
    """
    Compute the softmax of `x`,

    Parameters
    ----------
    x : array_like
        (N, dim) array with N samples by p dimensions. dim=10 for MNIST classification.

    Returns
    -------
    softmax(x) : float or array_like
        softmax applied to `x` along the first axis.
    """

    return np.exp(x)/np.sum(np.exp(x),axis=-1,keepdims=True)

#-----------------Don't change anything below-----------------------#
rng = np.random.default_rng(seed=8803)
test_x = rng.random((5, 10))
print(softmax(test_x))
```

```
[[0.11308373 0.09495527 0.10991099 0.12813875 0.07212035 0.11325271
  0.10747322 0.0645221  0.08195541 0.11458748]
 [0.114959   0.09385948 0.12861697 0.06580683 0.11962318 0.14543819
  0.08600428 0.07373911 0.074713   0.09723995]
 [0.06312498 0.13306552 0.07843508 0.13109671 0.06385997 0.07206117
  0.06007973 0.14235254 0.11508851 0.14083578]
 [0.13846278 0.08526933 0.07106004 0.06695615 0.09966726 0.05240115
  0.12935594 0.11170343 0.13277632 0.11234761]
 [0.10690917 0.10604298 0.15582542 0.07124933 0.10735521 0.13459844
  0.07940872 0.06721449 0.07579166 0.09560457]]
```

Problem 5 (b-iii) Forward Propagation

The inputs `x`, weight matricies `w`, and activations are in principle all we need to define the forward pass; however, for efficency reasons we'll want to store the outputs of the hidden layer neurons when performing the forward pass. Storing these values will help us later more quickly calculate the gradients during the backward pass. The `init_layers` functions will initalize these hidden layers as NumPy arrays, doing this before we begin training will help us save some overhead we would otherwise inccur reinitalizing these hidden layers before each forward pass. These hidden layer values will be stored in multi-dimensional matricies, called *tensors*. One dimension of these tensors will be the

`batch size` which will indicate the number of samples simultaneously passed to MLP during one training loop (forward propagation and backpropagation).

```
def init_layers(batch_size, layer_sizes):
    """
    Initalize arrays to store the hidden layer ouputs.

    Parameters
    ----------
    batch_size : int
        Number of samples to concurrently feed through the network.
    layer_sizes : array_like
        List of length `N_l`. Each entry is the number of neurons in each layer.

    Returns
    -------
    hidden_layers : list
        List of empty numpy arrays used to hold hidden layer outputs.
    """
    hidden_layers = [np.empty((batch_size,layer_size)) for layer_size in layer_sizes]
    return hidden_layers
```

In order to perform a forward pass our input `x` is consequtively multiplied by weight matricies passed into the associated activaiton functions. The parameters in these weight matricies will ultimately be learned through backpropagation, but each weight matrix must first be initalized to random values. There are a number of different methods for doing this initalization, but for the moment we'll use a simple approach of just drawing the numerical values from a normal distribution with mean zero and standard deviation 1. We could have also reasonably choosen to simply draw from a uniform distribution on the range `[-1,1]`.

Think about the following question. Why would it be wrong to initalize the weight matricies with all zeros?

```
def init_weights(layer_sizes, fix_seed = False):
    """
    Initalize the paramters of the weight matricies.

    Parameters
    ----------
    layer_sizes : array_like
        List of length `N_l`. Each entry is the number of neurons in each layer.

    Returns
    -------
    weights : array_like
        List of randomly initalized weight numpy matricies based on the layer sizes.
    """
    weights = list()

    for i in range(layer_sizes.shape[0]-1):
      if fix_seed:
        if i == 1:
          rng = np.random.default_rng(seed=8803)
        else:
          rng = np.random.default_rng(seed=4803)
        weights.append(rng.uniform(-1,1,size=[layer_sizes[i],layer_sizes[i+1]]))
      else:
        weights.append(np.random.uniform(-1,1,size=[layer_sizes[i],layer_sizes[i+1]]))

    return weights
```

Finally, we define the `feed_forward` function to iterate through the calculations to perform the forward pass. Take every layer, multiply the vector associated with the layer, and then pass the output of the multiplication through a sigmoid function. Remember that the final hidden layer will be given input to a softmax function.

Problem 5 (b-iii) Solution

```
def feed_forward(batch,hidden_layers,weights):
    """
    Perform a forward pass of the neural network.
```

```
    Parameters
    ----------
    batch : array_like
        (batch_size, dim) matrix of inputs
    hidden_layers : list
        List of hidden layer outputs
    weights : array_like
        Array of weight matricies

    Returns
    -------
    output : array_like
        Forward pass output of the MLP
    hidden_layers : array_like
        List of hidden layer outputs, populated from the forward pass.
    """
    h_l = batch
    hidden_layers[0] = h_l
    for i,weight in enumerate(weights):
        h_l = sigmoid(np.dot(h_l,weight))
        hidden_layers[i+1] = h_l
    output = softmax(h_l)
    return output, hidden_layers


#-----------------Don't change anything below-----------------------#

layer_sizes = np.array([X_train.shape[1]]+[128]*3+[np.squeeze(np.eye(10)[Y_train.astype(int).reshape(-1)]).shape[1]])
test_weights = init_weights(layer_sizes, fix_seed = True)
test_batch_size = 2
hidden_layers = init_layers(test_batch_size, layer_sizes)

test_output, test_hidden_layers = feed_forward(X_train[0:test_batch_size,:], hidden_layers, test_weights)
print("Layer sizes: ", layer_sizes)
for i, weight in enumerate(test_weights):
  print("Shape of the", i, " weight arrays: ", test_weights[i].shape)
for i, weight in enumerate(hidden_layers):
  print("Shape of the", i, " hidden layer: ", hidden_layers[i].shape)
for i in range(test_batch_size):
  print("Output for the test input ",i ," : ", test_output[i])
print("A number just for grading purposes: ", np.max(test_hidden_layers[4]))
```

```
Layer sizes:  [784 128 128 128  10]
Shape of the 0  weight arrays:  (784, 128)
Shape of the 1  weight arrays:  (128, 128)
Shape of the 2  weight arrays:  (128, 128)
Shape of the 3  weight arrays:  (128, 10)
Shape of the 0  hidden layer:  (2, 784)
Shape of the 1  hidden layer:  (2, 128)
Shape of the 2  hidden layer:  (2, 128)
Shape of the 3  hidden layer:  (2, 128)
Shape of the 4  hidden layer:  (2, 10)
Output for the test input  0  :  [0.10290618 0.07406508 0.15378217 0.11907858 0.05749531 0.1463068
 0.08404077 0.10641721 0.06097971 0.09492819]
Output for the test input  1  :  [0.12743459 0.07623965 0.17051266 0.07784414 0.06752908 0.18120419
 0.06783913 0.08470183 0.07175165 0.0749431 ]
A number just for grading purposes:  0.9981910969248503
```

Problem 5 (c) Backpropagation

Now that we have an understanding of how an MLP generates outputs from inputs, we engage the problem of how to actually *train* this neural network. *Training* a neural network (in a supervised setting, which means each input comes with a known output) refers to the process of iteratively updating the weights of the network to improve it's performance. The performance of the neural network is evaluated using a *loss function* which quantitatively measures how "close" the neural network output is to the true output. In short, using *backpropagation* we aim to minimize the loss function with respect to the weights (also called *parameters*) of the neural network.

Problem 5 (c-i) Derivative of Sigmoid Function

Initalizing all the weights, layers, and activations prior to the forward pass makes much of the backward pass implimentation actually quite simple. For convinience we'll define a `sigmoid_prime` function, which simply computes the derivative of the sigmoid activation σ'. We'll use this when computing the gradients during the backward pass. The derivative of the sigmoid function is given by the following equation.

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

Complete the following `sigmoid_prime` function.

Problem 5 (c-i) Solution

```
def sigmoid_prime(sigmoid_out):
    """
    Calculate derivative of sigmoid activation based on sigmoid output.

    Parameters
    ----------
    sigmoid_out : array_like
        Output of th sigmoid function.

    Returns
    -------
    sigmoid_prime(h) : array_like
        Derivative of sigmoid, based on value of sigmoid.
    """
    return sigmoid_out*(1-sigmoid_out)
```

Problem 5 (c-ii) Completing Backpropagation

With everything else in place computing we're finally ready to write the backpropagation algorithm. Again, the primary goal of this step is to update the parameters of the weight matricies using stochastic gradient descent on batches of training samples, measuring the error by comparing the outputs of neural network with the true labels.

After softmax function, we select the class with the highest probability as our estimate.

For the loss function, a cross-entropy loss function is selected, which can be formulated as follows;

$$L = -\sum_i y_i log(p_i)$$

If every iteration, only a batch $\mathcal{B}$, of samples are given to the forward and backpropagation, this equation can be updated as follows;

$$L = -\frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} y_i log(p_i)$$

The derivative of this loss function with respect to the last hidden layer (which is not passed through the softmax function) can be related as follows;

$$\frac{\partial L}{\partial o_i} = p_i - y_i$$

Note that, in this equation, o_i represents the last hidden layer, p_i represents the output of the softmax function, which took o_i as input, and y_i represents the ground truth information in one-hot coded form. For the derivation of this graident, please see the link.

Complete the following backpropagation function.

Problem 5 (c-ii) Solution

```
from re import U
def back_prop(output, batch_y, hidden_layers, weights, batch_size, lr):
        """
        Calculate derivative of sigmoid activation based on sigmoid output.

        Parameters
        ----------
        output : array_like
            Forward pass output of the MLP
        batch_y : array_like
            True labels for the samples in the batch
        hidden_layers : list
            List of hidden layer outputs
        weights : array_like
            Array of weight matricies
        lr : float
            Learning rate for SGD
        batch_size : int
            Size of a training mini-batch
```

```
        Returns
        -------
        weights : array_like
            Array of weight matricies, updated from the backpropagation.

        """
        delta_t = (output - batch_y)*sigmoid_prime(hidden_layers[-1])
        for i in range(1, len(weights)+1):
            weights[-i] -= lr * (hidden_layers[-i-1].T.dot(delta_t))/batch_size
            if i < len(weights):  # Don't calculate delta_t for the input layer
                delta_t = delta_t.dot(weights[-i].T) * sigmoid_prime(hidden_layers[-i-1])
        return weights

#-----------------Don't change anything below-----------------------#

layer_sizes = np.array([X_train.shape[1]]+[128]*3+[np.squeeze(np.eye(10)[Y_train.astype(int).reshape(-1)]).shape[1]])
test_weights = init_weights(layer_sizes, fix_seed = True)
test_batch_size = 2
hidden_layers = init_layers(test_batch_size, layer_sizes)

test_output, test_hidden_layers = feed_forward(X_train[0:test_batch_size,:], hidden_layers, test_weights)
updated_weights = back_prop(test_output, y_train[0:test_batch_size, :], test_hidden_layers, test_weights, test_batch_size, 0.5)

for i, updated_weight in enumerate(updated_weights):
  print(i, "For grading purposes: ")
  print(updated_weights[i][0:2,0:5])
```

```
0 For grading purposes:
[[ 0.624641   -0.24525379 -0.14259701 -0.82629855 -0.47104275]
 [-0.40082366  0.67766191 -0.89770929 -0.3852758   0.57698234]]
1 For grading purposes:
[[ 0.7128274   0.36318954  0.65539518  0.96251201 -0.18703268]
 [ 0.96627078  0.05399214 -0.14761789 -0.7121318   0.63291713]]
2 For grading purposes:
[[ 0.62468509 -0.24525376 -0.14269261 -0.82925825 -0.4712999 ]
 [-0.4008181   0.67766192 -0.89772159 -0.38561922  0.57695223]]
3 For grading purposes:
[[ 0.62398812 -0.2454467  -0.14282968 -0.82652336 -0.47105427]
 [-0.96627555  0.3345222   0.53030067 -0.6152625  -0.98501046]]
```

Problem 5 (d) Training

Everything that's been outlined suffices for running the MLP. However, as it stands things are still quite clunkly. We can clean everything up by wrapping all this code into a Python Class.

Our original function defintions have been slightly changed to help absuse some of the properties of having a class structure. A number of new functions, such as `predict` and `evaluate`, have been defined in the `MLP` class. Carefully go through this code and try to understand exactly what's being done and why.

Complete the sigmoid, softmax, sigmoid_prime, feed_forward, back_prop and the print function at the end with the proper variables.

Problem 5 (d) Solution

```
class MLP():

    def __init__(self,X,Y,X_val,Y_val,L=1,N_l=128):
        self.X = np.concatenate((X,np.ones((X.shape[0],1))),axis=1)
        self.Y = np.squeeze(np.eye(10)[Y.astype(int).reshape(-1)])
        self.X_val = np.concatenate((X_val,np.ones((X_val.shape[0],1))),axis=1)
        self.Y_val = np.squeeze(np.eye(10)[Y_val.astype(int).reshape(-1)])
        self.L = L
        self.N_l = N_l
        self.n_samples = self.X.shape[0]
        self.layer_sizes = np.array([self.X.shape[1]]+[N_l]*L+[self.Y.shape[1]])
        self.__init_weights()
        self.train_loss = list()
        self.train_acc = list()
        self.val_loss = list()
        self.val_acc = list()
        self.train_time = list()
        self.tot_time = list()
        self.metrics = [self.train_loss,self.train_acc,self.val_loss,self.val_acc,self.train_time,self.tot_time]
```

```
    def __sigmoid(self,x):
        # VCompute the sigmoid
        return 1 / (1 + np.exp(-x))

    def __softmax(self,x):
        # Compute softmax along the rows of the input
        return np.exp(x) / np.sum(np.exp(x), axis=-1, keepdims=True)

    def __loss(self,y_pred,y):
        # Compute the loss along the rows, averaging along the number of samples
        return ((-np.log(y_pred))*y).sum(axis=1).mean()

    def __accuracy(self,y_pred,y):
        # Compute the accuracy along the rows, averaging along the number of samples
        return np.all(y_pred==y,axis=1).mean()

    def __sigmoid_prime(self,h):
        # Compute the derivative of sigmoid where h=sigmoid(x)
        return h*(1-h)

    def __to_categorical(self,x):
        # Transform probabilities into categorical predictions row-wise, by simply taking the max probability
        categorical = np.zeros((x.shape[0],self.Y.shape[1]))
        categorical[np.arange(x.shape[0]),x.argmax(axis=1)] = 1
        return categorical

    def __init_weights(self):
        # Initialize the weights of the network given the sizes of the layers
        self.weights = list()
        for i in range(self.layer_sizes.shape[0]-1):
            self.weights.append(np.random.uniform(-1,1,size=[self.layer_sizes[i],self.layer_sizes[i+1]]))


    def __init_layers(self,batch_size):

        # Initialize and allocate arrays for the hidden layer activations
        self.__h = [np.empty((batch_size,layer)) for layer in self.layer_sizes]

    def __feed_forward(self,batch):
        # Perform a forward pass of `batch` samples (N_samples x N_features)
        h_l = batch
        self.__h[0] = h_l
        for i,weights in enumerate(self.weights):
            h_l = self.__sigmoid(np.dot(h_l,weights))
            self.__h[i+1]= h_l
        self.__out = self.__softmax(self.__h[-1])

    def __back_prop(self,batch_y):
        # Update the weights of the network through back-propagation
        delta_t = (self.__out - batch_y)*self.__sigmoid_prime(self.__h[-1])
        for i in range(1,len(self.weights)+1):
            self.weights[-i] -= self.lr*(self.__h[-i-1].T.dot(delta_t))/self.batch_size
            if i < len(self.weights):  # Don't calculate delta_t for the input layer
                delta_t = delta_t.dot(self.weights[-i].T) * self.__sigmoid_prime(self.__h[-i-1])

    def predict(self,X):
        # Generate a categorical, one-hot, prediction given an input X
        X = np.concatenate((X,np.ones((X.shape[0],1))),axis=1)
        self.__init_layers(X.shape[0])
        self.__feed_forward(X)
        return self.__to_categorical(self.__out)

    def evaluate(self,X,Y):
        # Evaluate the performance (accuracy) predicting on X with true labels Y
        prediction = self.predict(X)
        return self.__accuracy(prediction,Y)

    def train(self,batch_size=8,epochs=25,lr=1.0):
        # Train the model with a given batch size, epochs, and learning rate. Store and print relevant metrics.
        self.lr = lr
        self.batch_size=batch_size
        for epoch in range(epochs):
            start = time.time()

            self.__init_layers(self.batch_size)
            shuffle = np.random.permutation(self.n_samples)
            train_loss = 0
```

```
            train_acc = 0
            X_batches = np.array_split(self.X[shuffle],self.n_samples/self.batch_size)
            Y_batches = np.array_split(self.Y[shuffle],self.n_samples/self.batch_size)
            for batch_x,batch_y in zip(X_batches,Y_batches):
                self.__feed_forward(batch_x)
                train_loss += self.__loss(self.__out,batch_y)
                train_acc += self.__accuracy(self.__to_categorical(self.__out),batch_y)
                self.__back_prop(batch_y)

            train_loss = (train_loss/len(X_batches))
            train_acc = (train_acc/len(X_batches))
            self.train_loss.append(train_loss)
            self.train_acc.append(train_acc)

            train_time = round(time.time()-start,3)
            self.train_time.append(train_time)

            self.__init_layers(self.X_val.shape[0])
            self.__feed_forward(self.X_val)
            val_loss = self.__loss(self.__out,self.Y_val)
            val_acc = self.__accuracy(self.__to_categorical(self.__out),self.Y_val)
            self.val_loss.append(val_loss)
            self.val_acc.append(val_acc)

            tot_time = round(time.time()-start,3)
            self.tot_time.append(tot_time)


            print(f"Epoch {epoch+1}: loss = {train_loss.round(3)} | acc = {train_acc.round(3)} | val_loss = {val_loss.round(3)} | val_acc =
```

Great, now let's give this a try. Let's create a really simple MLP with only a single hidden layer `L=1` with 128 neurons `N_l=128`. We'll train with a `batch_size=8` for `epochs=25` and a learning rate `lr=1.0`.

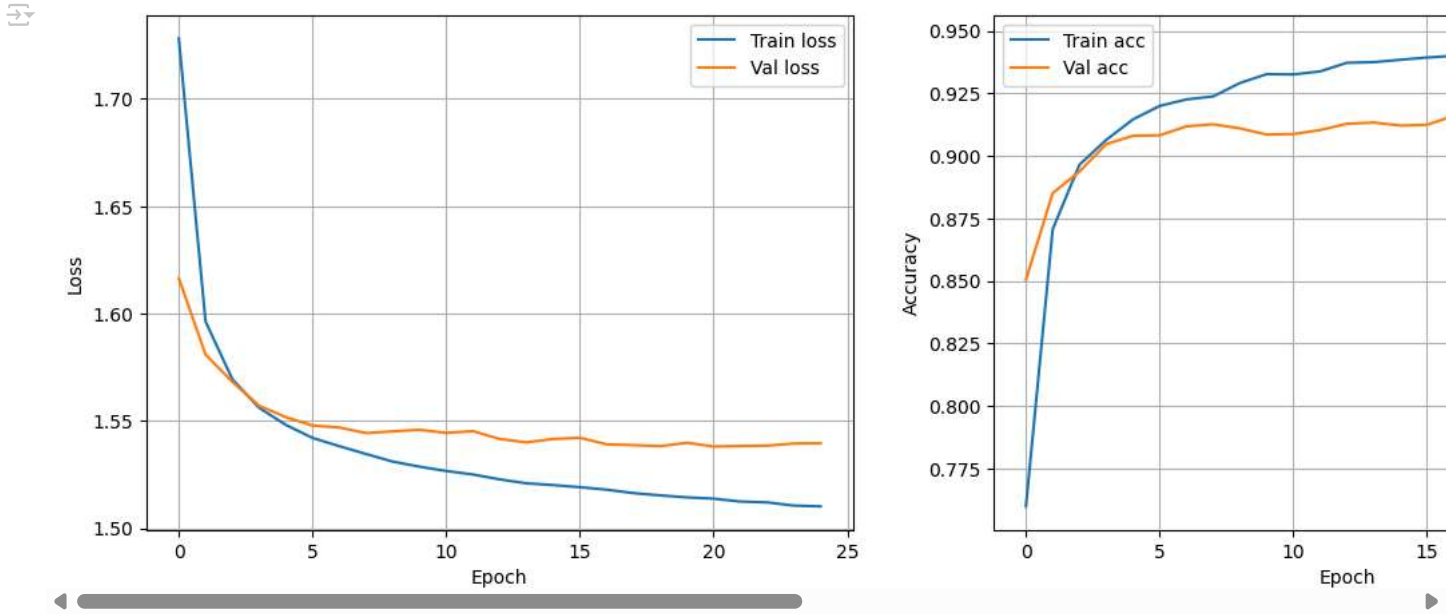
```
model = MLP(X_train,Y_train,X_test,Y_test,L=1,N_l=16)
model.train(batch_size=8,epochs=25,lr=1.0)
```

```
Epoch 1: loss = 1.728 | acc = 0.76 | val_loss = 1.616 | val_acc = 0.85 | train_time = 0.41 | tot_time = 0.435
Epoch 2: loss = 1.596 | acc = 0.871 | val_loss = 1.581 | val_acc = 0.885 | train_time = 0.511 | tot_time = 0.57
Epoch 3: loss = 1.569 | acc = 0.896 | val_loss = 1.568 | val_acc = 0.894 | train_time = 0.589 | tot_time = 0.641
Epoch 4: loss = 1.556 | acc = 0.906 | val_loss = 1.557 | val_acc = 0.905 | train_time = 0.599 | tot_time = 0.65
Epoch 5: loss = 1.548 | acc = 0.915 | val_loss = 1.552 | val_acc = 0.908 | train_time = 0.699 | tot_time = 0.737
Epoch 6: loss = 1.542 | acc = 0.92 | val_loss = 1.548 | val_acc = 0.908 | train_time = 0.533 | tot_time = 0.552
Epoch 7: loss = 1.538 | acc = 0.923 | val_loss = 1.547 | val_acc = 0.912 | train_time = 0.384 | tot_time = 0.402
Epoch 8: loss = 1.535 | acc = 0.924 | val_loss = 1.544 | val_acc = 0.913 | train_time = 0.379 | tot_time = 0.398
Epoch 9: loss = 1.531 | acc = 0.929 | val_loss = 1.545 | val_acc = 0.911 | train_time = 0.409 | tot_time = 0.428
Epoch 10: loss = 1.529 | acc = 0.933 | val_loss = 1.546 | val_acc = 0.908 | train_time = 0.375 | tot_time = 0.395
Epoch 11: loss = 1.527 | acc = 0.933 | val_loss = 1.544 | val_acc = 0.909 | train_time = 0.391 | tot_time = 0.411
Epoch 12: loss = 1.525 | acc = 0.934 | val_loss = 1.545 | val_acc = 0.91 | train_time = 0.403 | tot_time = 0.422
Epoch 13: loss = 1.523 | acc = 0.937 | val_loss = 1.542 | val_acc = 0.913 | train_time = 0.391 | tot_time = 0.413
Epoch 14: loss = 1.521 | acc = 0.937 | val_loss = 1.54 | val_acc = 0.913 | train_time = 0.406 | tot_time = 0.426
Epoch 15: loss = 1.52 | acc = 0.938 | val_loss = 1.542 | val_acc = 0.912 | train_time = 0.417 | tot_time = 0.437
Epoch 16: loss = 1.519 | acc = 0.939 | val_loss = 1.542 | val_acc = 0.912 | train_time = 0.411 | tot_time = 0.43
Epoch 17: loss = 1.518 | acc = 0.94 | val_loss = 1.539 | val_acc = 0.916 | train_time = 0.382 | tot_time = 0.401
Epoch 18: loss = 1.516 | acc = 0.942 | val_loss = 1.539 | val_acc = 0.915 | train_time = 0.389 | tot_time = 0.408
Epoch 19: loss = 1.515 | acc = 0.942 | val_loss = 1.538 | val_acc = 0.916 | train_time = 0.406 | tot_time = 0.425
Epoch 20: loss = 1.514 | acc = 0.943 | val_loss = 1.54 | val_acc = 0.913 | train_time = 0.394 | tot_time = 0.413
Epoch 21: loss = 1.514 | acc = 0.944 | val_loss = 1.538 | val_acc = 0.918 | train_time = 0.425 | tot_time = 0.445
Epoch 22: loss = 1.513 | acc = 0.944 | val_loss = 1.538 | val_acc = 0.917 | train_time = 0.376 | tot_time = 0.395
Epoch 23: loss = 1.512 | acc = 0.945 | val_loss = 1.539 | val_acc = 0.914 | train_time = 0.367 | tot_time = 0.387
Epoch 24: loss = 1.511 | acc = 0.946 | val_loss = 1.54 | val_acc = 0.914 | train_time = 0.399 | tot_time = 0.419
Epoch 25: loss = 1.51 | acc = 0.947 | val_loss = 1.54 | val_acc = 0.915 | train_time = 0.382 | tot_time = 0.401
```

Note, along with the training data, `X_train` and `Y_train`, we include a validation dataset, `X_val` and `Y_val`. The purpose of this data is to evaluate the generizability of our model. We expect our model to perform well on our training data, because of course the objective of our optimization is to minimize the error with respect to the training data, but we'd like our model to generalize to new, never before seen, data. We therefore evaluate the accuracy and loss on a hold-out set which the model never sees during training. If the performance is good on this hold-out set we can be confident that our model is generalizing well, meaning we've mananged to generally teach a computer to read hand-writing.

```
fig,ax = plt.subplots(1,2,figsize=(15,5))
ax[0].plot(model.train_loss,label="Train loss")
ax[0].plot(model.val_loss,label="Val loss")
ax[0].legend()
```

```
ax[0].set_xlabel("Epoch")
ax[0].set_ylabel("Loss")
ax[0].grid()

ax[1].plot(model.train_acc,label="Train acc")
ax[1].plot(model.val_acc,label="Val acc")
ax[1].legend()
ax[1].set_xlabel("Epoch")
ax[1].set_ylabel("Accuracy")
ax[1].grid()
```



Although the results here are impressive, the *hyperparameters* were carefully curated to achieve this good performance. The *hyperparamters* refer to the paramters of the model which are not learned during the training loop (e.g. the `batch_size`, learning rate `lr`, number of hidden layers `N_l`, number of neurons-per-layer `L`). In general, these parameters need to be carefully tuned depending on the details and idiosycrisies of each problem. This tuning process tpyically involves training a model to completion under a number of different hyperparamter settings and selecting the combination of hyperparamters which yields the highest performance (measured in this case by the accuracy on our training set).

Problem 6: 2D Convolution (5pts)

Implement `torch.conv2d` function with default settings(no padding, 1 stride etc.) using numpy. Write a function for convolution operation in the cell below. Write your own code with your own algorithm. Recall the 2D convolution can be formulated as the following.

$$C(j,k) = \sum_p \sum_q A(p,q)B(j-p+1, k-q+1)$$

Problem 6 Solution

```
import numpy as np
import torch

def my_convolution(input_numpy, kernel_numpy):
    '''
    In this function, you are required to implement the 3rd step of the function above without using torch.
    You are free to use "for" loops or vectorized code.
    '''
    # Extract dimensions
    batch_size, input_channels, input_height, input_width = input_numpy.shape
    output_channels, kernel_input_channels, kernel_height, kernel_width = kernel_numpy.shape


    # Calculate output dimensions (no padding, stride=1)
    output_height = input_height - kernel_height + 1
    output_width = input_width - kernel_width + 1
```

```
    # Initialize output array
    my_conv_output = np.zeros((batch_size, output_channels, output_height, output_width))

    # Perform convolution operation
    for b in range(batch_size):
        for oc in range(output_channels):
            for oh in range(output_height):
                for ow in range(output_width):
                    # For each position in the output, compute the weighted sum
                    for ic in range(input_channels):
                        for kh in range(kernel_height):
                            for kw in range(kernel_width):
                                ih = oh + kh
                                iw = ow + kw
                                # Using the formula: C(j,k) = ∑p∑q A(p,q) B(j-p+1, k-q+1)
                                # In our implementation, A is the kernel and B is the input
                                my_conv_output[b, oc, oh, ow] += input_numpy[b, ic, ih, iw] * kernel_numpy[oc, ic, kh, kw]

    return my_conv_output

#-----------------Don't change anything below-----------------------#

# input shape: [batch size, input_channels, input_height, input_width]
sample_input  = np.random.random([5, 3, 24, 24])

# kernel shape: [output_channels, input_channels, filter_height, filter width]
sample_kernel = np.random.random([16, 3, 5, 5])

def torch_convolution(input_numpy, kernel_numpy):
    '''
    This function;
    1) Takes numpy arrays as input and kernel,
    2) Converts them to tensors,
    3) Applies torch.conv2d,
    4) Converts the output back to numpy array
    5) Returns the output numpy array

    '''
    input_torch, kernel_torch = torch.Tensor(input_numpy), torch.Tensor(kernel_numpy)
    output_torch = torch.conv2d(input_torch, kernel_torch)

    output = output_torch.numpy()
    return output

# output_shape: [batch_size, output_channels, output_height, output_width]
torch_output = torch_convolution(sample_input, sample_kernel)
my_output    = my_convolution(sample_input, sample_kernel)

# Output shapes of torch implementation and your implementation need to match.
print(f'Torch implementation output shape: {torch_output.shape}')
print(f'Your implementation output shape: {my_output.shape}')

print('This part just for visualization of outputs for different methods.')
print('PyTorch Implementation',torch_output[0][0][0])
print('Your Implementation',my_output[0][0][0])

# You may also expect exactly equal outputs if your implementation is correct. However due to finite
# precision floating point arithmetic, results will never be exactly equal, but really close instead.
```

```
Torch implementation output shape: (5, 16, 20, 20)
Your implementation output shape: (5, 16, 20, 20)
This part just for visualization of outputs for different methods.
PyTorch Implementation [19.112411 18.960297 19.958858 20.647522 21.450832 20.22276  18.77208
 21.140997 19.563335 20.986729 20.377384 21.437555 20.18457  20.374739
 19.370749 20.210482 19.154749 19.603947 21.36853  21.992628]
Your Implementation [19.1124117  18.96029734 19.95885496 20.64752425 21.45083256 20.2227563
 18.77208267 21.14099622 19.56333293 20.98673162 20.37738367 21.43755282
 20.18457061 20.3747386  19.37075089 20.21048104 19.1547485  19.60394356
 21.36852624 21.99262734]
```

Double-click (or enter) to edit